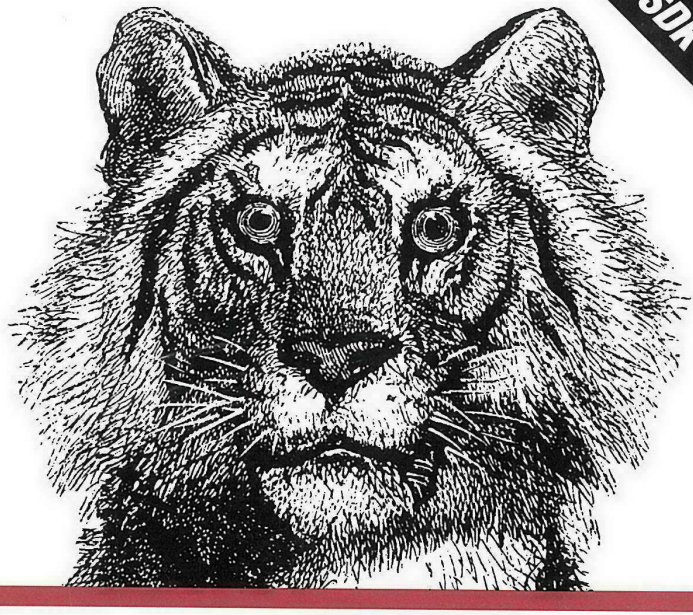
3rd Edition
Java 2/SDK 1.2 & 1.3



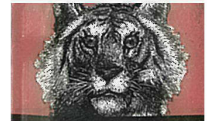
JAVA™

IN A NUTSHELL

A Desktop Quick Reference

O'REILLY®

David Flanagan



THIRD EDITION

JAVA™

JAVA 2/1.2 & 1.3

IN A NUTSHELL

FLANAGAN

O'REILLY®

JAVA™ IN A NUTSHELL

A Desktop Quick Reference

THE JAVA™ SERIES

Exploring Java™

Java™ Threads

Java™ Network Programming

Java™ Virtual Machine

Java™ AWT Reference

Java™ Language Reference

Java™ Fundamental Classes Reference

Database Programming with JDBC™ and Java™

Java™ Distributed Computing

Developing Java Beans™

Java™ Security

Java™ Cryptography

Java™ Swing

Java™ Servlet Programming

Java™ I/O

Java™ 2D Graphics

Enterprise JavaBeans™

Also from O'Reilly

Java™ in a Nutshell

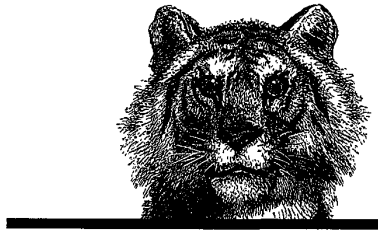
Java™ in a Nutshell, Deluxe Edition

Java™ Examples in a Nutshell

Java™ Enterprise in a Nutshell

Java™ Foundation Classes in a Nutshell

Java™ Power Reference: A Complete Searchable Resource on CD-ROM



JAVA™ IN A NUTSHELL

A Desktop Quick Reference

Third Edition

David Flanagan

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Java™ in a Nutshell, Third Edition

by David Flanagan

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editor: Paula Ferguson

Production Editor: Mary Anne Weeks Mayo

Printing History:

February 1996: First Edition.

May 1997: Second Edition.

November 1999: Third Edition.

ISBN: 1-56592-487-8 [2/00]
[M]

Table of Contents

Preface

This book is a desktop quick reference for Java™ programmers, designed to sit faithfully by your keyboard while you program. Part I of the book is a fast-paced, "no-fluff" introduction to the Java programming language and the core APIs of the Java platform. Part II is a quick-reference section that succinctly details every class and interface of those core APIs. The book covers Versions 1.0, 1.1, 1.2, and 1.3 beta of Java.

Changes Since the Second Edition

Readers who are familiar with the second edition of this book will notice a number of changes in this edition. Most notably, the AWT and applet APIs are no longer documented in this book. The Java platform tripled in size between Java 1.1 and Java 1.2. Accordingly, and unavoidably, *Java in a Nutshell* has been split into three volumes. The volume you are now reading documents only the essential APIs of the platform, including the basic language and utility classes, as well as classes for input/output, networking, and security. See the Table of Contents for a complete list of the packages documented here.

If you are a client-side programmer who is working with graphics or graphical user interfaces, you will probably want to supplement this book with *Java Foundation Classes in a Nutshell*, which documents all the graphics- and GUI-related classes, including the AWT, Swing, Java 2D, and applet APIs. And, if you are an server-side or enterprise programmer, you will likely be interested in *Java Enterprise in a Nutshell.*

Another big change is that Part I of this book has been almost entirely rewritten. The first and second editions of this book assumed knowledge of and experience with C or C++. Now that Java has come thoroughly into its own, that assumption no longer seems appropriate, so I have rewritten and expanded Chapters 2 and 3 to explain Java without any reference to C, C++, or any other programming language. Programmers with a modest amount of experience should now be able to learn Java programming from this book. These introductory chapters are written in

a tight, concise style, so programmers who already know Java should find them useful as a language reference.

Another new feature of Part I is Chapter 4, *The Java Platform.* This chapter is an introduction to the APIs documented in the reference section of the book. It includes more than 60 detailed API usage examples that show how to accomplish common tasks with the predefined classes of the Java platform.

Finally, the quick-reference section in Part II of the book has a new look that dramatically improves the readability of the reference material and packs even more API information into a small space. Even if you are already familiar with the second edition, you should take the time to read the *How To Use This Quick Reference* section at the beginning of Part II; it explains the new quick-reference format and shows you how to get the most out of it.

Contents of This Book

The first eight chapters of this book document the Java language, the Java platform, and the Java development tools that are supplied with Sun's Java SDK (software development kit):

Chapter 1: Introduction
: This chapter is an overview of the Java language and the Java platform that explains the important features and benefits of Java. It concludes with an example Java program and walks the new Java programmer through it line by line.

Chapter 2: Java Syntax From the Ground Up
: This chapter explains the details of the Java programming language. It is a long and detailed chapter. Experienced Java programmers can use it as a language reference. Programmers with substantial experience with languages such as C and C++ should be able to pick up Java syntax by reading this chapter. The chapter does not assume years of programming experience, or even familiarity, with C or C++, however. Even beginning programmers, with only modest experience, should be able to learn Java programming by studying this chapter carefully.

Chapter 3: Object-Oriented Programming in Java
: This chapter describes how the basic Java syntax documented in Chapter 2 is used to write object-oriented programs in Java. The chapter assumes no prior experience with OO programming. It can be used as a tutorial by new programmers or as a reference by experienced Java programmers.

Chapter 4: The Java Platform
: This chapter is an overview of the essential Java APIs covered in this book. It contains numerous short examples that demonstrate how to perform common tasks with the classes and interfaces that comprise the Java platform. Programmers who are new to Java, and especially those who learn best by example, should find this a valuable chapter.

Chapter 5: Java Security
: This chapter explains the Java security architecture that allows untrusted code to run in a secure environment from which it cannot do any malicious damage to the host system. It is important for all Java programmers to have at least a passing familiarity with Java security mechanisms.

Chapter 6: JavaBeans
: This chapter documents the JavaBeans™ component framework and explains what programmers need to know to create and use the reusable, embeddable Java classes known as beans.

Chapter 7: Java Programming and Documentation Conventions
: This chapter documents important and widely adopted Java programming conventions and also explains how you can make your Java code self-documenting by including specially formatted documentation comments.

Chapter 8: Java Development Tools
: The Java SDK shipped by Sun includes a number of useful Java development tools, most notably the Java interpreter and the Java compiler. This chapter documents those tools.

These first eight chapters teach you the Java language and get you up and running with the Java APIs. The bulk of the book, however, is the API quick reference, Chapters 9 through 29, which is a succinct but detailed API reference formatted for optimum ease of use. Please be sure to read the *How To Use This Quick Reference* section, which appears at the beginning of the reference section; it explains how to get the most out of this section.

Related Books

O'Reilly & Associates, Inc. publishes an entire series of books on Java programming. These books include *Java Foundation Classes in a Nutshell* and *Java Enterprise in a Nutshell*, which, as mentioned earlier, are companions to this book.

A related reference work is the *Java Power Reference*. It is an electronic Java quick reference on CD-ROM that uses the *Java in a Nutshell* style. But since it is designed for viewing in a web browser, it is fully hyperlinked and includes a powerful search engine. It is wider in scope but narrower in depth than the *Java in a Nutshell* books. The *Java Power Reference* covers all the APIs of the Java 2 platform, plus the APIs of many standard extensions. But it does not include tutorial chapters on the various APIs, nor does it include descriptions of the individual classes.

You can find a complete list of Java books from O'Reilly & Associates at *http://java.oreilly.com/*. Books that focus on the core Java APIs, as this one does, include:

Exploring Java, by Pat Niemeyer and Joshua Peck
: A comprehensive tutorial introduction to Java, with an emphasis on client-side Java programming.

Java Threads, by Scott Oaks and Henry Wong
Java makes multithreaded programming easy, but doing it right can still be tricky. This book explains everything you need to know.

Java I/O, by Elliotte Rusty Harold
Java's stream-based input/output architecture is a thing of beauty. This book covers it in the detail it deserves.

Java Network Programming, by Elliotte Rusty Harold
This book documents the Java networking APIs in detail.

Java Security, by Scott Oaks
This book explains the Java access-control mechanisms in detail and also documents the authentication mechanisms of digital signatures and message digests.

Java Cryptography, by Jonathan Knudsen
Thorough coverage of the Java Cryptography Extension, the `javax.crypto.*` packages, and everything you need to know about cryptography in Java.

Developing Java Beans, by Robert Englander
A complete guide to writing components that work with the JavaBeans API.

Java Programming Resources Online

This book is a quick reference designed for speedy access to frequently needed information. It does not, and cannot, tell you everything you need to know about Java. In addition to the books listed earlier, there are several valuable (and free) electronic sources of information about Java programming.

Sun's main web site for all things related to Java is *http://java.sun.com/*. The web site specifically for Java developers is *http://developer.java.sun.com/*. Much of the content on this developer site is password-protected, and access to it requires (free) registration.

Sun distributes electronic documentation for all Java classes and methods in its *javadoc* HTML format. Although this documentation is somewhat difficult to navigate and is rough or outdated in places, it is still an excellent starting point when you need to know more about a particular Java package, class, method, or field. If you do not already have the *javadoc* files with your Java distribution, see *http://java.sun.com/docs/* for a link to the latest available version. Sun also distributes its excellent *Java Tutorial* online. You can browse and download it from *http://java.sun.com/docs/books/tutorial/*.

For Usenet discussion (in English) about Java, try the *comp.lang.java.programmer* and related *comp.lang.java.** newsgroups. You can find the very comprehensive *comp.lang.java.programmer* FAQ by Peter van der Linden at *http://www.afu.com/javafaq.htm*.

Finally, don't forget O'Reilly's Java web site. *http://java.oreilly.com/* contains Java news and commentary and a monthly tips-and-tricks column by O'Reilly Java author Jonathan Knudsen.

Examples Online

The examples in this book are available online and can be downloaded from the home page for the book at *http://www.oreilly.com/catalog/javanut3*. You also may want to visit this site to see if any important notes or errata about the book have been published there.

Conventions Used in This Book

We use the following formatting conventions in this book:

Italic
: Used for emphasis and to signify the first use of a term. Italic is also used for commands, email addresses, web sites, FTP sites, file and directory names, and newsgroups.

Bold
: Occasionally used to refer to particular keys on a computer keyboard or to portions of a user interface, such as the **Back** button or the **Options** menu.

`Letter Gothic`
: Used in all Java code and generally for anything that you would type literally when programming, including keywords, data types, constants, method names, variables, class names, and interface names.

`Letter Gothic Oblique`
: Used for the names of function arguments and generally as a placeholder to indicate an item that should be replaced with an actual value in your program.

Franklin Gothic Book Condensed
: Used for the Java class synopses in the quick-reference section. This very narrow font allows us to fit a lot of information on the page without a lot of distracting line breaks. This font is also used for code entities in the descriptions in the quick-reference section.

Franklin Gothic Demi Condensed
: Used for highlighting class, method, field, property, and constructor names in the quick-reference section, which makes it easier to scan the class synopses.

Franklin Gothic Book Compressed Italic
: Used for method parameter names and comments in the quick-reference section.

Request for Comments

Please help us improve future editions of this book by reporting any errors, inaccuracies, bugs, misleading or confusing statements, and even plain old typos that you find. Please also let us know what we can do to make this book more useful to you. We take your comments seriously and will try to incorporate reasonable suggestions into future editions. You can contact us by writing:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
1-800-998-9938 (in the United States or Canada)
1-707-829-0515 (international/local)
1-707-829-0104 (fax)

You can also send us messages electronically. To be put on the mailing list or request a catalog, send email to:

info@oreilly.com

To ask technical questions or comment on the book, send email to:

bookquestions@oreilly.com

We have a web site for the book, where we'll list examples, errata, and any plans for future editions. You can access this page at:

http://www.oreilly.com/catalog/javanut3/

For more information about this book and others, see the O'Reilly web site:

http://www.oreilly.com

How the Quick Reference Is Generated

For the nerdy or merely inquisitive reader, this section explains a bit about how the quick-reference material in *Java in a Nutshell* and related books is created.

As Java has evolved, so has my system for generating Java quick-reference material. The current system is part of a larger commercial documentation browser system I'm developing (visit *http://www.davidflanagan.com/Jude/* for more information about it). The program works in two passes: the first pass collects and organizes the API information, and the second pass outputs that information in the form of quick-reference chapters.

The first pass begins by reading the class files for all of the classes and interfaces to be documented. Almost all of the API information in the quick reference is available in these class files. The notable exception is the names of method arguments, which are not stored in class files. These argument names are obtained by parsing the Java source file for each class and interface. Where source files are not available, I obtain method argument names by parsing the API documentation generated by *javadoc*. The parsers I use to extract API information from the source files and *javadoc* files are created using the Antlr parser generator developed by Terrence Parr of the Magelang Institute. (See *http://www.antlr.org/* for details on this very powerful programming tool.)

Once the API information has been obtained by reading class files, source files, and *javadoc* files, the program spends some time sorting and cross-referencing everything. Then it stores all the API information into a single large data file.

The second pass reads API information from that data file and outputs quick-reference chapters using a custom SGML format. The SGML markup is fairly complex, but the code that generates it is quite mundane. Once I've generated the SGML

output, I hand it off to the production team at O'Reilly & Associates. They process it and convert it to troff source code. The troff source is processed with the GNU *groff* program (*ftp://ftp.gnu.org/gnu/groff/*) and a custom set of troff macros to produce PostScript output that is shipped directly to the printer.

Acknowledgments

Many people helped in the creation of this book, and I am grateful to them all. I am indebted to the many, many readers of the first two editions who wrote in with comments, suggestions, bug reports, and praise. Their many small contributions are scattered throughout the book. Also, my apologies to those who made the many good suggestions that could not be incorporated into this edition.

Paula Ferguson, a friend and colleague, has been the editor of all three editions of this book. Her careful reading and always-practical suggestions have made the book stronger, clearer, and more useful. She guided the evolution of *Java in a Nutshell* from a single book into a three-volume series and, at times, juggled editing tasks for all three books at once. Finally, Paula went above and beyond the call of editorial duty by designing the hierarchy diagrams found at the start of each reference chapter.

Mike Loukides provided high-level direction and guidance for the first edition of the book. Eric Raymond and Troy Downing reviewed that first edition—they helped spot my errors and omissions and offered good advice on making the book more useful to Java programmers.

For the second edition, John Zukowski reviewed my Java 1.1 AWT quick-reference material, and George Reese reviewed most of the remaining new material. The second edition was also blessed with a "dream team" of technical reviewers from Sun. John Rose, the author of the Java inner class specification, reviewed the chapter on inner classes. Mark Reinhold, author of the new character stream classes in `java.io`, reviewed my documentation of these classes. Nakul Saraiya, the designer of the new Java Reflection API, reviewed my documentation of the `java.lang.reflect` package. I am very grateful to these engineers and architects; their efforts made this a stronger, more accurate book.

The third edition also benefited greatly from the contributions of reviewers who are intimately familiar with the Java platform. Joshua Bloch, one of the primary authors of the Java collections framework, reviewed my descriptions of the collections classes and interfaces. Joshua was also helpful in discussing the `Timer` and `TimerTask` classes of Java 1.3 with me. Mark Reinhold, creator of the `java.lang.ref` package, explained the package to me and reviewed my documentation of it. Scott Oaks reviewed my descriptions of the Java security and cryptography classes and interfaces. Joshua, Mark, and Scott are all engineers with Sun Microsystems, and I'm very grateful for their time. The documentation of the `javax.crypto` package and its subpackages was also reviewed by Jon Eaves. Jon worked on a clean-room implementation of the Java Cryptography Extension (which is available from *http://www.aba.net.au/*), and his comments were quite helpful. Jon now works for Fluent Technologies (*http://www.fluent.com.au/*) consulting in Java and electronic commerce. Finally, Chapter 1 was improved by the comments of reviewers who were *not* already familiar with the Java platform:

Christina Byrne reviewed it from the standpoint of a novice programmer, and Judita Byrne of Virginia Power offered her comments as a professional COBOL programmer.

The O'Reilly & Associates production team has done its usual fine work of creating a book out of the electronic files I submit. My thanks to them all. And a special thanks to Lenny Muellner and Chris Maden, who worked overtime to implement the new and improved format of the quick-reference section.

As always, my thanks and love to Christie.

David Flanagan
http://www.davidflanagan.com/
September 1999

PART I

Introducing Java

Part I is an introduction to the Java language and the Java platform. These chapters provide enough information for you to get started using Java right away.

CHAPTER 1

Introduction

Welcome to Java. Since its introduction in late 1995, the Java language and platform have taken the programming world by storm. This chapter begins by explaining what Java is and why it has become so popular. Then, as a tutorial introduction to the language, it walks you through a simple Java program you can type in, compile, and run.

What Is Java?

In discussing Java, it is important to distinguish between the Java programming language, the Java Virtual Machine, and the Java platform. The Java programming language is the language in which Java applications (including applets, servlets, and JavaBeans components) are written. When a Java program is compiled, it is converted to byte codes that are the portable machine language of a CPU architecture known as the Java Virtual Machine (also called the Java VM or JVM). The JVM can be implemented directly in hardware, but it is usually implemented in the form of a software program that interprets and executes byte codes.

The Java platform is distinct from both the Java language and Java VM. The Java platform is the predefined set of Java classes that exist on every Java installation; these classes are available for use by all Java programs. The Java platform is also sometimes referred to as the Java runtime environment or the core Java APIs (application programming interfaces). The Java platform can be extended with optional standard extensions. These extension APIs exist in some Java installations, but are not guaranteed to exist in all installations.

The Java Programming Language

The Java programming language is a state-of-the-art, object-oriented language that has a syntax similar to that of C. The language designers strove to make the Java language powerful, but, at the same time, they tried to avoid the overly complex features that have bogged down other object-oriented languages, such as C++. By keeping the language simple, the designers also made it easier for programmers to

write robust, bug-free code. As a result of its elegant design and next-generation features, the Java language has proved wildly popular with programmers, who typically find it a pleasure to work with Java after struggling with more difficult, less powerful languages.

The Java Virtual Machine

The Java Virtual Machine, or Java interpreter, is the crucial piece of every Java installation. By design, Java programs are portable, but they are only portable to platforms to which a Java interpreter has been ported. Sun ships VM implementations for its own Solaris operating system and for Microsoft Windows (95/98/NT) platforms. Many other vendors, including Apple and various Unix vendors, provide Java interpreters for their platforms. There is a freely available port of Sun's VM for Linux platforms, and there are also other third-party VM implementations available. The Java VM is not only for desktop systems, however. It has been ported to set-top boxes, and versions are even available for hand-held devices that run Windows CE and PalmOS.

Although interpreters are not typically considered high-performance systems, Java VM performance is remarkably good and has been improving steadily. Of particular note is a VM technology called just-in-time (JIT) compilation, whereby Java byte codes are converted on-the-fly into native-platform machine language, boosting execution speed for code that is run repeatedly. Sun's new Hotspot technology is a particularly good implementation of JIT compilation.

The Java Platform

The Java platform is just as important as the Java programming language and the Java Virtual Machine. All programs written in the Java language rely on the set of predefined classes* that comprise the Java platform. Java classes are organized into related groups known as *packages*. The Java platform defines packages for functionality such as input/output, networking, graphics, user-interface creation, security, and much more.

The Java 1.2 release was a major milestone for the Java platform. This release almost tripled the number of classes in the platform and introduced significant new functionality. In recognition of this, Sun named the new version the Java 2 Platform. This is a trademarked name created for marketing purposes; it serves to emphasize how much Java has grown since its first release. However, most programmers refer to the Java platform by its official version number, which, at the time of this writing, is 1.2.†

It is important to understand what is meant by the term platform. To a computer programmer, a platform is defined by the APIs he or she can rely on when writing programs. These APIs are usually defined by the operating system of the target computer. Thus, a programmer writing a program to run under Microsoft Windows

---

* A *class* is a module of Java code that defines a data structure and a set of methods (also called procedures, functions, or subroutines) that operate on that data.

† Although there is currently a beta release of Java 1.3 available

must use a different set of APIs than a programmer writing the same program for the Macintosh or for a Unix-based system. In this respect, Windows, Macintosh, and Unix are three distinct platforms.

Java is not an operating system.* Nevertheless, the Java platform—particularly the Java 2 Platform—provides APIs with a comparable breadth and depth to those defined by an operating system. With the Java 2 Platform, you can write applications in Java without sacrificing the advanced features available to programmers writing native applications targeted at a particular underlying operating system. An application written on the Java platform runs on any operating system that supports the Java platform. This means you do not have to create distinct Windows, Macintosh, and Unix versions of your programs, for example. A single Java program runs on all these operating systems, which explains why "Write once, run anywhere" is Sun's motto for Java.

It also explains why companies like Microsoft might feel threatened by Java. The Java platform is not an operating system, but for programmers, it is an alternative development target and a very popular one at that. The Java platform reduces programmers' reliance on the underlying operating system, and, by allowing programs to run on top of any operating system, it increases end users' freedom to choose an operating system.

Versions of Java

As of this writing, there have been four major versions of Java. They are:

Java 1.0
: This was the first public version of Java. It contained 212 classes organized in 8 packages. There is a large installed base of web browsers that run this version of Java, so this version is still in use for writing simple applets—Java programs that are included in web pages. (See *Java Foundation Classes in a Nutshell* (O'Reilly) for a discussion of applets.)

Java 1.1
: This release of Java doubled the size of the Java platform to 504 classes in 23 packages. It introduced inner classes, an important change to the Java language itself, and included significant performance improvements in the Java VM. This version of Java is out of date, but is still in use on systems that do not yet have a stable port of Java 1.2.

Java 1.2
: This is the latest and greatest significant release of Java; it tripled the size of the Java platform to 1520 classes in 59 packages. Because of the many new features included in this release, the platform was renamed and is now called the Java 2 Platform.

* There is a Java-based operating system, however; it is known as JavaOS.

Java 1.3 (beta)

This release includes minor corrections and updates to the Java platform, but does not include major changes or significant new functionality.

In addition, Sun has instituted a process for proposing and developing standard extensions to the Java platform. In the future, most new functionality is expected to take the form of a standard extension, rather than be a required part of every Java installation.

In order to work with Java 1.0 or Java 1.1, you have to obtain the Java Development Kit (JDK) for that release. As of Java 1.2, the JDK has been renamed and is now called a Software Development Kit (SDK), so we have the Java 2 SDK or, more precisely, the Java 2 SDK, Standard Edition, Version 1.2 (or Version 1.3 beta). Despite the new name, many programmers still refer to the development kit as the JDK.

Don't confuse the JDK (or SDK) with the Java Runtime Environment (JRE). The JRE contains everything you need to run Java programs, but does not contain the tools you need to develop Java programs (i.e., the compiler). You should also be aware of the Java Plug-in, a version of the Java 1.2 (and 1.3) JRE that is designed to be integrated into the Netscape Navigator and Microsoft Internet Explorer web browsers.

In addition to evolving the Java platform over time, Sun is also trying to produce different versions of the platform for different uses. The Standard Edition is the only version currently available, but Sun is also working on the Java 2 Platform, Enterprise Edition (J2EE), for enterprise developers and the Java 2 Platform, Micro Edition, for consumer electronic systems, like handheld PDAs and cellular telephones.

Key Benefits of Java

Why use Java at all? Is it worth learning a new language and a new platform? This section explores some of the key benefits of Java.

Write Once, Run Anywhere

Sun identifies "Write once, run anywhere" as the core value proposition of the Java platform. Translated from business jargon, this means that the most important promise of Java technology is that you only have to write your application once—for the Java platform—and then you'll be able to run it *anywhere*.

Anywhere, that is, that supports the Java platform. Fortunately, Java support is becoming ubiquitous. It is integrated, or being integrated, into practically all major operating systems. It is built into the popular web browsers, which places it on virtually every Internet-connected PC in the world. It is even being built into consumer electronic devices, such as television set-top boxes, PDAs, and cell phones.

Security

Another key benefit of Java is its security features. Both the language and the platform were designed from the ground up with security in mind. The Java platform allows users to download untrusted code over a network and run it in a secure environment in which it cannot do any harm: it cannot infect the host system with a virus, cannot read or write files from the hard drive, and so forth. This capability alone makes the Java platform unique.

The Java 2 Platform takes the security model a step further. It makes security levels and restrictions highly configurable and extends them beyond applets. As of Java 1.2, any Java code, whether it is an applet, a servlet, a JavaBeans component, or a complete Java application, can be run with restricted permissions that prevent it from doing harm to the host system.

The security features of the Java language and platform have been subjected to intense scrutiny by security experts around the world. Security-related bugs, some of them potentially serious, have been found and promptly fixed. Because of the security promises Java makes, it is big news when a new security bug is found. Remember, however, that no other mainstream platform can make security guarantees nearly as strong as those Java makes. If Java's security is not yet perfect, it has been proven strong enough for practical day-to-day use and is certainly better than any of the alternatives.

Network-centric Programming

Sun's corporate motto has always been "The network is the computer." The designers of the Java platform believed in the importance of networking and designed the Java platform to be network-centric. From a programmer's point of view, Java makes it unbelievably easy to work with resources across a network and to create network-based applications using client/server or multitier architectures. This means that Java programmers have a serious head start in the emerging network economy.

Dynamic, Extensible Programs

Java is both dynamic and extensible. Java code is organized in modular object-oriented units called *classes*. Classes are stored in separate files and are loaded into the Java interpreter only when needed. This means that an application can decide as it is running what classes it needs and can load them when it needs them. It also means that a program can dynamically extend itself by loading the classes it needs to expand its functionality.

The network-centric design of the Java platform means that a Java application can dynamically extend itself by loading new classes over a network. An application that takes advantage of these features ceases to be a monolithic block of code. Instead, it becomes an interacting collection of independent software components. Thus, Java enables a powerful new metaphor of application design and development.

Internationalization

The Java language and the Java platform were designed from the start with the rest of the world in mind. Java is the only commonly used programming language that has internationalization features at its very core, rather than tacked on as an afterthought. While most programming languages use 8-bit characters that represent only the alphabets of English and Western European languages, Java uses 16-bit Unicode characters that represent the phonetic alphabets and ideographic character sets of the entire world. Java's internationalization features are not restricted to just low-level character representation, however. The features permeate the Java platform, making it easier to write internationalized programs with Java than it is with any other environment.

Performance

As I described earlier, Java programs are compiled to a portable intermediate form known as byte codes, rather than to native machine-language instructions. The Java Virtual Machine runs a Java program by interpreting these portable byte-code instructions. This architecture means that Java programs are faster than programs or scripts written in purely interpreted languages, but they are typically slower than C and C++ programs compiled to native machine language. Keep in mind, however, that although Java programs are compiled to byte code, not all of the Java platform is implemented with interpreted byte codes. For efficiency, computationally intensive portions of the Java platform—such as the string-manipulation methods—are implemented using native machine code.

Although early releases of Java suffered from performance problems, the speed of the Java VM has improved dramatically with each new release. The VM has been highly tuned and optimized in many significant ways. Furthermore, many implementations include a just-in-time compiler, which converts Java byte codes to native machine instructions on the fly. Using sophisticated JIT compilers, Java programs can execute at speeds comparable to the speeds of native C and C++ applications.

Java is a portable, interpreted language; Java programs run almost as fast as native, non-portable C and C++ programs. Performance used to be an issue that made some programmers avoid using Java. Now, with the improvements made in Java 1.2, performance issues should no longer keep anyone away. In fact, the winning combination of performance plus portability is a unique feature no other language can offer.

Programmer Efficiency and Time-to-Market

The final, and perhaps most important, reason to use Java is that programmers like it. Java is an elegant language combined with a powerful and well-designed set of APIs. Programmers enjoy programming in Java and are usually amazed at how quickly they can get results with it. Studies have consistently shown that switching to Java increases programmer efficiency. Because Java is a simple and elegant language with a well-designed, intuitive set of APIs, programmers write better code with fewer bugs than for other platforms, again reducing development time.

An Example Program

Example 1-1 shows a Java program to compute factorials.* The numbers at the beginning of each line are not part of the program; they are there for ease of reference when we dissect the program line-by-line.

Example 1-1: Factorial.java: A Program to Compute Factorials

```
 1 /**
 2  * This program computes the factorial of a number
 3  */
 4 public class Factorial {                    // Define a class
 5   public static void main(String[] args) { // The program starts here
 6     int input = Integer.parseInt(args[0]); // Get the user's input
 7     double result = factorial(input);      // Compute the factorial
 8     System.out.println(result);            // Print out the result
 9   }                                        // The main() method ends here
10
11   public static double factorial(int x) {  // This method computes x!
12     if (x < 0)                             // Check for bad input
13       return 0.0;                          //   if bad, return 0
14     double fact = 1.0;                     // Begin with an initial value
15     while(x > 1) {                         // Loop until x equals 1
16       fact = fact * x;                     //   multiply by x each time
17       x = x - 1;                           //   and then decrement x
18     }                                      // Jump back to start of loop
19     return fact;                           // Return the result
20   }                                        // factorial() ends here
21 }                                          // The class ends here
```

Compiling and Running the Program

Before we look at how the program works, we must first discuss how to run it. In order to compile and run the program, you need a Java software development kit (SDK) of some sort. Sun Microsystems created the Java language and ships a free Java SDK for its Solaris operating system and for Microsoft Windows (95/98/NT) platforms. At the time of this writing, the current version of Sun's SDK is entitled Java 2 SDK, Standard Edition, Version 1.2.2 and is available for download from *http://java.sun.com/products/jdk/1.2/* (Sun's Java SDK is still often called the JDK, even internally). Be sure to get the SDK and not the Java Runtime Environment. The JRE enables you to run existing Java programs, but not to write your own.

Sun supports its SDK only on Solaris and Windows platforms. Many other companies have licensed and ported the SDK to their platforms, however. Contact your operating-system vendor to find if a version of the Java SDK is available for your system. Linux users should visit *http://www.blackdown.org/*.

The Sun SDK is not the only Java programming environment you can use. Companies such as Borland, Inprise, Metrowerks, Oracle, Sybase, and Symantec offer commercial products that enable you to write Java programs. This book assumes

* The factorial of an integer is the product of the number and all positive integers less than the number. So, for example, the factorial of 4, which is also written 4!, is 4 times 3 times 2 times 1, or 24. By definition, 0! is 1.

that you are using Sun's SDK. If you are using a product from some other vendor, be sure to read that vendor's documentation to learn how to compile and run a simple program, like that shown in Example 1-1.

Once you have a Java programming environment installed, the first step towards running our program is to type it in. Using your favorite text editor, enter the program as it is shown in Example 1-1. Omit the line numbers, as they are just there for reference. Note that Java is a case-sensitive language, so you must type lowercase letters in lowercase and uppercase letters in uppercase. You'll notice that many of the lines of this program end with semicolons. It is a common mistake to forget these characters, but the program won't work without them, so be careful! If you are not a fast typist, you can omit everything from // to the end of a line. Those are *comments*; they are there for your benefit and are ignored by Java.*

When writing Java programs, you should use a text editor that saves files in plain-text format, not a word processor that supports fonts and formatting and saves files in a proprietary format. My favorite text editor on Unix systems is *emacs*. If you use a Windows system, you might use *Notepad* or *WordPad*, if you don't have a more specialized programmer's editor. If you are using a commercial Java programming environment, it probably includes an appropriate text editor; read the documentation that came with the product. When you are done entering the program, save it in a file named *Factorial.java*. This is important; the program will not work if you save it by any other name.

After writing a program like this one, the next step is to compile it. With Sun's SDK, the Java compiler is known as *javac*. *javac* is a command-line tool, so you can only use it from a terminal window, such as an MS-DOS window on a Windows system or an *xterm* window on a Unix system. Compile the program by typing the following command line:†

```
C:\> javac Factorial.java
```

If this command prints any error messages, you probably got something wrong when you typed in the program. If it does not print any error messages, however, the compilation has succeeded, and *javac* creates a file called *Factorial.class*. This is the compiled version of the program.

Once you have compiled a Java program, you must still run it. Unlike some other languages, Java programs are not compiled into native machine language, so they cannot be executed directly by the system. Instead, they are run by another program known as the Java interpreter. In Sun's SDK, the interpreter is a command-line program named, appropriately enough, *java*. To run the factorial program, type:

```
C:\> java Factorial 4
```

java is the command to run the Java interpreter, *Factorial* is the name of the Java program we want the interpreter to run, and *4* is the input data—the number we

---

* I recommend that you type this example in by hand, to get a feel for the language. If you *really* don't want to, however, you can download this, and all examples in the book, from *http://www.oreilly.com/catalog/javanut3/*.

† The "C:\>" characters represent the command-line prompt; *don't* type these characters yourself.

want the interpreter to compute the factorial of. The program prints a single line of output, telling us that the factorial of 4 is 24:

```
C:\> java Factorial 4
24.0
```

Congratulations! You've just written, compiled, and run your first Java program. Try running it again to compute the factorials of some other numbers.

Analyzing the Program

Now that you have run the factorial program, let's analyze it line by line, to see what makes a Java program tick.

Comments

The first three lines of the program are a comment. Java ignores them, but they tell a human programmer what the program does. A comment begins with the characters /* and ends with the characters */. Any amount of text, including multiple lines of text, may appear between these characters. Java also supports another type of comment, which you can see in lines 4 through 21. If the characters // appear in a Java program, Java ignores those characters and any other text that appears between those characters and the end of the line.

Defining a class

Line 4 is the beginning of the program. It says that we are defining a class named `Factorial`. This explains why the program had to be stored in a file named *Factorial.java*. That filename indicates that the file contains Java source code for a class named `Factorial`. The word `public` is a *modifier*; it says that the class is publicly available and that anyone may use it. The open curly-brace character ({) marks the beginning of the body of the class, which extends all the way to line 21, where we find the matching close curly-brace character (}). The program contains a number of pairs of curly braces; the lines are indented to show the nesting within these braces.

A class is the fundamental unit of program structure in Java, so it is not surprising that the first line of our program declares a class. All Java programs are classes, although some programs use many classes instead of just one. Java is an object-oriented programming language, and classes are a fundamental part of the object-oriented paradigm. Each class defines a unique kind of object. Example 1-1 is not really an object-oriented program, however, so I'm not going to go into detail about classes and objects here. That is the topic of Chapter 3, *Object-Oriented Programming in Java*. For now, all you need to understand is that a class defines a set of interacting *members*. Those members may be fields, methods, or other classes. The `Factorial` class contains two members, both of which are methods. They are described in upcoming sections.

Defining a method

Line 5 begins the definition of a *method* of our `Factorial` class. A method is a named chunk of Java code. A Java program can call, or *invoke*, a method to execute the code in it. If you have programmed in other languages, you have probably seen methods before, but they may have been called functions, procedures, or subroutines. The interesting thing about methods is that they have *parameters* and *return values*. When you call a method, you pass it some data you want it to operate on, and it returns a result to you. A method is like an algebraic function:

```
y = f(x)
```

Here, the mathematical function `f` performs some computation on the value represented by `x` and returns a value, which we represent by `y`.

To return to line 5, the `public` and `static` keywords are modifiers. `public` means the method is publicly accessible; anyone can use it. The meaning of the `static` modifier is not important here; it is explained in Chapter 3. The `void` keyword specifies the return value of the method. In this case, it specifies that this method does not have a return value.

The word `main` is the name of the method. `main` is a special name. When you run the Java interpreter, it reads in the class you specify, then looks for a method named `main()`.* When the interpreter finds this method, it starts running the program at that method. When the `main()` method finishes, the program is done, and the Java interpreter exits. In other words, the `main()` method is the main entry point into a Java program. It is not actually sufficient for a method to be named `main()`, however. The method must be declared `public static void` exactly as shown in line 5. In fact, the only part of line 5 you can change is the word `args`, which you can replace with any word you want. You'll be using this line in all of your Java programs, so go ahead and commit it to memory now!†

Following the name of the `main()` method is a list of method parameters, contained in parentheses. This `main()` method has only a single parameter. `String[]` specifies the type of the parameter, which is an array of strings (i.e., a numbered list of strings of text). `args` specifies the name of the parameter. In the algebraic equation `f(x)`, `x` is simply a way of referring to an unknown value. `args` serves the same purpose for the `main()` method. As we'll see, the name `args` is used in the body of the method to refer to the unknown value that is passed to the method.

As I've just explained, the `main()` method is a special one that is called by the Java interpreter when it starts running a Java class (program). When you invoke the Java interpreter like this:

---

* By convention, when this book refers to a method, it follows the name of the method by a pair of parentheses. As you'll see, parentheses are an important part of method syntax, and they serve here to keep method names distinct from the names of classes, fields, variables, and so on.

† All Java programs that are run directly by the Java interpreter must have a `main()` method. Programs of this sort are often called *applications*. It is possible to write programs that are not run directly by the interpreter, but are dynamically loaded into some other already running Java program. Examples are *applets*, which are programs run by a web browser, and *servlets*, which are programs run by a web server. Applets are discussed in *Java Foundation Classes in a Nutshell* (O'Reilly), while servlets are discussed in *Java Enterprise in a Nutshell* (O'Reilly) In this book, we consider only applications.

```
C:\> java Factorial 4
```

the string "4" is passed to the `main()` method as the value of the parameter named `args`. More precisely, an array of strings containing only one entry, "4", is passed to `main()`. If we invoke the program like this:

```
C:\> java Factorial 4 3 2 1
```

then an array of four strings, "4", "3", "2", and "1", are passed to the `main()` method as the value of the parameter named `args`. Our program looks only at the first string in the array, so the other strings are ignored.

Finally, the last thing on line 5 is an open curly brace. This marks the beginning of the body of the `main()` method, which continues until the matching close curly brace on line 9. Methods are composed of *statements*, which the Java interpreter executes in sequential order. In this case, lines 6, 7, and 8 are three statements that compose the body of the `main()` method. Each statement ends with a semicolon to separate it from the next. This is an important part of Java syntax; beginning programmers often forget the semicolons.

Declaring a variable and parsing input

The first statement of the `main()` method, line 6, declares a variable and assigns a value to it. In any programming language, a *variable* is simply a symbolic name for a value. Think back to algebra class again:

$$c^2 = a^2 + b^2$$

The letters `a`, `b`, and `c` are names we use to refer to unknown values. They make this formula (the Pythagorean theorem) a general one that applies to arbitrary values of `a`, `b`, and `c`, not just a specific set like:

$$5^2 = 4^2 + 3^2$$

A variable in a Java program is exactly the same thing: it is a name we use to refer to a value. More precisely, a variable is a name that refers to a storage space for a value. We often say that a variable holds a value.

Line 6 begins with the words `int input`. This declares a variable named `input` and specifies that the variable has the type `int`; that is, it is an integer. Java can work with several different types of values, including integers, real or floating-point numbers, characters (e.g., letters, digits), and strings. Java is a *strongly typed* language, which means that all variables must have a type specified and can only refer to values of that type. Our `input` variable always refers to an integer; it cannot refer to a floating point number or a string. Method parameters are also typed. Recall that the `args` parameter had a type of `String[]`.

Continuing with line 6, the variable declaration `int input` is followed by the = character. This is the assignment operator in Java; it sets the value of a variable. When reading Java code, don't read = as "equals," but instead read it as "is assigned the value." As we'll see in Chapter 2, *Java Syntax from the Ground Up*, there is a different operator for "equals."

The value being assigned to our `input` variable is `Integer.parseInt(args[0])`. This is a method invocation. This first statement of the `main()` method invokes

another method whose name is `Integer.parseInt()`. As you might guess, this method "parses" an integer; that is, it converts a string representation of an integer, such as "4", to the integer itself. The `Integer.parseInt()` method is not part of the Java language, but it is a core part of the Java API or Application Programming Interface. Every Java program can use the powerful set of classes and methods defined by this core API. The second half of this book is a quick-reference that documents that core API.

When you call a method, you pass values (called *arguments*) that are assigned to the corresponding parameters defined by the method, and the method returns a value. The argument passed to `Integer.parseInt()` is `args[0]`. Recall that `args` is the name of the parameter for `main()`; it specifies an array (or list) of strings. The elements of an array are numbered sequentially, and the first one is always numbered 0. We only care about the first string in the `args` array, so we use the expression `args[0]` to refer to that string. Thus, when we invoke the program as shown earlier, line 6 takes the first string specified after the name of the class, "4", and passes it to the method named `Integer.parseInt()`. This method converts the string to the corresponding integer and returns the integer as its return value. Finally, this returned integer is assigned to the variable named `input`.

Computing the result

The statement on line 7 is a lot like the statement on line 6. It declares a variable and assigns a value to it. The value assigned to the variable is computed by invoking a method. The variable is named `result`, and it has a type of `double`. `double` means a double-precision floating-point number. The variable is assigned a value that is computed by the `factorial()` method. The `factorial()` method, however, is not part of the standard Java API. Instead, it is defined as part of our program, by lines 11 through 19. The argument passed to `factorial()` is the value referred to by the `input` variable, which was computed on line 6. We'll consider the body of the `factorial()` method shortly, but you can surmise from its name that this method takes an input value, computes the factorial of that value, and returns the result.

Displaying output

Line 8 simply calls a method named `System.out.println()`. This commonly used method is part of the core Java API; it causes the Java interpreter to print out a value. In this case, the value that it prints is the value referred to by the variable named `result`. This is the result of our factorial computation. If the `input` variable holds the value 4, the `result` variable holds the value 24, and this line prints out that value.

The `System.out.println()` method does not have a return value, so there is no variable declaration or = assignment operator in this statement, since there is no value to assign to anything. Another way to say this is that, like the `main()` method of line 5, `System.out.println()` is declared `void`.

The end of a method

Line 9 contains only a single character, }. This marks the end of the method. When the Java interpreter gets here, it is done executing the `main()` method, so it stops running. The end of the `main()` method is also the end of the *variable scope* for the `input` and `result` variables declared within `main()` and for the `args` parameter of `main()`. These variable and parameter names have meaning only within the `main()` method and cannot be used elsewhere in the program, unless other parts of the program declare different variables or parameters that happen to have the same name.

Blank lines

Line 10 is a blank line. You can insert blank lines, spaces, and tabs anywhere in a program, and you should use them liberally to make the program readable. A blank line appears here to separate the `main()` method from the `factorial()` method that begins on line 11. You'll notice that the program also uses spaces and tabs to indent the various lines of code. This kind of indentation is optional; it emphasizes the structure of the program and greatly enhances the readability of the code.

Another method

Line 11 begins the definition of the `factorial()` method that was used by the `main()` method. Compare this line to line 5 to note its similarities and differences. The `factorial()` method has the same `public` and `static` modifiers. It takes a single integer parameter, which we call `x`. Unlike the `main()` method, which had no return value (`void`), `factorial()` returns a value of type `double`. The open curly brace marks the beginning of the method body, which continues past the nested braces on lines 15 and 18 to line 20, where the matching close curly brace is found. The body of the `factorial()` method, like the body of the `main()` method, is composed of statements, which are found on lines 12 through 19.

Checking for valid input

In the `main()` method, we saw variable declarations, assignments, and method invocations. The statement on line 12 is different. It is an `if` statement, which executes another statement conditionally. We saw earlier that the Java interpreter executes the three statements of the `main()` method one after another. It always executes them in exactly that way, in exactly that order. An `if` statement is a flow-control statement; it can affect the way the interpreter runs a program.

The `if` keyword is followed by a parenthesized expression and a statement. The Java interpreter first evaluates the expression. If it is `true`, the interpreter executes the statement. If the expression is `false`, however, the interpreter skips the statement and goes to the next one. The condition for the `if` statement on line 12 is `x < 0`. It checks whether the value passed to the `factorial()` method is less than zero. If it is, this expression is `true`, and the statement on line 13 is executed. Line 12 does not end with a semicolon because the statement on line 13 is part of the `if` statement. Semicolons are required only at the end of a statement.

Line 13 is a `return` statement. It says that the return value of the `factorial()` method is 0.0. `return` is also a flow-control statement. When the Java interpreter sees a `return`, it stops executing the current method and returns the specified value immediately. A `return` statement can stand alone, but in this case, the `return` statement is part of the `if` statement on line 12. The indentation of line 13 helps emphasize this fact. (Java ignores this indentation, but it is very helpful for humans who read Java code!) Line 13 is executed only if the expression on line 12 is `true`.

Before we move on, we should pull back a bit and talk about why lines 12 and 13 are necessary in the first place. It is an error to try to compute a factorial for a negative number, so these lines make sure that the input value `x` is valid. If it is not valid, they cause `factorial()` to return a consistent invalid result, 0.0.

An important variable

Line 14 is another variable declaration; it declares a variable named `fact` of type `double` and assigns it an initial value of 1.0. This variable holds the value of the factorial as we compute it in the statements that follow. In Java, variables can be declared anywhere; they are not restricted to the beginning of a method or block of code.

Looping and computing the factorial

Line 15 introduces another type of statement: the `while` loop. Like an `if` statement, a `while` statement consists of a parenthesized expression and a statement. When the Java interpreter sees a `while` statement, it evaluates the associated expression. If that expression is `true`, the interpreter executes the statement. The interpreter repeats this process, evaluating the expression and executing the statement if the expression is `true`, until the expression evaluates to `false`. The expression on line 15 is `x > 1`, so the `while` statement loops *while* the parameter `x` holds a value that is greater than 1. Another way to say this is that the loop continues *until* `x` holds a value less than or equal to 1. We can assume from this expression that if the loop is ever going to terminate, the value of `x` must somehow be modified by the statement that the loop executes.

The major difference between the `if` statement on lines 12–13 and the `while` loop on lines 15–18 is that the statement associated with the `while` loop is a *compound statement.* A compound statement is zero or more statements grouped between curly braces. The `while` keyword on line 15 is followed by an expression in parentheses and then by an open curly brace. This means that the body of the loop consists of all statements between that opening brace and the closing brace on line 18. Earlier in the chapter, I said that all Java statements end with semicolons. This rule does not apply to compound statements, however, as you can see by the lack of a semicolon at the end of line 18. The statements inside the compound statement (lines 16 and 17) do end with semicolons, of course.

The body of the `while` loop consists of the statements on line 16 and 17. Line 16 multiplies the value of `fact` by the value of `x` and stores the result back into `fact`. Line 17 is similar. It subtracts 1 from the value of `x` and stores the result back into `x`. The * character on line 16 is important: it is the multiplication *operator.* And, as

you can probably guess, the - on line 17 is the subtraction operator. An operator is a key part of Java syntax: it performs a computation on one or two *operands* to produce a new value. Operands and operators combine to form *expressions*, such as `fact * x` or `x - 1`. We've seen other operators in the program. Line 15, for example, uses the greater-than operator (>) in the expression `x > 1`, which compares the value of the variable `x` to 1. The value of this expression is a boolean truth value—either `true` or `false`, depending on the result of the comparison.

To understand this `while` loop, it is helpful to think like the Java interpreter. Suppose we are trying to compute the factorial of 4. Before the loop starts, `fact` is 1.0, and `x` is 4. After the body of the loop has been executed once—after the first *iteration*—`fact` is 4.0, and `x` is 3. After the second iteration, `fact` is 12.0, and `x` is 2. After the third iteration, `fact` is 24.0, and `x` is 1. When the interpreter tests the loop condition after the third iteration, it finds that `x > 1` is no longer true, so it stops running the loop, and the program resumes at line 19.

Returning the result

Line 19 is another `return` statement, like the one we saw on line 13. This one does not return a constant value like 0.0, but instead returns the value of the `fact` variable. If the value of `x` passed into the `factorial()` function is 4, then, as we saw earlier, the value of `fact` is 24.0, so this is the value returned. Recall that the `factorial()` method was invoked on line 7 of the program. When this `return` statement is executed, control returns to line 7, where the return value is assigned to the variable named `result`.

Exceptions

If you've made it all the way through the line-by-line analysis of Example 1-1, you are well on your way to understanding the basics of the Java language.* It is a simple but nontrivial program that illustrates many of the features of Java. There is one more important feature of Java programming I want to introduce, but it is one that does not appear in the program listing itself. Recall that the program computes the factorial of the number you specify on the command line. What happens if you run the program without specifying a number?

```
C:\> java Factorial
java.lang.ArrayIndexOutOfBoundsException: 0
        at Factorial.main(Factorial.java:6)
C:\>
```

And what happens if you specify a value that is not a number?

```
C:\> java Factorial ten
java.lang.NumberFormatException: ten
```

---

* If you didn't understood all the details of this factorial program, don't worry. We'll cover the details of the Java language a lot more thoroughly in Chapter 2 and Chapter 3. However, if you feel like you didn't understand any of the line-by-line analysis, you may also find that the upcoming chapters are over your head. In that case, you should probably go elsewhere to learn the basics of the Java language and return to this book to solidify your understanding, and, of course, to use as a reference. One resource you may find useful in learning the language is Sun's online Java tutorial, available at *http://java.sun.com/docs/books/tutorial/*.

```
        at java.lang.Integer.parseInt(Integer.java)
        at java.lang.Integer.parseInt(Integer.java)
        at Factorial.main(Factorial.java:6)
C:\>
```

In both cases, an error occurs or, in Java terminology, an *exception* is thrown. When an exception is thrown, the Java interpreter prints out a message that explains what type of exception it was and where it occurred (both exceptions above occurred on line 6). In the first case, the exception is thrown because there are no strings in the `args` list, meaning we asked for a nonexistent string with `args[0]`. In the second case, the exception is thrown because `Integer.parseInt()` cannot convert the string "ten" to a number. We'll see more about exceptions in Chapter 2 and learn how to handle them gracefully as they occur.

CHAPTER 2

Java Syntax from the Ground Up

This chapter is a terse but comprehensive introduction to Java syntax. It is written primarily for readers who are new to the language, but have at least some previous programming experience. Determined novices with no prior programming experience may also find it useful. If you already know Java, you should find it a useful language reference. In previous editions of this book, this chapter was written explicitly for C and C++ programmers making the transition to Java. It has been rewritten for this edition to make it more generally useful, but it still contains comparisons to C and C++ for the benefit of programmers coming from those languages.*

This chapter documents the syntax of Java programs by starting at the very lowest level of Java syntax and building from there, covering increasingly higher orders of structure. It covers:

- The characters used to write Java programs and the encoding of those characters.
- Data types, literal values, identifiers, and other tokens that comprise a Java program.
- The operators used in Java to group individual tokens into larger expressions.
- Statements, which group expressions and other statements to form logical chunks of Java code.
- Methods (also called functions, procedures, or subroutines), which are named collections of Java statements that can be invoked by other Java code.

---

* Readers who want even more thorough coverage of the Java language should consider *The Java Programming Language, Second Edition*, by Ken Arnold and James Gosling (the creator of Java) (Addison Wesley Longman). And hard-core readers may want to go straight to the primary source: *The Java Language Specification*, by James Gosling, Bill Joy, and Guy Steele (Addison Wesley Longman). This specification is available in printed book form, but is also freely available for download from Sun's web site at *http://java.sun.com/docs/books/jls/*. I found both documents quite helpful while writing this chapter.

- Classes, which are collections of methods and fields. Classes are the central program element in Java and form the basis for object-oriented programming. Chapter 3, *Object-Oriented Programming in Java*, is devoted entirely to a discussion of classes and objects.
- Packages, which are collections of related classes.
- Java programs, which consist of one or more interacting classes that may be drawn from one or more packages.

The syntax of most programming languages is complex, and Java is no exception. In general, it is not possible to document all elements of a language without referring to other elements that have not yet been discussed. For example, it is not really possible to explain in a meaningful way the operators and statements supported by Java without referring to objects. But it is also not possible to document objects thoroughly without referring to the operators and statements of the language. The process of learning Java, or any language, is therefore an iterative one. If you are new to Java (or a Java-style programming language), you may find that you benefit greatly from working through this chapter and the next *twice*, so that you can grasp the interrelated concepts.

The Unicode Character Set

Java programs are written using the Unicode character set. Unlike the 7-bit ASCII encoding, which is useful only for English, and the 8-bit ISO Latin-1 encoding, which is useful only for major Western European languages, the 16-bit Unicode encoding can represent virtually every written language in common use on the planet. Very few text editors support Unicode, however, and in practice, most Java programs are written in plain ASCII. 16-bit Unicode characters are typically written to files using an encoding known as UTF-8, which converts the 16-bit characters into a stream of bytes. The format is designed so that plain ASCII and Latin-1 text are valid UTF-8 byte streams. Thus, you can simply write plain ASCII programs, and they will work as valid Unicode.

If you want to embed a Unicode character within a Java program that is written in plain ASCII, use the special Unicode escape sequence `\uxxxx`. That is, a backslash and a lowercase u, followed by four hexadecimal characters. For example, `\u0020` is the space character, and `\u3c00` is the character π. You can use Unicode characters anywhere in a Java program, including comments and variable names.

Comments

Java supports three types of comments. The first type is a single-line comment, which begins with the characters // and continues until the end of the current line. For example:

```
int i = 0;   // initialize the loop variable
```

The second kind of comment is a multiline comment. It begins with the characters /* and continues, over any number of lines, until the characters */. Any text between the /* and the */ is ignored by the Java compiler. Although this style of

comment is typically used for multiline comments, it can also be used for single-line comments. This type of comment cannot be nested (i.e., one /* */ comment cannot appear within another one). When writing multiline comments, programmers often use extra * characters to make the comments stand out. Here is a typical multiline comment:

```
/*
 * Step 4: Print static methods, both public and protected,
 *         but don't list deprecated ones.
 */
```


The third type of comment is a special case of the second. If a comment begins with /**, it is regarded as a special *doc comment.* Like regular multiline comments, doc comments end with */ and cannot be nested. When you write a Java class you expect other programmers to use, use doc comments to embed documentation about the class and each of its methods directly into the source code. A program named *javadoc* extracts these comments and processes them to create online documentation for your class. A doc comment can contain HTML tags and can use additional syntax understood by *javadoc.* For example:

```
/**
 * Display a list of classes, many to a line.
 *
 * @param classes The classes to display
 * @return <tt>true</tt> on success,
 * <tt>false</tt> on failure.
 * @author David Flanagan
 */
```

See Chapter 7, *Java Programming and Documentation Conventions*, for more information on the doc-comment syntax and Chapter 8, *Java Development Tools*, for more information on the *javadoc* program.

Identifiers and Reserved Words

An *identifier* is any symbolic name that refers to something in a Java program. Class, method, parameter, and variable names are all identifiers. An identifier must begin with a letter, an underscore (_), or a Unicode currency symbol (e.g., $, £, ¥). This initial letter can be followed by any number of letters, digits, underscores, or currency symbols. Remember that Java uses the Unicode character set, which contains quite a few letters and digits other than those in the ASCII character set. The following are legal identifiers:

```
i
engine3
theCurrentTime
the_current_time
θ
```

Identifiers can include numbers, but cannot begin with a number. In addition, they cannot contain any punctuation characters other than underscores and currency characters. By convention, dollar signs and other currency characters are reserved for identifiers automatically generated by a compiler or some kind of code preprocessor. It is best to avoid these characters in your own identifiers.

Another important restriction on identifiers is that you cannot use any of the keywords and literals that are part of the Java language itself. These reserved words are listed in Table 2-1.

Table 2–1: Java Reserved Words

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

Note that const and goto are reserved words, but aren't part of the Java language.

Primitive Data Types

Java supports eight basic data types known as *primitive types*. In addition, it supports classes and arrays as composite data types, or reference types. Classes and arrays are documented later in this chapter. The primitive types are: a boolean type, a character type, four integer types, and two floating-point types. The four integer types and the two floating-point types differ in the number of bits that represent them, and therefore in the range of numbers they can represent. Table 2-2 summarizes these primitive data types.

Table 2–2: Java Primitive Data Types

Type	*Contains*	*Default*	*Size*	*Range*
boolean	true or false	false	1 bit	NA
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
byte	Signed integer	0	8 bits	–128 to 127
short	Signed integer	0	16 bits	–32768 to 32767
int	Signed integer	0	32 bits	–2147483648 to 2147483647
long	Signed integer	0	64 bits	–9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
double	IEEE 754 floating point	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

The boolean Type

The `boolean` type represents a truth value. There are only two possible values of this type, representing the two boolean states: on or off, yes or no, true or false. Java reserves the words `true` and `false` to represent these two boolean values.

C and C++ programmers should note that Java is quite strict about its `boolean` type: `boolean` values can never be converted to or from other data types. In particular, a `boolean` is not an integral type, and integer values cannot be used in place of a `boolean`. In other words, you cannot take shortcuts such as the following in Java:

```
if (o) {
  while(i) {
  }
}
```

Instead, Java forces you to write cleaner code by explicitly stating the comparisons you want:

```
if (o != null) {
  while(i != 0) {
  }
}
```

The char Type

The `char` type represents Unicode characters. It surprises many experienced programmers to learn that Java `char` values are 16 bits long, but in practice this fact is totally transparent. To include a character literal in a Java program, simply place it between single quotes (apostrophes):

```
char c = 'A';
```

You can, of course, use any Unicode character as a character literal, and you can use the `\u` Unicode escape sequence. In addition, Java supports a number of other escape sequences that make it easy both to represent commonly used nonprinting ASCII characters such as newline and to escape certain punctuation characters that have special meaning in Java. For example:

```
char tab = '\t', apostrophe = '\'', nul = '', aleph='\u05D0';
```

Table 2-3 lists the escape characters that can be used in `char` literals. These characters can also be used in string literals, which are covered later in this chapter.

Table 2-3: Java Escape Characters

Escape Sequence	*Character Value*
`\b`	Backspace
`\t`	Horizontal tab
`\n`	Newline
`\f`	Form feed

Table 2-3: Java Escape Characters (continued)

Escape Sequence	*Character Value*
\r	Carriage return
\"	Double quote
\'	Single quote
\\	Backslash
xxx	The Latin-1 character with the encoding *xxx*, where *xxx* is an octal (base 8) number between 000 and 377. The forms *x* and *xx* are also legal, as in '\0', but are not recommended because they can cause difficulties in string constants where the escape sequence is followed by a regular digit.
\u*xxxx*	The Unicode character with encoding *xxxx*, where *xxxx* is four hexadecimal digits. Unicode escapes can appear anywhere in a Java program, not only in character and string literals.

`char` values can be converted to and from the various integral types. Unlike `byte`, `short`, `int`, and `long`, however, `char` is an unsigned type. The `Character` class defines a number of useful `static` methods for working with characters, including `isDigit()`, `isJavaLetter()`, `isLowerCase()`, and `toUpperCase()`.

Integer Types

The integer types in Java are `byte`, `short`, `int`, and `long`. As shown in Table 2-2, these four types differ only in the number of bits and, therefore, in the range of numbers each type can represent. All integral types represent signed numbers; there is no `unsigned` keyword as there is in C and C++.

Literals for each of these types are written exactly as you would expect: as a string of decimal digits. Although it is not technically part of the literal syntax, any integer literal can be preceded by the unary minus operator to indicate a negative number. Here are some legal integer literals:

```
0
1
123
-42000
```

Integer literals can also be expressed in hexadecimal or octal notation. A literal that begins with `0x` or `0X` is taken as a hexadecimal number, using the letters `A` to `F` (or `a` to `f`) as the additional digits required for base-16 numbers. Integer literals beginning with a leading `0` are taken to be octal (base-8) numbers and cannot include the digits 8 or 9. Java does not allow integer literals to be expressed in binary (base-2) notation. Legal hexadecimal and octal literals include:

```
0xff            // Decimal 255, expressed in hexadecimal
0377            // The same number, expressed in octal (base 8)
0xCAFEBABE      // A magic number used to identify Java class files
```

Integer literals are 32-bit `int` values unless they end with the character `L` or `l`, in which case they are 64-bit `long` values:

```
1234        // An int value
1234L       // A long value
0xffL       // Another long value
```

Integer arithmetic in Java is modular, which means that it never produces an overflow or an underflow when you exceed the range of a given integer type. Instead, numbers just wrap around. For example:

```
byte b1 = 127, b2 = 1;   // Largest byte is 127
byte sum = b1 + b2;      // Sum wraps to -128, which is the smallest byte
```


Neither the Java compiler nor the Java interpreter warns you in any way when this occurs. When doing integer arithmetic, you simply must ensure that the type you are using has a sufficient range for the purposes you intend. Integer division by zero and modulo by zero are illegal and cause an `ArithmeticException` to be thrown.

Each integer type has a corresponding wrapper class: `Byte`, `Short`, `Integer`, and `Long`. Each of these classes defines `MIN_VALUE` and `MAX_VALUE` constants that describe the range of the type. The classes also define useful static methods, such as `Byte.parseByte()` and `Integer.parseInt()`, for converting strings to integer values.

Floating-Point Types

Real numbers in Java are represented with the `float` and `double` data types. As shown in Table 2-3, `float` is a 32-bit, single-precision floating-point value, and `double` is a 64-bit, double-precision floating-point value. Both types adhere to the IEEE 754-1985 standard, which specifies both the format of the numbers and the behavior of arithmetic for the numbers.

Floating-point values can be included literally in a Java program as an optional string of digits, followed by a decimal point and another string of digits. Here are some examples:

```
123.45
0.0
.01
```

Floating-point literals can also use exponential, or scientific, notation, in which a number is followed by the letter `e` or `E` (for exponent) and another number. This second number represents the power of ten by which the first number is multiplied. For example:

```
1.2345E02      // 1.2345 × 10^2, or 123.45
1e-6           // 1 × 10^-6, or 0.000001
6.02e23        // Avagadro's Number: 6.02 × 10^23
```

Floating-point literals are `double` values by default. To include a `float` value literally in a program, follow the number by the character `f` or `F`:

```
double d = 6.02E23;
float f = 6.02e23f;
```

Floating-point literals cannot be expressed in hexadecimal or octal notation.

Most real numbers, by their very nature, cannot be represented exactly in any finite number of bits. Thus, it is important to remember that `float` and `double` values are only approximations of the numbers they are meant to represent. A `float` is a 32-bit approximation, which results in at least 6 significant decimal digits, and a `double` is a 64-bit approximation, which results in at least 15 significant digits. In practice, these data types are suitable for most real-number computations.

In addition to representing ordinary numbers, the `float` and `double` types can also represent four special values: positive and negative infinity, zero, and NaN. The infinity values result when a floating-point computation produces a value that overflows the representable range of a `float` or `double`. When a floating-point computation underflows the representable range of a `float` or a `double`, a zero value results. The Java floating-point types make a distinction between positive zero and negative zero, depending on the direction from which the underflow occurred. In practice, positive and negative zero behave pretty much the same. Finally, the last special floating-point value is NaN, which stands for not-a-number. The NaN value results when an illegal floating-point operation, such as 0/0, is performed. Here are examples of statements that result in these special values:

```
double inf = 1/0;             // Infinity
double neginf = -1/0;         // -Infinity
double negzero = -1/inf;      // Negative zero
double NaN = 0/0;             // NaN
```

Because the Java floating-point types can handle overflow to infinity and underflow to zero and have a special NaN value, floating-point arithmetic never throws exceptions, even when performing illegal operations, like dividing zero by zero or taking the square root of a negative number.

The `float` and `double` primitive types have corresponding classes, named `Float` and `Double`. Each of these classes defines the following useful constants: `MIN_VALUE`, `MAX_VALUE`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, and `NaN`.

The infinite floating-point values behave as you would expect. Adding or subtracting anything to or from infinity, for example, yields infinity. Negative zero behaves almost identically to positive zero, and, in fact, the == equality operator reports that negative zero is equal to positive zero. The only way to distinguish negative zero from positive, or regular, zero is to divide by it. 1/0 yields positive infinity, but 1 divided by negative zero yields negative infinity. Finally, since NaN is not-a-number, the == operator says that it is not equal to any other number, including itself! To check whether a `float` or `double` value is NaN, you must use the `Float.isNan()` and `Double.isNan()` methods.

Strings

In addition to the boolean, character, integer, and floating-point data types, Java also has a data type for working with strings of text (usually simply called *strings*). The `String` type is a class, however, and is not one of the primitive types of the

language. Because strings are so commonly used, though, Java does have a syntax for including string values literally in a program. A `String` literal consists of arbitrary text within double quotes. For example:

```
"Hello, world"
"'This' is a string!"
```

String literals can contain any of the escape sequences that can appear as `char` literals (see Table 2-3). Use the `\"` sequence to include a double-quote within a `String` literal. Strings and string literals are discussed in more detail later in this chapter. Chapter 4, *The Java Platform*, demonstrates some of the ways you can work with `String` objects in Java.

Type Conversions

Java allows conversions between integer values and floating-point values. In addition, because every character corresponds to a number in the Unicode encoding, `char` types can be converted to and from the integer and floating-point types. In fact, `boolean` is the only primitive type that cannot be converted to or from another primitive type in Java.

There are two basic types of conversions. A *widening conversion* occurs when a value of one type is converted to a wider type—one that is represented with more bits and therefore has a wider range of legal values. A *narrowing conversion* occurs when a value is converted to a type that is represented with fewer bits. Java performs widening conversions automatically when, for example, you assign an `int` literal to a `double` variable or a `char` literal to an `int` variable.

Narrowing conversions are another matter, however, and are not always safe. It is reasonable to convert the integer value 13 to a `byte`, for example, but it is not reasonable to convert 13000 to a `byte`, since `byte` can only hold numbers between $ndash;128 and 127. Because you can lose data in a narrowing conversion, the Java compiler complains when you attempt any narrowing conversion, even if the value being converted would in fact fit in the narrower range of the specified type:

```
int i = 13;
byte b = i;    // The compiler does not allow this
```

The one exception to this rule is that you can assign an integer literal (an `int` value) to a `byte` or `short` variable, if the literal falls within the range of the variable.

If you need to perform a narrowing conversion and are confident you can do so without losing data or precision, you can force Java to perform the conversion using a language construct known as a *cast*. Perform a cast by placing the name of the desired type in parentheses before the value to be converted. For example:

```
int i = 13;
byte b = (byte) i;   // Force the int to be converted to a byte
i = (int) 13.456;    // Force this double literal to the int 13
```

Casts of primitive types are most often used to convert floating-point values to integers. When you do this, the fractional part of the floating-point value is simply truncated (i.e., the floating-point value is rounded towards zero, not towards the

nearest integer). The methods `Math.round()`, `Math.floor()`, and `Math.ceil()` perform other types of rounding.

The `char` type acts like an integer type in most ways, so a `char` value can be used anywhere an `int` or `long` value is required. Recall, however, that the `char` type is *unsigned*, so it behaves differently than the `short` type, even though both of them are 16 bits wide:

```
short s = (short) 0xffff; // These bits represent the number -1
char c = '\uffff';        // The same bits, representing a Unicode character
int i1 = s;               // Converting the short to an int yields -1
int i2 = c;               // Converting the char to an int yields 65535
```

Table 2-4 is a grid that shows which primitive types can be converted to which other types and how the conversion is performed. The letter N in the table means that the conversion cannot be performed. The letter Y means that the conversion is a widening conversion and is therefore performed automatically and implicitly by Java. The letter C means that the conversion is a narrowing conversion and requires an explicit cast. Finally, the notation Y* means that the conversion is an automatic widening conversion, but that some of the least significant digits of the value may be lost by the conversion. This can happen when converting an `int` or `long` to a `float` or `double`. The floating-point types have a larger range than the integer types, so any `int` or `long` can be represented by a `float` or `double`. However, the floating-point types are approximations of numbers and cannot always hold as many significant digits as the integer types.

Table 2-4: Java Primitive Type Conversions

Convert From:	*Convert To:* `boolean`	`byte`	`short`	`char`	`int`	`long`	`float`	`double`
`boolean`	–	N	N	N	N	N	N	N
`byte`	N	–	Y	C	Y	Y	Y	Y
`short`	N	C	–	C	Y	Y	Y	Y
`char`	N	C	C	–	Y	Y	Y	Y
`int`	N	C	C	C	–	Y	Y*	Y
`long`	N	C	C	C	C	–	Y*	Y*
`float`	N	C	C	C	C	C	–	Y
`double`	N	C	C	C	C	C	C	–

Reference Types

In addition to its eight primitive types, Java defines two additional categories of data types: classes and arrays. Java programs consist of class definitions; each class defines a new data type that can be manipulated by Java programs. For example, a program might define a class named `Point` and use it to store and manipulate X,Y points in a Cartesian coordinate system. This makes `Point` a new data type in that program. An array type represents a list of values of some other type. `char` is a data type, and an array of `char` values is another data type, written `char[]`. An

array of `Point` objects is a data type, written `Point[]`. And an array of `Point` arrays is yet another type, written `Point[][]`.

As you can see, there are an infinite number of possible class and array data types. Collectively, these data types are known as *reference types*. The reason for this name will become clear later in this chapter. For now, however, what is important to understand is that class and array types differ significantly from primitive types, in that they are compound, or composite, types. A primitive data type holds exactly one value. Classes and arrays are aggregate types that contain multiple values. The `Point` type, for example, holds two `double` values representing the X and Y coordinates of the point. And `char[]` is obviously a compound type because it represents a list of characters. By their very nature, class and array types are more complicated than the primitive data types. We'll discuss classes and arrays in detail later in this chapter and examine classes in even more detail in Chapter 3.

Expressions and Operators

So far in this chapter, we've learned about the primitive types that Java programs can manipulate and seen how to include primitive values as *literals* in a Java program. We've also used *variables* as symbolic names that represent, or hold, values. These literals and variables are the tokens out of which Java programs are built.

An *expression* is the next higher level of structure in a Java program. The Java interpreter *evaluates* an expression to compute its value. The very simplest expressions are called *primary expressions* and consist of literals and variables. So, for example, the following are all expressions:

```
1.7         // An integer literal
true        // A boolean literal
sum         // A variable
```

When the Java interpreter evaluates a literal expression, the resulting value is the literal itself. When the interpreter evaluates a variable expression, the resulting value is the value stored in the variable.

Primary expressions are not very interesting. More complex expressions are made by using *operators* to combine primary expressions. For example, the following expression uses the assignment operator to combine two primary expressions—a variable and a floating-point literal—into an assignment expression:

```
sum = 1.7
```

But operators are used not only with primary expressions; they can also be used with expressions at any level of complexity. Thus, the following are all legal expressions:

```
sum = 1 + 2 + 3*1.2 + (4 + 8)/3.0
sum/Math.sqrt(3.0 * 1.234)
(int)(sum + 33)
```

Operator Summary

The kinds of expressions you can write in a programming language depend entirely on the set of operators available to you. Table 2-5 summarizes the operators available in Java. The P and A columns of the table specify the precedence and associativity of each group of related operators, respectively.

Table 2–5: Java Operators

P	*A*	*Operator*	*Operand Type(s)*	*Operation Performed*
15	L	.	object, member	object member access
		[]	array, int	array element access
		(*args*)	method, arglist	method invocation
		++, --	variable	post-increment, decrement
14	R	++, --	variable	pre-increment, decrement
		+, -	number	unary plus, unary minus
		~	integer	bitwise complement
		!	boolean	boolean NOT
13	R	new	class, arglist	object creation
		(*type*)	type, any	cast (type conversion)
12	L	*, /, %	number, number	multiplication, division, remainder
11	L	+, -	number, number	addition, subtraction
		+	string, any	string concatenation
10	L	<<	integer, integer	left shift
		>>	integer, integer	right shift with sign extension
		>>>	integer, integer	right shift with zero extension
9	L	<, <=	number, number	less than, less than or equal
		>, >=	number, number	greater than, greater than or equal
		instanceof	reference, type	type comparison
8	L	==	primitive, primitive	equal (have identical values)
		!=	primitive, primitive	not equal (have different values)
		==	reference, reference	equal (refer to same object)
		!=	reference, reference	not equal (refer to different objects)
7	L	&	integer, integer	bitwise AND
		&	boolean, boolean	boolean AND
6	L	^	integer, integer	bitwise XOR
		^	boolean, boolean	boolean XOR
5	L	|	integer, integer	bitwise OR

Table 2-5: Java Operators (continued)

P	A	Operator	Operand Type(s)	Operation Performed
		|	boolean, boolean	boolean OR
4	L	&&	boolean, boolean	conditional AND
3	L	||	boolean, boolean	conditional OR
2	R	?:	boolean, any, any	conditional (ternary) operator
1	R	=	variable, any	assignment
		*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	variable, any	assignment with operation

Precedence

The P column of Table 2-5 specifies the *precedence* of each operator. Precedence specifies the order in which operations are performed. Consider this expression:

```
a + b * c
```

The multiplication operator has higher precedence than the addition operator, so `a` is added to the product of `b` and `c`. Operator precedence can be thought of as a measure of how tightly operators bind to their operands. The higher the number, the more tightly they bind.

Default operator precedence can be overridden through the use of parentheses, to explicitly specify the order of operations. The previous expression can be rewritten as follows to specify that the addition should be performed before the multiplication:

```
(a + b) * c
```

The default operator precedence in Java was chosen for compatibility with C; the designers of C chose this precedence so that most expressions can be written naturally without parentheses. There are only a few common Java idioms for which parentheses are required. Examples include:

```
// Class cast combined with member access
((Integer) o).intValue();

// Assignment combined with comparison
while((line = in.readLine()) != null) { ... }

// Bitwise operators combined with comparison
if ((flags & (PUBLIC | PROTECTED)) != 0) { ... }
```

Associativity

When an expression involves several operators that have the same precedence, the operator associativity governs the order in which the operations are performed. Most operators are left-to-right associative, which means that the

operations are performed from left to right. The assignment and unary operators, however, have right-to-left associativity. The A column of Table 2-5 specifies the associativity of each operator or group of operators. The value L means left to right, and R means right to left.

The additive operators are all left-to-right associative, so the expression `a+b-c` is evaluated from left to right: `(a+b)-c`. Unary operators and assignment operators are evaluated from right to left. Consider this complex expression:

```
a = b += c = -~d
```

This is evaluated as follows:

```
a = (b += (c = -(~d)))
```

As with operator precedence, operator associativity establishes a default order of evaluation for an expression. This default order can be overridden through the use of parentheses. However, the default operator associativity in Java has been chosen to yield a natural expression syntax, and you rarely need to alter it.

Operand number and type

The fourth column of Table 2-5 specifies the number and type of the operands expected by each operator. Some operators operate on only one operand; these are called unary operators. For example, the unary minus operator changes the sign of a single number:

```
-n              // The unary minus operator
```

Most operators, however, are binary operators that operate on two operand values. The - operator actually comes in both forms:

```
a - b           // The subtraction operator is a binary operator
```

Java also defines one ternary operator, often called the conditional operator. It is like an `if` statement inside an expression. Its three operands are separated by a question mark and a colon; the second and third operators must both be of the same type:

```
x > y ? x : y  // Ternary expression; evaluates to the larger of x and y
```

In addition to expecting a certain number of operands, each operator also expects particular types of operands. Column four of the table lists the operand types. Some of the codes used in that column require further explanation:

number
: An integer, floating-point value, or character (i.e., any primitive type except `boolean`)

integer
: A `byte`, `short`, `int`, `long`, or `char` value (`long` values are not allowed for the array access operator `[]`)

reference
An object or array

variable
A variable or anything else, such as an array element, to which a value can be assigned

Return type

Just as every operator expects its operands to be of specific types, each operator produces a value of a specific type. The arithmetic, increment and decrement, bitwise, and shift operators return a `double` if at least one of the operands is a `double`. Otherwise, they return a `float` if at least one of the operands is a `float`. Otherwise, they return a `long` if at least one of the operands is a `long`. Otherwise, they return an `int`, even if both operands are `byte`, `short`, or `char` types that are narrower than `int`.

The comparison, equality, and boolean operators always return `boolean` values. Each assignment operator returns whatever value it assigned, which is of a type compatible with the variable on the left side of the expression. The conditional operator returns the value of its second or third argument (which must both be of the same type).

Side effects

Every operator computes a value based on one or more operand values. Some operators, however, have *side effects* in addition to their basic evaluation. If an expression contains side effects, evaluating it changes the state of a Java program in such a way that evaluating the expression again may yield a different result. For example, the ++ increment operator has the side effect of incrementing a variable. The expression `++a` increments the variable `a` and returns the newly incremented value. If this expression is evaluated again, the value will be different. The various assignment operators also have side effects. For example, the expression `a*=2` can also be written as `a=a*2`. The value of the expression is the value of `a` multiplied by 2, but the expression also has the side effect of storing that value back into `a`. The method invocation operator `()` has side effects if the invoked method has side effects. Some methods, such as `Math.sqrt()`, simply compute and return a value without side effects of any kind. Typically, however, methods do have side effects. Finally, the `new` operator has the profound side effect of creating a new object.

Order of evaluation

When the Java interpreter evaluates an expression, it performs the various operations in an order specified by the parentheses in the expression, the precedence of the operators, and the associativity of the operators. Before any operation is performed, however, the interpreter first evaluates the operands of the operator. (The exceptions are the `&&`, `||`, and `?:` operators, which do not always evaluate all their operands.) The interpreter always evaluates operands in order from left to right. This matters if any of the operands are expressions that contain side effects. Consider this code, for example:

```
int a = 2;
int v = ++a + ++a * ++a;
```

Although the multiplication is performed before the addition, the operands of the + operator are evaluated first. Thus, the expression evaluates to 3+4*5, or 23.

Arithmetic Operators

Since most programs operate primarily on numbers, the most commonly used operators are often those that perform arithmetic operations. The arithmetic operators can be used with integers, floating-point numbers, and even characters (i.e., they can be used with any primitive type other than `boolean`). If either of the operands is a floating-point number, floating-point arithmetic is used; otherwise, integer arithmetic is used. This matters because integer arithmetic and floating-point arithmetic differ in the way division is performed and in the way underflows and overflows are handled, for example. The arithmetic operators are:

Addition (+)

The + operator adds two numbers. As we'll see shortly, the + operator can also be used to concatenate strings. If either operand of + is a string, the other one is converted to a string as well. Be sure to use parentheses when you want to combine addition with concatenation. For example:

```
System.out.println("Total: " + 3 + 4);    // Prints "Total: 34", not 7!
```

Subtraction (-)

When - is used as a binary operator, it subtracts its second operand from its first. For example, 7–3 evaluates to 4. The - operator can perform unary negation.

Multiplication (*)

The * operator multiplies its two operands. For example, 7*3 evaluates to 21.

Division (/)

The / operator divides its first operand by its second. If both operands are integers, the result is an integer, and any remainder is lost. If either operand is a floating-point value, however, the result is a floating-point value. When dividing two integers, division by zero throws an `ArithmeticException`. For floating-point calculations, however, division by zero simply yields an infinite result or NaN:

```
7/3          // Evaluates to 2
7/3.0f       // Evaluates to 2.333333f
7/0          // Throws an ArithmeticException
7/0.0        // Evaluates to positive infinity
0.0/0.0      // Evaluates to NaN
```

Modulo (%)

The % operator computes the first operand modulo the second operand (i.e., it returns the remainder when the first operand is divided by the second operand an integral number of times). For example, 7%3 is 1. The sign of the result is the same as the sign of the first operand. While the modulo operator is typically used with integer operands, it also works for floating-point values. For example, 4.3%2.1 evaluates to 0.1. When operating with integers, trying to

compute a value modulo zero causes an `ArithmeticException`. When working with floating-point values, anything modulo 0.0 evaluates to `NaN`, as does infinity modulo anything.

Unary Minus (–)
When – is used as a unary operator, before a single operand, it performs unary negation. In other words, it converts a positive value to an equivalently negative value, and vice versa.

String Concatenation Operator

In addition to adding numbers, the + operator (and the related += operator) also concatenates, or joins, strings. If either of the operands to + is a string, the operator converts the other operand to a string. For example:

```
System.out.println("Quotient: " + 7/3.0f);  // Prints "Quotient: 2.3333333"
```

As a result, you must be careful to put any addition expressions in parentheses when combining them with string concatenation. If you do not, the addition operator is interpreted as a concatenation operator.

The Java interpreter has built-in string conversions for all primitive types. An object is converted to a string by invoking its `toString()` method. Some classes define custom `toString()` methods, so that objects of that class can easily be converted to strings in this way. An array is converted to a string by invoking the built-in `toString()` method, which, unfortunately, does not return a useful string representation of the array contents.

Increment and Decrement Operators

The ++ operator increments its single operand, which must be a variable, an element of an array, or a field of an object, by one. The behavior of this operator depends on its position relative to the operand. When used before the operand, where it is known as the *pre-increment* operator, it increments the operand and evaluates to the incremented value of that operand. When used after the operand, where it is known as the *post-increment* operator, it increments its operand, but evaluates to the value of that operand before it was incremented.

For example, the following code sets both `i` and `j` to 2:

```
i = 1;
j = ++i;
```

But these lines set `i` to 2 and `j` to 1:

```
i = 1;
j = i++;
```

Similarly, the -- operator decrements its single numeric operand, which must be a variable, an element of an array, or a field of an object, by one. Like the ++ operator, the behavior of -- depends on its position relative to the operand. When used before the operand, it decrements the operand and returns the decremented value.

When used after the operand, it decrements the operand, but returns the *un-decremented* value.

The expressions x++ and x-- are equivalent to x=x+1 and x=x-1, respectively, except that when using the increment and decrement operators, x is only evaluated once. If x is itself an expression with side effects, this makes a big difference. For example, these two expressions are not equivalent:

```
a[i++]++;             // Increments an element of an array
a[i++] = a[i++] + 1;  // Adds one to an array element and stores it in another
```

These operators, in both prefix and postfix forms, are most commonly used to increment or decrement the counter that controls a loop.

Comparison Operators

The comparison operators consist of the equality operators that test values for equality or inequality and the relational operators used with ordered types (numbers and characters) to test for greater than and less than relationships. Both types of operators yield a `boolean` result, so they are typically used with `if` statements and `while` and `for` loops to make branching and looping decisions. For example:

```
if (o != null) ...;           // The not equals operator
while(i < a.length) ...;      // The less than operator
```

Java provides the following equality operators:

Equals (==)
: The == operator evaluates to `true` if its two operands are equal and `false` otherwise. With primitive operands, it tests whether the operand values themselves are identical. For operands of reference types, however, it tests whether the operands refer to the same object or array. In other words, it does not test the equality of two distinct objects or arrays. In particular, note that you cannot test two distinct strings for equality with this operator.

  If == is used to compare two numeric or character operands that are not of the same type, the narrower operand is converted to the type of the wider operand before the comparison is done. For example, when comparing a `short` to a `float`, the `short` is first converted to a `float` before the comparison is performed. For floating-point numbers, the special negative zero value tests equal to the regular, positive zero value. Also, the special NaN (not-a-number) value is not equal to any other number, including itself. To test whether a floating-point value is NaN, use the `Float.isNan()` or `Double.isNan()` method.

Not Equals (!=)
: The != operator is exactly the opposite of the == operator. It evaluates to `true` if its two primitive operands have different values or if its two reference operands refer to different objects or arrays. Otherwise, it evaluates to `false`.

The relational operators can be used with numbers and characters, but not with `boolean` values, objects, or arrays because those types are not ordered. Java provides the following relational operators:

Less Than (<)
Evaluates to `true` if the first operand is less than the second.

Less Than or Equal (<=)
Evaluates to `true` if the first operand is less than or equal to the second.

Greater Than (>)
Evaluates to `true` if the first operand is greater than the second.

Greater Than or Equal (>=)
Evaluates to `true` if the first operand is greater than or equal to the second.

Boolean Operators

As we've just seen, the comparison operators compare their operands and yield a `boolean` result, which is often used in branching and looping statements. In order to make branching and looping decisions based on conditions more interesting than a single comparison, you can use the Boolean (or logical) operators to combine multiple comparison expressions into a single, more complex, expression. The Boolean operators require their operands to be `boolean` values and they evaluate to `boolean` values. The operators are:

Conditional AND (&&)
This operator performs a Boolean AND operation on its operands. It evaluates to `true` if and only if both its operands are `true`. If either or both operands are `false`, it evaluates to `false`. For example:

```
if (x < 10 && y > 3) ... // If both comparisons are true
```

This operator (and all the Boolean operators except the unary ! operator) have a lower precedence than the comparison operators. Thus, it is perfectly legal to write a line of code like the one above. However, some programmers prefer to use parentheses to make the order of evaluation explicit:

```
if ((x < 10) && (y > 3)) ...
```

You should use whichever style you find easier to read.

This operator is called a conditional AND because it conditionally evaluates its second operand. If the first operand evaluates to `false`, the value of the expression is `false`, regardless of the value of the second operand. Therefore, to increase efficiency, the Java interpreter takes a shortcut and skips the second operand. Since the second operand is not guaranteed to be evaluated, you must use caution when using this operator with expressions that have side effects. On the other hand, the conditional nature of this operator allows us to write Java expressions such as the following:

```
if (data != null && i < data.length && data[i] != -1) ...
```

The second and third comparisons in this expression would cause errors if the first or second comparisons evaluated to `false`. Fortunately, we don't have to worry about this because of the conditional behavior of the && operator.

Conditional OR (||)

This operator performs a Boolean OR operation on its two `boolean` operands. It evaluates to `true` if either or both of its operands are `true`. If both operands are `false`, it evaluates to `false`. Like the && operator, || does not always evaluate its second operand. If the first operand evaluates to `true`, the value of the expression is `true`, regardless of the value of the second operand. Thus, the operator simply skips that second operand in that case.

Boolean NOT (!)

This unary operator changes the `boolean` value of its operand. If applied to a `true` value, it evaluates to `false`, and if applied to a `false` value, it evaluates to `true`. It is useful in expressions like these:

```
if (!found) ...          // found is a boolean variable declared somewhere
while (!c.isEmpty()) ... // The isEmpty() method returns a boolean value
```

Because ! is a unary operator, it has a high precedence and often must be used with parentheses:

```
if (!(x > y && y > z))
```

Boolean AND (&)

When used with `boolean` operands, the & operator behaves like the && operator, except that it always evaluates both operands, regardless of the value of the first operand. This operator is almost always used as a bitwise operator with integer operands, however, and many Java programmers would not even recognize its use with `boolean` operands as legal Java code.

Boolean OR (|)

This operator performs a Boolean OR operation on its two `boolean` operands. It is like the || operator, except that it always evaluates both operands, even if the first one is `true`. The | operator is almost always used as a bitwise operator on integer operands; its use with `boolean` operands is very rare.

Boolean XOR (^)

When used with `boolean` operands, this operator computes the Exclusive OR (XOR) of its operands. It evaluates to `true` if exactly one of the two operands is `true`. In other words, it evaluates to `false` if both operands are `false` or if both operands are `true`. Unlike the && and || operators, this one must always evaluate both operands. The ^ operator is much more commonly used as a bitwise operator on integer operands. With `boolean` operands, this operator is equivalent to the != operator.

Bitwise and Shift Operators

The bitwise and shift operators are low-level operators that manipulate the individual bits that make up an integer value. The bitwise operators are most commonly used for testing and setting individual flag bits in a value. In order to understand their behavior, you must understand binary (base-2) numbers and the twos-complement format used to represent negative integers. You cannot use these operators with floating-point, `boolean`, array, or object operands. When used with `boolean` operands, the &, |, and ^ operators perform a different operation, as described in the previous section.

If either of the arguments to a bitwise operator is a `long`, the result is a `long`. Otherwise, the result is an `int`. If the left operand of a shift operator is a `long`, the result is a `long`; otherwise, the result is an `int`. The operators are:

Bitwise Complement (~)

The unary ~ operator is known as the bitwise complement, or bitwise NOT, operator. It inverts each bit of its single operand, converting ones to zeros and zeros to ones. For example:

```
byte b = ~12;           // ~00000110 ==> 11111001 or -13 decimal
flags = flags & ~f;     // Clear flag f in a set of flags
```


Bitwise AND (&)

This operator combines its two integer operands by performing a Boolean AND operation on their individual bits. The result has a bit set only if the corresponding bit is set in both operands. For example:

```
10 & 7                  // 00001010 & 00000111 ==> 00000010 or 2
if ((flags & f) != 0)   // Test whether flag f is set
```

When used with `boolean` operands, & is the infrequently used Boolean AND operator described earlier.

Bitwise OR (|)

This operator combines its two integer operands by performing a Boolean OR operation on their individual bits. The result has a bit set if the corresponding bit is set in either or both of the operands. It has a zero bit only where both corresponding operand bits are zero. For example:

```
10 | 7              // 00001010 | 00000111 ==> 00001111 or 15
flags = flags | f;  // Set flag f
```

When used with `boolean` operands, | is the infrequently used Boolean OR operator described earlier.

Bitwise XOR (^)

This operator combines its two integer operands by performing a Boolean XOR (Exclusive OR) operation on their individual bits. The result has a bit set if the corresponding bits in the two operands are different. If the corresponding operand bits are both ones or both zeros, the result bit is a zero. For example:

```
10 & 7              // 00001010 ^ 00000111 ==> 00001101 or 13
```

When used with `boolean` operands, ^ is the infrequently used Boolean XOR operator.

Left Shift (<<)

The << operator shifts the bits of the left operand left by the number of places specified by the right operand. High-order bits of the left operand are lost, and zero bits are shifted in from the right. Shifting an integer left by n places is equivalent to multiplying that number by 2^n. For example:

```
10 << 1     // 00001010 << 1 = 00010100 = 20 = 10*2
7 << 3      // 00000111 << 3 = 00111000 = 56 = 7*8
-1 << 2     // 0xFFFFFFFF << 2 = 0xFFFFFFFC = -4 = -1*4
```

If the left operand is a `long`, the right operand should be between 0 and 63. Otherwise, the left operand is taken to be an `int`, and the right operand should be between 0 and 31.

Signed Right Shift (>>)

The >> operator shifts the bits of the left operand to the right by the number of places specified by the right operand. The low-order bits of the left operand are shifted away and are lost. The high-order bits shifted in are the same as the original high-order bit of the left operand. In other words, if the left operand is positive, zeros are shifted into the high-order bits. If the left operand is negative, ones are shifted in instead. This technique is known as *sign extension*; it is used to preserve the sign of the left operand. For example:

```
10 >> 1      // 00001010 >> 1 = 00000101 = 5 = 10/2
27 >> 3      // 00011011 >> 3 = 00000011 = 3 = 27/8
-50 >> 2     // 11001110 >> 2 = 11110011 = -13 != -50/4
```

If the left operand is positive and the right operand is n, the >> operator is the same as integer division by 2^n.

Unsigned Right Shift (>>>)

This operator is like the >> operator, except that it always shifts zeros into the high-order bits of the result, regardless of the sign of the left-hand operand. This technique is called *zero extension*; it is appropriate when the left operand is being treated as an unsigned value (despite the fact that Java integer types are all signed). Examples:

```
-50 >>> 2     // 11001110 >>> 2 = 00110011 = 51
0xff >>> 4    // 11111111 >>> 4 = 00001111 = 15  = 255/16
```

Assignment Operators

The assignment operators store, or assign, a value into some kind of variable. The left operand must evaluate to an appropriate local variable, array element, or object field. The right side can be any value of a type compatible with the variable. An assignment expression evaluates to the value that is assigned to the variable. More importantly, however, the expression has the side effect of actually performing the assignment. Unlike all other binary operators, the assignment operators are right-associative, which means that the assignments in `a=b=c` are performed right-to-left, as follows: `a=(b=c)`.

The basic assignment operator is =. Do not confuse it with the equality operator, ==. In order to keep these two operators distinct, I recommend that you read = as "is assigned the value."

In addition to this simple assignment operator, Java also defines 11 other operators that combine assignment with the 5 arithmetic operators and the 6 bitwise and shift operators. For example, the += operator reads the value of the left variable, adds the value of the right operand to it, stores the sum back into the left variable as a side effect, and returns the sum as the value of the expression. Thus, the expression `x+=2` is almost the same `x=x+2`. The difference between these two expressions is that when you use the += operator, the left operand is evaluated

only once. This makes a difference when that operand has a side effect. Consider the following two expressions, which are not equivalent:

```
a[i++] += 2;
a[i++] = a[i++] + 2;
```

The general form of these combination assignment operators is:

```
var op= value
```

This is equivalent (unless there are side effects in `var`) to:

```
var = var op value
```

The available operators are:

```
+=    -=    *=    /=    %=    // Arithmetic operators plus assignment
&=    |=    ^=                // Bitwise operators plus assignment
<<=   >>=   >>>=              // Shift operators plus assignment
```

The most commonly used operators are `+=` and `-=`, although `&=` and `|=` can also be useful when working with `boolean` flags. For example:

```
i += 2;          // Increment a loop counter by 2
c -= 5;          // Decrement a counter by 5
flags |= f;      // Set a flag f in an integer set of flags
flags &= ~f;     // Clear a flag f in an integer set of flags
```

The Conditional Operator

The conditional operator `?:` is a somewhat obscure ternary (three-operand) operator inherited from C. It allows you to embed a conditional within an expression. You can think of it as the operator version of the `if/else` statement. The first and second operands of the conditional operator are separated by a question mark (?), while the second and third operands are separated by a colon (:). The first operand must evaluate to a `boolean` value. The second and third operands can be of any type, but they must both be of the same type.

The conditional operator starts by evaluating its first operand. If it is `true`, the operator evaluates its second operand and uses that as the value of the expression. On the other hand, if the first operand is `false`, the conditional operator evaluates and returns its third operand. The conditional operator never evaluates both its second and third operand, so be careful when using expressions with side effects with this operator. Examples of this operator are:

```
int max = (x > y) ? x : y;
String name = (name != null) ? name : "unknown";
```

Note that the `?:` operator has lower precedence than all other operators except the assignment operators, so parentheses are not usually necessary around the operands of this operator. Many programmers find conditional expressions easier to read if the first operand is placed within parentheses, however. This is especially true because the conditional `if` statement always has its conditional expression written within parentheses.

The instanceof Operator

The `instanceof` operator requires an object or array value as its left operand and the name of a reference type as its right operand. It evaluates to `true` if the object or array is an *instance* of the specified type; it returns `false` otherwise. If the left operand is `null`, `instanceof` always evaluates to `false`. If an `instanceof` expression evaluates to `true`, it means that you can safely cast and assign the left operand to a variable of the type of the right operand.

The `instanceof` operator can be used only with array and object types and values, not primitive types and values. Object and array types are discussed in detail later in this chapter. Examples of `instanceof` are:

```
"string" instanceof String      // True: all strings are instances of String
"" instanceof Object            // True: strings are also instances of Object
new int[] {1} instanceof int[]  // True: the array value is an int array
new int[] {1} instanceof byte[] // False: the array value is not a byte array
new int[] {1} instanceof Object // True: all arrays are instances of Object
null instanceof String          // False: null is never instanceof anything

// Use instanceof to make sure that it is safe to cast an object
if (object instanceof Point) {
  Point p = (Point) object;
}
```

Special Operators

There are five language constructs in Java that are sometimes considered operators and sometimes considered simply part of the basic language syntax. These "operators" are listed in Table 2-5 in order to show their precedence relative to the other true operators. The use of these language constructs is detailed elsewhere in this chapter, but is described briefly here, so that you can recognize these constructs when you encounter them in code examples:

Object member access (`.`)
: An *object* is a collection of data and methods that operate on that data; the data fields and methods of an object are called its members. The dot (`.`) operator accesses these members. If `o` is an expression that evaluates to an object reference, and `f` is the name of a field of the object, `o.f` evaluates to the value contained in that field. If `m` is the name of a method, `o.m` refers to that method and allows it to be invoked using the `()` operator shown later.

Array element access (`[]`)
: An *array* is a numbered list of values. Each element of an array can be referred to by its number, or *index*. The `[]` operator allows you to refer to the individual elements of an array. If `a` is an array, and `i` is an expression that evaluates to an `int`, `a[i]` refers to one of the elements of `a`. Unlike other operators that work with integer values, this operator restricts array index values to be of type `int` or narrower.

Method invocation (())

A *method* is a named collection of Java code that can be run, or *invoked*, by following the name of the method with zero or more comma-separated expressions contained within parentheses. The values of these expressions are the *arguments* to the method. The method processes the arguments and optionally returns a value that becomes the value of the method invocation expression. If `o.m` is a method that expects no arguments, the method can be invoked with `o.m()`. If the method expects three arguments, for example, it can be invoked with an expression such as `o.m(x,y,z)`. Before the Java interpreter invokes a method, it evaluates each of the arguments to be passed to the method. These expressions are guaranteed to be evaluated in order from left to right (which matters if any of the arguments have side effects).

Object creation (new)

In Java, objects are created with the new operator, which is followed by the type of the object to be created and a parenthesized list of arguments to be passed to the object *constructor*. A constructor is a special method that initializes a newly created object, so the object creation syntax is similar to the Java method invocation syntax. For example:

```
new ArrayList();
new Point(1,2)
```

Type conversion or casting (())

As we've already seen, parentheses can also be used as an operator to perform narrowing type conversions, or casts. The first operand of this operator is the type to be converted to; it is placed between the parentheses. The second operand is the value to be converted; it follows the parentheses. For example:

```
(byte) 28          // An integer literal cast to a byte type
(int) (x + 3.14f)  // A floating-point sum value cast to an integer value
(String)h.get(k)   // A generic object cast to a more specific string type
```

Statements

A *statement* is a single "command" that is executed by the Java interpreter. By default, the Java interpreter runs one statement after another, in the order they are written. Many of the statements defined by Java, however, are flow-control statements, such as conditionals and loops, that alter this default order of execution in well-defined ways. Table 2-6 summarizes the statements defined by Java.

Table 2-6: Java Statements

Statement	*Purpose*	*Syntax*
expression	side effects	*var* = *expr*; *expr*++; *method*(); new *Type*();
compound	group statements	{ *statements* }

Table 2-6: Java Statements (continued)

Statement	*Purpose*	*Syntax*
empty	do nothing	;
labeled	name a statement	*label* : *statement*
variable	declare a variable	[final] *type name* [= *value*] [, *name* [= *value*]] ... ;
if	conditional	if (*expr*) *statement* [else *statement*]

switch	conditional	switch (*expr*) {
		[case *expr* : *statements*] ...
		[default: *statements*]
		}

while	loop	while (*expr*) *statement*
do	loop	do *statement* while (*expr*);
for	simplified loop	for (*init* ; *test* ; *increment*) *statement*
break	exit block	break [*label*] ;
continue	restart loop	continue [*label*] ;
return	end method	return [*expr*] ;
synchronized	critical section	synchronized (*expr*) { *statements* }
throw	throw exception	throw *expr* ;
try	handle exception	try { *statements* } [catch (*type name*) { *statements* }] ... [finally { *statements* }]

Expression Statements

As we saw earlier in the chapter, certain types of Java expressions have side effects. In other words, they do not simply evaluate to some value, but also change the program state in some way. Any expression with side effects can be used as a statement simply by following it with a semicolon. The legal types of expression statements are assignments, increments and decrements, method calls, and object creation. For example:

```
a = 1;                              // Assignment
x *= 2;                             // Assignment with operation
i++;                                // Post-increment
--c;                                // Pre-decrement
System.out.println("statement");    // Method invocation
```

Compound Statements

A *compound statement* is any number and kind of statements grouped together within curly braces. You can use a compound statement anywhere a *statement* is required by Java syntax:

```
for(int i = 0; i < 10; i++) {
  a[i]++;                       // Body of this loop is a compound
```

```
    statement. b[i]--;          // It consists of two expression statements
}                               // within curly braces.
```

The Empty Statement

An *empty statement* in Java is written as a single semicolon. The empty statement doesn't do anything, but the syntax is occasionally useful. For example, you can use it to indicate an empty loop body of a `for` loop:

```
for(int i = 0; i < 10; a[i++]++)  // Increment array elements
    /* empty */;                  // Loop body is empty statement
```


Labeled Statements

A *labeled statement* is simply a statement that has been given a name by prepending a identifier and a colon to it. Labels are used by the `break` and `continue` statements. For example:

```
rowLoop: for(int r = 0; r < rows.length; r++) {        // A labeled loop
  colLoop: for(int c = 0; c < columns.length; c++) {   // Another one
    break rowLoop;                                     // Use a label
  }
}
```

Local Variable Declaration Statements

A *local variable*, often simply called a variable, is a symbolic name for a location where a value can be stored that is defined within a method or compound statement. All variables must be declared before they can be used; this is done with a variable declaration statement. Because Java is a strongly typed language, a variable declaration specifies the type of the variable, and only values of that type can be stored in the variable.

In its simplest form, a variable declaration specifies a variable's type and name:

```
int counter;
String s;
```

A variable declaration can also include an *initializer*: an expression that specifies an initial value for the variable. For example:

```
int i = 0;
String s = readLine();
int[] data = {x+1, x+2, x+3};  // Array initializers are documented later
```

The Java compiler does not allow you to use a variable that has not been initialized, so it is usually convenient to combine variable declaration and initialization into a single statement. The initializer expression need not be a literal value or a constant expression that can be evaluated by the compiler; it can be an arbitrarily complex expression whose value is computed when the program is run.

A single variable declaration statement can declare and initialize more than one variable, but all variables must be of the same type. Variable names and optional initializers are separated from each other with commas:

```
int i, j, k;
float x = 1.0, y = 1.0;
String question = "Really Quit?", response;
```

In Java 1.1 and later, variable declaration statements can begin with the `final` keyword. This modifier specifies that once an initial value is specified for the variable, that value is never allowed to change:

```
final String greeting = getLocalLanguageGreeting();
```

C programmers should note that Java variable declaration statements can appear anywhere in Java code; they are not restricted to the beginning of a method or block of code. Local variable declarations can also be integrated with the *initialize* portion of a `for` loop, as we'll discuss shortly.

Local variables can be used only within the method or block of code in which they are defined. This is called their *scope* or *lexical scope*:

```
void method() {           // A generic method
  int i = 0;              // Declare variable i
  while (i < 10) {        // i is in scope here
    int j = 0;            // Declare j; i and j are in scope here
  }                       // j is no longer in scope; can't use it anymore
  System.out.println(i);  // i is still in scope here
}                         // The scope of i ends here
```

The if/else Statement

The `if` statement is the fundamental control statement that allows Java to make decisions or, more precisely, to execute statements conditionally. The `if` statement has an associated expression and statement. If the expression evaluates to `true`, the interpreter executes the statement. If the expression evaluates to `false`, however, the interpreter skips the statement. For example:

```
if (username == null)         // If username is null,
  username = "John Doe";      // define it.
```

Although they look extraneous, the parentheses around the expression are a required part of the syntax for the `if` statement.

As I already mentioned, a block of statements enclosed in curly braces is itself a statement, so we can also write `if` statements that look as follows:

```
if ((address == null) || (address.equals(""))) {
  address = "[undefined]";
  System.out.println("WARNING: no address specified.");
}
```

An `if` statement can include an optional `else` keyword that is followed by a second statement. In this form of the statement, the expression is evaluated, and, if it is `true`, the first statement is executed. Otherwise, the second statement is executed. For example:

```
if (username != null)
  System.out.println("Hello " + username);
else {
  username = askQuestion("What is your name?");
```

```
  System.out.println("Hello " + username + ". Welcome!");
}
```

When you use nested `if/else` statements, some caution is required to ensure that the `else` clause goes with the appropriate `if` statement. Consider the following lines:

```
if (i == j)
  if (j == k)
    System.out.println("i equals k");
else
  System.out.println("i doesn't equal j");    // WRONG!!
```


In this example, the inner `if` statement forms the single statement allowed by the syntax of the outer `if` statement. Unfortunately, it is not clear (except from the hint given by the indentation) which `if` the `else` goes with. And in this example, the indentation hint is wrong. The rule is that an `else` clause like this is associated with the nearest `if` statement. Properly indented, this code looks like this:

```
if (i == j)
  if (j == k)
    System.out.println("i equals k");
  else
    System.out.println("i doesn't equal j");    // WRONG!!
```

This is legal code, but it is clearly not what the programmer had in mind. When working with nested `if` statements, you should use curly braces to make your code easier to read. Here is a better way to write the code:

```
if (i == j) {
  if (j == k)
    System.out.println("i equals k");
}
else {
  System.out.println("i doesn't equal j");
}
```

The else if clause

The `if/else` statement is useful for testing a condition and choosing between two statements or blocks of code to execute. But what about when you need to choose between several blocks of code? This is typically done with an `else if` clause, which is not really new syntax, but a common idiomatic usage of the standard `if/else` statement. It looks like this:

```
if (n == 1) {
   // Execute code block #1
}
else if (n == 2) {
   // Execute code block #2
}
else if (n == 3) {
   // Execute code block #3
}
else {
   // If all else fails, execute block #4
}
```

There is nothing special about this code. It is just a series of `if` statements, where each `if` is part of the `else` clause of the previous statement. Using the `else if` idiom is preferable to, and more legible than, writing these statements out in their fully nested form:

```
if (n == 1) {
  // Execute code block #1
}
else {
  if (n == 2) {
    // Execute code block #2
  }
  else {
    if (n == 3) {
      // Execute code block #3
    }
    else {
      // If all else fails, execute block #4
    }
  }
}
```

The switch Statement

An `if` statement causes a branch in the flow of a program's execution. You can use multiple `if` statements, as shown in the previous section, to perform a multi-way branch. This is not always the best solution, however, especially when all of the branches depend on the value of a single variable. In this case, it is inefficient to repeatedly check the value of the same variable in multiple `if` statements.

A better solution is to use a `switch` statement, which is inherited from the C programming language. Although the syntax of this statement is not nearly as elegant as other parts of Java, the brute practicality of the construct makes it worthwhile. If you are not familiar with the `switch` statement itself, you may at least be familiar with the basic concept, under the name computed goto or jump table. A `switch` statement has an integer expression and a body that contains various numbered entry points. The expression is evaluated, and control jumps to the entry point specified by that value. For example, the following `switch` statement is equivalent to the repeated `if` and `else/if` statements shown in the previous section:

```
switch(n) {
  case 1:                          // Start here if n == 1
    // Execute code block #1
    break;                         // Stop here
  case 2:                          // Start here if n == 2
    // Execute code block #2
    break;                         // Stop here
  case 3:                          // Start here if n == 3
    // Execute code block #3
    break;                         // Stop here
  default:                         // If all else fails...
                                   // Execute code block #4
    break;                         // Stop here
}
```

As you can see from the example, the various entry points into a switch statement are labeled either with the keyword case, followed by an integer value and a colon, or with the special default keyword, followed by a colon. When a switch statement executes, the interpreter computes the value of the expression in parentheses and then looks for a case label that matches that value. If it finds one, the interpreter starts executing the block of code at the first statement following the case label. If it does not find a case label with a matching value, the interpreter starts execution at the first statement following a special-case default: label. Or, if there is no default: label, the interpreter skips the body of the switch statement altogether.

Note the use of the break keyword at the end of each case in the previous code. The break statement is described later in this chapter, but, in this case, it causes the interpreter to exit the body of the switch statement. The case clauses in a switch statement specify only the *starting point* of the desired code. The individual cases are not independent blocks of code, and they do not have any implicit ending point. Therefore, you must explicitly specify the end of each case with a break or related statement. In the absence of break statements, a switch statement begins executing code at the first statement after the matching case label and continues executing statements until it reaches the end of the block. On rare occasions, it is useful to write code like this that falls through from one case label to the next, but 99% of the time you should be careful to end every case and default section with a statement that causes the switch statement to stop executing. Normally you use a break statement, but return and throw also work.

A switch statement can have more than one case clause labeling the same statement. Consider the switch statement in the following method:

```
boolean parseYesOrNoResponse(char response) {
  switch(response) {
    case 'y':
    case 'Y': return true;
    case 'n':
    case 'N': return false;
    default: throw new IllegalArgumentException("Response must be Y or N");
  }
}
```

There are some important restrictions on the switch statement and its case labels. First, the expression associated with a switch statement must have a byte, char, short, or int value. The floating-point and boolean types are not supported, and neither is long, even though long is an integer type. Second, the value associated with each case label must be a constant value or a constant expression the compiler can evaluate. A case label cannot contain a runtime expressions involving variables or method calls, for example. Third, the case label values must be within the range of the data type used for the switch expression. And finally, it is obviously not legal to have two or more case labels with the same value or more than one default label.

The while Statement

Just as the `if` statement is the basic control statement that allows Java to make decisions, the `while` statement is the basic statement that allows Java to perform repetitive actions. It has the following syntax:

```
while (expression)
  statement
```

The `while` statement works by first evaluating the *`expression`*. If it is `false`, the interpreter skips the *`statement`* associated with the loop and moves to the next statement in the program. If it is `true`, however, the *`statement`* that forms the body of the loop is executed, and the *`expression`* is reevaluated. Again, if the value of *`expression`* is `false`, the interpreter moves on to the next statement in the program; otherwise it executes the *`statement`* again. This cycle continues while the *`expression`* remains `true` (i.e., until it evaluates to `false`), at which point the `while` statement ends, and the interpreter moves on to the next statement. You can create an infinite loop with the syntax `while(true)`.

Here is an example `while` loop that prints the numbers 0 to 9:

```
int count = 0;
while (count < 10) {
  System.out.println(count);
  count++;
}
```

As you can see, the variable `count` starts off at 0 in this example and is incremented each time the body of the loop runs. Once the loop has executed 10 times, the expression becomes `false` (i.e., `count` is no longer less than 10), the `while` statement finishes, and the Java interpreter can move to the next statement in the program. Most loops have a counter variable like `count`. The variable names `i`, `j`, and `k` are commonly used as a loop counters, although you should use more descriptive names if it makes your code easier to understand.

The do Statement

A `do` loop is much like a `while` loop, except that the loop expression is tested at the bottom of the loop, rather than at the top. This means that the body of the loop is always executed at least once. The syntax is:

```
do
  statement
while ( expression ) ;
```

There are a couple of differences to notice between the `do` loop and the more ordinary `while` loop. First, the `do` loop requires both the `do` keyword to mark the beginning of the loop and the `while` keyword to mark the end and introduce the loop condition. Also, unlike the `while` loop, the `do` loop is terminated with a semicolon. This is because the `do` loop ends with the loop condition, rather than simply ending with a curly brace that marks the end of the loop body. The following `do` loop prints the same output as the `while` loop shown above:

```
int count = 0;
do {
  System.out.println(count);
  count++;
} while(count < 10);
```

Note that the do loop is much less commonly used than its while cousin. This is because, in practice, it is unusual to encounter a situation where you are sure you always want a loop to execute at least once.

The for Statement

The for statement provides a looping construct that is often more convenient than the while and do loops. The for statement takes advantage of a common looping pattern. Most loops have a counter, or state variable of some kind, that is initialized before the loop starts, tested to determine whether to execute the loop body, and then incremented, or updated somehow, at the end of the loop body before the test expression is evaluated again. The initialization, test, and update steps are the three crucial manipulations of a loop variable, and the for statement makes these three steps an explicit part of the loop syntax:

```
for(initialize ; test ; increment)
   statement
```

This for loop is basically equivalent to the following while loop:*

```
initialize;
while(test) {
  statement;
  increment;
}
```

Placing the *initialize*, *test*, and *increment* expressions at the top of a for loop makes it especially easy to understand what the loop is doing, and it prevents mistakes such as forgetting to initialize or increment the loop variable. The interpreter discards the values of the *initialize* and *increment* expressions, so in order to be useful, these expressions must have side effects. *initialize* is typically an assignment expression, while *increment* is usually an increment, decrement, or some other assignment.

The following for loop prints the numbers 0 to 9, just as the previous while and do loops have done:

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

Notice how this syntax places all the important information about the loop variable on a single line, making it very clear how the loop executes. Placing the increment expression in the for statement itself also simplifies the body of the

* As you'll see when we consider the continue statement, this while loop is not exactly equivalent to the for loop. We'll discuss how to write the true equivalent when we talk about the try/catch/finally statement.

loop to a single statement; we don't even need to use curly braces to produce a statement block.

The `for` loop supports some additional syntax that makes it even more convenient to use. Because many loops use their loop variables only within the loop, the `for` loop allows the *`initialize`* expression to be a full variable declaration, so that the variable is scoped to the body of the loop and is not visible outside of it. For example:

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

Furthermore, the `for` loop syntax does not restrict you to writing loops that use only a single variable. Both the *`initialize`* and *`increment`* expressions of a `for` loop can use a comma to separate multiple initializations and increment expressions. For example:

```
for(int i = 0, j = 10 ; i < 10 ; i++, j--)
    sum += i * j;
```

Even though all the examples so far have counted numbers, `for` loops are not restricted to loops that count numbers. For example, you might use a `for` loop to iterate through the elements of a linked list:

```
for(Node n = listHead; n != null; n = n.nextNode())
  process(n);
```

The *`initialize`*, *`test`*, and *`increment`* expressions of a `for` loop are all optional; only the semicolons that separate the expressions are required. If the *`test`* expression is omitted, it is assumed to be `true`. Thus, you can write an infinite loop as `for(;;)`.

The break Statement

A `break` statement causes the Java interpreter to skip immediately to the end of a containing statement. We have already seen the `break` statement used with the `switch` statement. The `break` statement is most often written as simply the keyword `break` followed by a semicolon:

```
break;
```

When used in this form, it causes the Java interpreter to immediately exit the innermost containing `while`, `do`, `for`, or `switch` statement. For example:

```
for(int i = 0; i < data.length; i++) {  // Loop through the data array.
  if (data[i] == target) {              // When we find what we're looking for,
    index = i;                          // remember where we found it
    break;                              // and stop looking!
  }
}   // The Java interpreter goes here after executing break
```

The `break` statement can also be followed by the name of a containing labeled statement. When used in this form, `break` causes the Java interpreter to immediately exit from the named block, which can be any kind of statement, not just a loop or `switch`. For example:

```
testfornull: if (data != null) {              // If the array is defined,
  for(int row = 0; row < numrows; row++) {    // loop through one dimension,
    for(int col = 0; col < numcols; col++) { // then loop through the other.
      if (data[row][col] == null)             // If the array is missing data,
        break testfornull;                    // treat the array as undefined.
    }
  }
} // Java interpreter goes here after executing break testfornull
```


The continue Statement

While a `break` statement exits a loop, a `continue` statement quits the current iteration of a loop and starts the next one. `continue`, in both its unlabeled and labeled forms, can be used only within a `while`, `do`, or `for` loop. When used without a label, `continue` causes the innermost loop to start a new iteration. When used with a label that is the name of a containing loop, it causes the named loop to start a new iteration. For example:

```
for(int i = 0; i < data.length; i++) {  // Loop through data.
  if (data[i] == -1)                    // If a data value is missing,
    continue;                           // skip to the next iteration.
  process(data[i]);                     // Process the data value.
}
```

`while`, `do`, and `for` loops differ slightly in the way that `continue` starts a new iteration:

- With a `while` loop, the Java interpreter simply returns to the top of the loop, tests the loop condition again, and, if it evaluates to `true`, executes the body of the loop again.
- With a `do` loop, the interpreter jumps to the bottom of the loop, where it tests the loop condition to decide whether to perform another iteration of the loop.
- With a `for` loop, the interpreter jumps to the top of the loop, where it first evaluates the *`increment`* expression and then evaluates the *`test`* expression to decide whether to loop again. As you can see, the behavior of a `for` loop with a `continue` statement is different from the behavior of the "basically equivalent" `while` loop I presented earlier; *`increment`* gets evaluated in the `for` loop, but not in the equivalent `while` loop.

The return Statement

A `return` statement tells the Java interpreter to stop executing the current method. If the method is declared to return a value, the `return` statement is followed by an expression. The value of the expression becomes the return value of the method. For example, the following method computes and returns the square of a number:

```
double square(double x) {      // A method to compute x squared
  return x * x;                // Compute and return a value
}
```

Some methods are declared `void` to indicate they do not return any value. The Java interpreter runs methods like this by executing its statements one by one until it reaches the end of the method. After executing the last statement, the interpreter

returns implicitly. Sometimes, however, a void method has to return explicitly before reaching the last statement. In this case, it can use the return statement by itself, without any expression. For example, the following method prints, but does not return, the square root of its argument. If the argument is a negative number, it returns without printing anything:

```
void printSquareRoot(double x) {     // A method to print square root of x
  if (x < 0) return;                 // If x is negative, return explicitly
  System.out.println(Math.sqrt(x));  // Print the square root of x
}                                    // End of method: return implicitly
```

The synchronized Statement

Since Java is a multithreaded system, you must often take care to prevent multiple threads from modifying an object simultaneously in a way that might corrupt the object's state. Sections of code that must not be executed simultaneously are known as *critical sections*. Java provides the synchronized statement to protect these critical sections. The syntax is:

```
synchronized ( expression ) {
  statements
}
```

expression is an expression that must evaluate to an object or an array. The *statements* constitute the code of the critical section and must be enclosed in curly braces. Before executing the critical section, the Java interpreter first obtains an exclusive lock on the object or array specified by *expression*. It holds the lock until it is finished running the critical section, then releases it. While a thread holds the lock on an object, no other thread can obtain that lock. Therefore, no other thread can execute this or any other critical sections that require a lock on the same object. If a thread cannot immediately obtain the lock required to execute a critical section, it simply waits until the lock becomes available.

Note that you do not have to use the synchronized statement unless your program creates multiple threads that share data. If only one thread ever accesses a data structure, there is no need to protect it with synchronized. When you do have to use synchronized, it might be in code like the following:

```
public static void SortIntArray(int[] a) {
  // Sort the array a. This is synchronized so that some other thread
  // cannot change elements of the array while we're sorting it (at
  // least not other threads that protect their changes to the array
  // with synchronized).
  synchronized (a) {
    // Do the array sort here...
  }
}
```

The synchronized keyword is also available as a modifier in Java and is more commonly used in this form than as a statement. When applied to a method, the synchronized keyword indicates that the entire method is a critical section. For a synchronized class method (a static method), Java obtains an exclusive lock on the class before executing the method. For a synchronized instance method, Java

obtains an exclusive lock on the class instance. (Class and instance methods are discussed in Chapter 3.)

The throw Statement

An *exception* is a signal that indicates some sort of exceptional condition or error has occurred. To *throw* an exception is to signal an exceptional condition. To *catch* an exception is to handle it—to take whatever actions are necessary to recover from it.

In Java, the `throw` statement is used to throw an exception:

```
throw expression ;
```

The *`expression`* must evaluate to an exception object that describes the exception or error that has occurred. We'll talk more about types of exceptions shortly; for now, all you need to know is that an exception is represented by an object. Here is some example code that throws an exception:

```
public static double factorial(int x) {
  if (x < 0)
    throw new IllegalArgumentException("x must be >= 0"));
  double fact;
  for(fact=1.0; x > 1; fact *= x, x--)
    /* empty */ ;          // Note use of the empty statement
  return fact;
}
```

When the Java interpreter executes a `throw` statement, it immediately stops normal program execution and starts looking for an exception handler that can catch, or handle, the exception. Exception handlers are written with the `try/catch/finally` statement, which is described in the next section. The Java interpreter first looks at the enclosing block of code to see if it has an associated exception handler. If so, it exits that block of code and starts running the exception-handling code associated with the block. After running the exception handler, the interpreter continues execution at the statement immediately following the handler code.

If the enclosing block of code does not have an appropriate exception handler, the interpreter checks the next higher enclosing block of code in the method. This continues until a handler is found. If the method does not contain an exception handler that can handle the exception thrown by the `throw` statement, the interpreter stops running the current method and returns to the caller. Now the interpreter starts looking for an exception handler in the blocks of code of the calling method. In this way, exceptions propagate up through the lexical structure of Java methods, up the call stack of the Java interpreter. If the exception is never caught, it propagates all the way up to the `main()` method of the program. If it is not handled in that method, the Java interpreter prints an error message, prints a stack trace to indicate where the exception occurred, and then exits.

Exception types

An exception in Java is an object. The type of this object is `java.lang.Throwable`, or more commonly, some subclass of `Throwable` that more specifically describes

the type of exception that occurred.* `Throwable` has two standard subclasses: `java.lang.Error` and `java.lang.Exception`. Exceptions that are subclasses of `Error` generally indicate unrecoverable problems: the virtual machine has run out of memory, or a class file is corrupted and cannot be read, for example. Exceptions of this sort can be caught and handled, but it is rare to do so. Exceptions that are subclasses of `Exception`, on the other hand, indicate less severe conditions. These are exceptions that can be reasonably caught and handled. They include such exceptions as `java.io.EOFException`, which signals the end of a file, and `java.lang.ArrayIndexOutOfBoundsException`, which indicates that a program has tried to read past the end of an array. In this book, I use the term "exception" to refer to any exception object, regardless of whether the type of that exception is `Exception` or `Error`.

Since an exception is an object, it can contain data, and its class can define methods that operate on that data. The `Throwable` class and all its subclasses include a `String` field that stores a human-readable error message that describes the exceptional condition. It's set when the exception object is created and can be read from the exception with the `getMessage()` method. Most exceptions contain only this single message, but a few add other data. The `java.io.InterruptedIOException`, for example, adds a field named `bytesTransferred` that specifies how much input or output was completed before the exceptional condition interrupted it.

Declaring exceptions

In addition to making a distinction between `Error` and `Exception` classes, the Java exception-handling scheme also makes a distinction between checked and unchecked exceptions. Any exception object that is an `Error` is unchecked. Any exception object that is an `Exception` is checked, unless it is a subclass of `java.lang.RuntimeException`, in which case it is unchecked. (`RuntimeException` is a subclass of `Exception`.) The reason for this distinction is that virtually any method can throw an unchecked exception, at essentially any time. There is no way to predict an `OutOfMemoryError`, for example, and any method that uses objects or arrays can throw a `NullPointerException` if it is passed an invalid `null` argument. Checked exceptions, on the other hand, arise only in specific, well-defined circumstances. If you try to read data from a file, for example, you must at least consider the possibility that a `FileNotFoundException` will be thrown if the specified file cannot be found.

Java has different rules for working with checked and unchecked exceptions. If you write a method that throws a checked exception, you must use a `throws` clause to declare the exception in the method signature. The reason these types of exceptions are called checked exceptions is that the Java compiler checks to make sure you have declared them in method signatures and produces a compilation error if you have not. The `factorial()` method shown earlier throws an exception of type `java.lang.IllegalArgumentException`. This is a subclass of `RuntimeException`, so it is an unchecked exception, and we do not have to declare it with a `throws` clause (although we can if we want to be explicit).

---

* We haven't talked about subclasses yet; they are covered in detail in Chapter 3

Even if you never throw an exception yourself, there are times when you must use a throws clause to declare an exception. If your method calls a method that can throw a checked exception, you must either include exception-handling code to handle that exception or use throws to declare that your method can also throw that exception.

How do you know if the method you are calling can throw a checked exception? You can look at its method signature to find out. Or, failing that, the Java compiler will tell you (by reporting a compilation error) if you've called a method whose exceptions you must handle or declare. The following method reads the first line of text from a named file. It uses methods that can throw various types of java.io.IOException objects, so it declares this fact with a throws clause:


```
public static String readFirstLine(String filename) throws IOException {
  BufferedReader in = new BufferedReader(new FileReader(filename));
  return in.readLine();
}
```

We'll talk more about method declarations and method signatures later in this chapter.

The try/catch/finally Statement

The try/catch/finally statement is Java's exception-handling mechanism. The try clause of this statement establishes a block of code for exception handling. This try block is followed by zero or more catch clauses, each of which is a block of statements designed to handle a specific type of exception. The catch clauses are followed by an optional finally block that contains cleanup code guaranteed to be executed regardless of what happens in the try block. Both the catch and finally clauses are optional, but every try block must be accompanied by at least one or the other. The try, catch, and finally blocks all begin and end with curly braces. These are a required part of the syntax and cannot be omitted, even if the clause contains only a single statement.

The following code illustrates the syntax and purpose of the try/catch/finally statement:

```
try {
  // Normally this code runs from the top of the block to the bottom
  // without problems. But it can sometimes throw an exception,
  // either directly with a throw statement or indirectly by calling
  // a method that throws an exception.
}
catch (SomeException e1) {
  // This block contains statements that handle an exception object
  // of type SomeException or a subclass of that type. Statements in
  // this block can refer to that exception object by the name e1.
}
catch (AnotherException e2) {
  // This block contains statements that handle an exception object
  // of type AnotherException or a subclass of that type. Statements
  // in this block can refer to that exception object by the name e2.
}
```

```
finally {
  // This block contains statements that are always executed
  // after we leave the try clause, regardless of whether we leave it:
  //   1) normally, after reaching the bottom of the block;
  //   2) because of a break, continue, or return statement;
  //   3) with an exception that is handled by a catch clause above; or
  //   4) with an uncaught exception that has not been handled.
  // If the try clause calls System.exit(), however, the interpreter
  // exits before the finally clause can be run.
}
```

try

The `try` clause simply establishes a block of code that either has its exceptions handled or needs special cleanup code to be run when it terminates for any reason. The `try` clause by itself doesn't do anything interesting; it is the `catch` and `finally` clauses that do the exception-handling and cleanup operations.

catch

A `try` block can be followed by zero or more `catch` clauses that specify code to handle various types of exceptions. Each `catch` clause is declared with a single argument that specifies the type of exceptions the clause can handle and also provides a name the clause can use to refer to the exception object it is currently handling. The type and name of an exception handled by a `catch` clause are exactly like the type and name of an argument passed to a method, except that for a `catch` clause, the argument type must be `Throwable` or one of its subclasses.

When an exception is thrown, the Java interpreter looks for a `catch` clause with an argument of the same type as the exception object or a superclass of that type. The interpreter invokes the first such `catch` clause it finds. The code within a `catch` block should take whatever action is necessary to cope with the exceptional condition. If the exception is a `java.io.FileNotFoundException` exception, for example, you might handle it by asking the user to check his spelling and try again. It is not required to have a `catch` clause for every possible exception; in some cases the correct response is to allow the exception to propagate up and be caught by the invoking method. In other cases, such as a programming error signaled by `NullPointerException`, the correct response is probably not to catch the exception at all, but allow it to propagate and have the Java interpreter exit with a stack trace and an error message.

finally

The `finally` clause is generally used to clean up after the code in the `try` clause (e.g., close files, shut down network connections). What is useful about the `finally` clause is that it is guaranteed to be executed if any portion of the `try` block is executed, regardless of how the code in the `try` block completes. In fact, the only way a `try` clause can exit without allowing the `finally` clause to be executed is by invoking the `System.exit()` method, which causes the Java interpreter to stop running.

In the normal case, control reaches the end of the `try` block and then proceeds to the `finally` block, which performs any necessary cleanup. If control leaves the `try` block because of a `return`, `continue`, or `break` statement, the `finally` block is executed before control transfers to its new destination.

If an exception occurs in the `try` block, and there is an associated `catch` block to handle the exception, control transfers first to the `catch` block and then to the `finally` block. If there is no local `catch` block to handle the exception, control transfers first to the `finally` block, and then propagates up to the nearest containing `catch` clause that can handle the exception.

If a `finally` block itself transfers control with a `return`, `continue`, `break`, or `throw` statement or by calling a method that throws an exception, the pending control transfer is abandoned, and this new transfer is processed. For example, if a `finally` clause throws an exception, that exception replaces any exception that was in the process of being thrown. If a `finally` clause issues a `return` statement, the method returns normally, even if an exception has been thrown and has not been handled yet.

`try` and `finally` can be used together without exceptions or any `catch` clauses. In this case, the `finally` block is simply cleanup code that is guaranteed to be executed, regardless of any `break`, `continue`, or `return` statements within the `try` clause.

In previous discussions of the `for` and `continue` statements, we've seen that a `for` loop cannot be naively translated into a `while` loop because the `continue` statement behaves slightly differently when used in a `for` loop than it does when used in a `while` loop. The `finally` clause gives us a way to write a `while` loop that is truly equivalent to a `for` loop. Consider the following generalized `for` loop:

```
for( initialize ; test ; increment )
   statement
```

The following `while` loop behaves the same, even if the `statement` block contains a `continue` statement:

```
initialize ;
while ( test ) {
  try { statement }
  finally { increment ; }
}
```

Methods

A *method* is a named collection of Java statements that can be invoked by other Java code. When a method is invoked, it is passed zero or more values known as arguments. The method performs some computations and, optionally, returns a value. A method invocation is an expression that is evaluated by the Java interpreter. Because method invocations can have side effects, however, they can also be used as expression statements.

You already know how to define the body of a method; it is simply an arbitrary sequence of statements enclosed within curly braces. What is more interesting about a method is its *signature*. The signature specifies:

- The name of the method
- The type and name of each of the parameters used by the method
- The type of the value returned by the method
- The exception types the method can throw
- Various method modifiers that provide additional information about the method

A method signature defines everything you need to know about a method before calling it. It is the method *specification* and defines the API for the method. The reference section of this book is essentially a list of method signatures for all publicly accessible methods of all publicly accessible classes of the Java platform. In order to use the reference section of this book, you need to know how to read a method signature. And, in order to write Java programs, you need to know how to define your own methods, each of which begins with a method signature.

A method signature looks like this:

```
modifiers type name ( paramlist ) [ throws exceptions ]
```

The signature (the method specification) is followed by the method body (the method implementation), which is simply a sequence of Java statements enclosed in curly braces. In certain cases (described in Chapter 3), the implementation is omitted, and the method body is replaced with a single semicolon.

Here are some example method definitions. The method bodies have been omitted, so we can focus on the signatures:

```
public static void main(String[] args) { ... }
public final synchronized int indexOf(Object element, int startIndex) { ... }
double distanceFromOrigin() { ... }
static double squareRoot(double x) throws IllegalArgumentException { ... }
protected abstract String readText(File f, String encoding)
          throws FileNotFoundException, UnsupportedEncodingException;
```

modifiers is zero or more special modifier keywords, separated from each other by spaces. A method might be declared with the `public` and `static` modifiers, for example. Other valid method modifiers are `abstract`, `final`, `native`, `private`, `protected`, and `synchronized`. The meanings of these modifiers are not important here; they are discussed in Chapter 3.

The *type* in a method signature specifies the return type of the method. If the method returns a value, this is the name of a primitive type, an array type, or a class. If the method does not return a value, *type* must be `void`. A *constructor* is a special type of method used to initialize newly created objects. As we'll see in Chapter 3, constructors are defined just like methods, except that their signatures do not include this *type* specification.

The *name* of a method follows the specification of its modifiers and type. Method names, like variable names, are Java identifiers and, like all Java identifiers, can

use any characters of the Unicode character set. It is legal (and sometimes useful) to define more than one method with the same name, as long as each version of the method has a different parameter list. Defining multiple methods with the same name is called *method overloading*. The `System.out.println()` method we've seen so much of is an overloaded method. There is one method by this name that prints a string and other methods by the same name that print the values of the various primitive types. The Java compiler decides which method to call based on the type of the argument passed to the method.

When you are defining a method, the name of the method is always followed by the method's parameter list, which must be enclosed in parentheses. The parameter list defines zero or more arguments that are passed to the method. The parameter specifications, if there are any, each consist of a type and a name and are separated from each other by commas (if there are multiple parameters). When a method is invoked, the argument values it is passed must match the number, type, and order of the parameters specified in this method signature line. The values passed need not have exactly the same type as specified in the signature, but they must be convertible to those types without casting. C and C++ programmers should note that when a Java method expects no arguments, its parameter list is simply `()`, not `(void)`.

The final part of a method signature is the `throws` clause, which I first described when we discussed the `throw` statement. If a method uses the `throw` statement to throw a checked exception, or if it calls some other method that throws a checked exception and does not catch or handle that exception, the method must declare that it can throw that exception. If a method can throw one or more checked exceptions, it specifies this by placing the `throws` keyword after the argument list and following it by the name of the exception class or classes it can throw. If a method does not throw any exceptions, it does not use the `throws` keyword. If a method throws more than one type of exception, separate the names of the exception classes from each other with commas.

Classes and Objects

Now that we have introduced operators, expressions, statements, and methods, we can finally talk about classes. A *class* is a named collection of fields that hold data values and methods that operate on those values. Some classes also contain nested inner classes. Classes are the most fundamental structural element of all Java programs. You cannot write Java code without defining a class. All Java statements appear within methods, and all methods are defined within classes.

Classes are more than just another structural level of Java syntax. Just as a cell is the smallest unit of life that can survive and reproduce on its own, a class is the smallest unit of Java code that can stand alone. The Java compiler and interpreter do not recognize fragments of Java code that are smaller than a class. A class is the basic unit of execution for Java, which makes classes very important. Java actually defines another construct, called an *interface*, that is quite similar to a class. The distinction between classes and interfaces will become clear in Chapter 3, but for now I'll use the term "class" to mean either a class or an interface.

Classes are important for another reason: every class defines a new data type. For example, you can define a class named `Point` to represent a data point in the two-dimensional Cartesian coordinate system. This class can define fields (each of type `double`) to hold the X and Y coordinates of a point and methods to manipulate and operate on the point. The `Point` class is a new data type.

When discussing data types, it is important to distinguish between the data type itself and the values the data type represents. `char` is a data type: it represents Unicode characters. But a `char` value represents a single specific character. A class is a data type; the value of a class type is called an *object*. We use the name class because each class defines a type (or kind, or species, or class) of objects. The `Point` class is a data type that represents X,Y points, while a `Point` object represents a single specific X,Y point. As you might imagine, classes and their objects are closely linked. In the sections that follow, we will be discussing both.

Defining a Class

Here is a possible definition of the `Point` class we have been discussing:

```
/** Represents a Cartesian (x,y) point */
public class Point {
  public double x, y;                    // The coordinates of the point.
  public Point(double x, double y) {     // A constructor that
    this.x = x; this.y = y;              // initializes the fields.
  }

  public double distanceFromOrigin() {   // A method that operates on
    return Math.sqrt(x*x + y*y);         // the x and y fields.
  }
}
```

This class definition is stored in a file named *Point.java* and compiled to a file named *Point.class*, at which point it is available for use by Java programs and other classes. This class definition is provided here for completeness and to provide context, but don't expect to understand all the details just yet; most of Chapter 3 is devoted to the topic of defining classes. Do pay extra attention to the first (non-comment) line of the class definition, however. Just as the first line of a method definition—the method signature—defines the API for the method, this line defines the basic API for a class (as described in the next chapter).

Keep in mind that you don't have to define every class you want to use in a Java program. The Java platform consists of over 1500 predefined classes that are guaranteed to be available on every computer that runs Java.

Creating an Object

Now that we have defined the `Point` class as a new data type, we can use the following line to declare a variable that holds a `Point` object:

```
Point p;
```

Declaring a variable to hold a `Point` object does not create the object itself, however. To actually create an object, you must use the `new` operator. This keyword is followed by the object's class (i.e., its type) and an optional argument list in

parentheses. These arguments are passed to the constructor method for the class, which initializes internal fields in the new object:

```
// Create a Point object representing (2,-3.5) and store it in variable p
Point p = new Point(2.0, -3.5);

// Create some other objects as well
Date d = new Date();         // A Date object that represents the current time
Vector list = new Vector();  // A Vector object to hold a list of objects
```


The `new` keyword is by far the most common way to create objects in Java. There are a few other ways that are worth mentioning, however. First, there are a couple of classes that are so important that the Java language defines special literal syntax for creating objects of those types (as we'll discuss in the next section). Second, Java supports a dynamic loading mechanism that allows programs to load classes and create instances of those classes dynamically. This dynamic instantiation is done with the `newInstance()` methods of `java.lang.Class` and `java.lang.Constructor`. Finally, in Java 1.1 and later, objects can also be created by deserializing them. In other words, an object that has had its state saved, or serialized, usually to a file, can be recreated using the `java.io.ObjectInputStream` class.

Object Literals

As I just said, Java defines special syntax for creating instances of two very important classes. The first class is `String`, which represents text as a string of characters. Since programs usually communicate with their users through the written word, the ability to manipulate strings of text is quite important in any programming language. In some languages, strings are a primitive type, on a par with integers and characters. In Java, however, strings are objects; the data type used to represent text is the `String` class.

Because strings are such a fundamental data type, Java allows you to include text literally in programs by placing it between double-quote (") characters. For example:

```
String name = "David";
System.out.println("Hello, " + name);
```

Don't confuse the double-quote characters that surround string literals with the single-quote (or apostrophe) characters that surround `char` literals. String literals can contain any of the escape sequences `char` literals can (see Table 2-3). Escape sequences are particularly useful for embedding double-quote characters within double-quoted string literals. For example:

```
String story = "\t\"How can you stand it?\" he asked sarcastically.\n";
```

String literals can be only a single line long. Java does not support any kind of continuation-character syntax that allows two separate lines to be treated as a single line. If you need to represent a long string of text that does not fit on a single line, break it into independent string literals and use the + operator to concatenate the literals. For example:

```
String s = "This is a test of the         // This is illegal; string literals
           emergency broadcast system";  // cannot be broken across lines.
```

```
String s = "This is a test of the " +     // Do this instead.
           "emergency broadcast system";
```

This concatenation of literals is done when your program is compiled, not when it is run, so you do not need to worry about any kind of performance penalty.

The second class that supports its own special object literal syntax is the class named `Class`. `Class` is a (self-referential) data type that represents all Java data types, including primitive types and array types, not just class types. To include a `Class` object literally in a Java program, follow the name of any data type with `.class`. For example:

```
Class typeInt = int.type;
Class typeIntArray = int[].type;
Class typePoint = Point.class;
```

This feature is supported by Java 1.1 and later.

The Java reserved word `null` is a special literal that can be used with any class. Instead of representing a literal object, it represents the absence of an object. For example:

```
String s = null;
Point p = null;
```

Finally, objects can also be included literally in a Java program through the use of a construct known as an anonymous inner class. Anonymous classes are discussed in Chapter 3.

Using an Object

Now that we've seen how to define classes and instantiate them by creating objects, we need to look at the Java syntax that allows us to use those objects. Recall that a class defines a collection of fields and methods. Each object has its own copies of those fields and has access to those methods. We use the dot character (`.`) to access the named fields and methods of an object. For example:

```
Point p = new Point(2, 3);          // Create an object
double x = p.x;                     // Read a field of the object
p.y = p.x * p.x;                    // Set the value of a field
double d = p.distanceFromOrigin(); // Access a method of the object
```

This syntax is central to object-oriented programming in Java, so you'll see it a lot. Note, in particular, the expression `p.distanceFromOrigin()`. This tells the Java compiler to look up a method named `distanceFromOrigin()` defined by the class `Point` and use that method to perform a computation on the fields of the object `p`. We'll cover the details of this operation in Chapter 3.

Array Types

Array types are the second kind of reference types in Java. An array is an ordered collection, or numbered list, of values. The values can be primitive values, objects, or even other arrays, but all of the values in an array must be of the same type.

The type of the array is the type of the values it holds, followed by the characters `[]`. For example:

```
byte b;                          // byte is a primitive type
byte[] arrayOfBytes;             // byte[] is an array type: array of byte
byte[][] arrayOfArrayOfBytes;    // byte[][] is another type: array of byte[]
Point[] points;                  // Point[] is an array of Point objects
```

For compatibility with C and C++, Java also supports another syntax for declaring variables of array type. In this syntax, one or more pairs of square brackets follow the name of the variable, rather than the name of the type:

```
byte arrayOfBytes[];             // Same as byte[] arrayOfBytes
byte arrayOfArrayOfBytes[][];    // Same as byte[][] arrayOfArrayOfBytes
byte[] arrayOfArrayOfBytes[];    // Ugh! Same as byte[][] arrayOfArrayOfBytes
```

This is almost always a confusing syntax, however, and it is not recommended.

With classes and objects, we have separate terms for the type and the values of that type. With arrays, the single word array does double duty as the name of both the type and the value. Thus, we can speak of the array type `int[]` (a type) and an array of `int` (a particular array value). In practice, it is usually clear from context whether a type or a value is being discussed.

Creating Arrays

To create an array value in Java, you use the `new` keyword, just as you do to create an object. Arrays don't need to be initialized like objects do, however, so you don't pass a list of arguments between parentheses. What you must specify, though, is how big you want the array to be. If you are creating a `byte[]`, for example, you must specify how many `byte` values you want it to hold. Array values have a fixed size in Java. Once an array is created, it can never grow or shrink. Specify the desired size of your array as a non-negative integer between square brackets:

```
byte[] buffer = new byte[1024];
String[] lines = new String[50];
```

When you create an array with this syntax, each of the values held in the array is automatically initialized to its default value. This is `false` for `boolean` values, `'\u0000'` for `char` values, 0 for integer values, 0.0 for floating-point values, and `null` for objects or array values.

Using Arrays

Once you've created an array with the `new` operator and the square-bracket syntax, you also use square brackets to access the individual values contained in the array. Remember that an array is an ordered collection of values. The elements of an array are numbered sequentially, starting with 0. The number of an array element refers to the element. This number is often called the *index*, and the process of looking up a numbered value in an array is sometimes called *indexing* the array.

To refer to a particular element of an array, simply place the index of the desired element in square brackets after the name of the array. For example:

```
String[] responses = new String[2];   // Create an array of two strings
responses[0] = "Yes";                 // Set the first element of the array
responses[1] = "No";                  // Set the second element of the array

// Now read these array elements
System.out.println(question + " (" + responses[0] + "/" +
                   responses[1] + " ): ");
```

In some programming languages, such as C and C++, it is a common bug to write code that tries to read or write array elements that are past the end of the array. Java does not allow this. Every time you access an array element, the Java interpreter automatically checks that the index you have specified is valid. If you specify a negative index or an index that is greater than the last index of the array, the interpreter throws an exception of type `ArrayIndexOutOfBoundsException`. This prevents you from reading or writing nonexistent array elements.

Array index values are integers; you cannot index an array with a floating-point value, a `boolean`, an object, or another array. `char` values can be converted to `int` values, so you *can* use characters as array indexes. Although `long` is an integer data type, `long` values cannot be used as array indexes. This may seem surprising at first, but consider that an `int` index supports arrays with over two billion elements. An `int[]` with this many elements would require eight gigabytes of memory. When you think of it this way, it is not surprising that `long` values are not allowed as array indexes.

Besides setting and reading the value of array elements, there is one other thing you can do with an array value. Recall that whenever we create an array, we must specify the number of elements the array holds. This value is referred to as the length of the array; it is an intrinsic property of the array. If you need to know the length of the array, append `.length` to the array name:

```
if (errorCode < errorMessages.length)
  System.out.println(errorMessages[errorCode]);
```

`.length` is special Java syntax for arrays. An expression like `a.length` looks as though it refers to a field of an object `a`, but this is not actually the case. The `.length` syntax can be used only to read the length of an array. It cannot be used to set the length of an array (because, in Java, an array has a fixed length that can never change).

In the previous example, the array index within square brackets is a variable, not an integer literal. In fact, arrays are most often used with loops, particularly `for` loops, where they are indexed using a variable that is incremented or decremented each time through the loop:

```
int[] values;                          // Array elements initialized elsewhere
int total = 0;                         // Store sum of elements here
for(int i = 0; i < values.length; i++) // Loop through array elements
  total += values[i];                  // Add them up
```

In Java, the first element of an array is always element number 0. If you are accustomed to a programming language that numbers array elements beginning with 1, this will take some getting used to. For an array `a`, the first element is `a[0]`, the second element is `a[1]`, and the last element is:

```
a[a.length - 1]         // The last element of any array named a
```

Array Literals

The `null` literal used to represent the absence of an object can also be used to represent the absence of an array. For example:

```
char[] password = null;
```


In addition to the `null` literal, Java also defines special syntax that allows you to specify array values literally in your programs. There are actually two different syntaxes for array literals. The first, and more commonly used, syntax can be used only when declaring a variable of array type. It combines the creation of the array object with the initialization of the array elements:

```
int[] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128};
```

This creates an array that contains the eight `int` elements listed within the curly braces. Note that we don't use the `new` keyword or specify the type of the array in this array literal syntax. The type is implicit in the variable declaration of which the initializer is a part. Also, the array length is not specified explicitly with this syntax; it is determined implicitly by counting the number of elements listed between the curly braces. There is a semicolon following the close curly brace in this array literal. This is one of the fine points of Java syntax. When curly braces delimit classes, methods, and compound statements, they are not followed by semicolons. However, for this array literal syntax, the semicolon is required to terminate the variable declaration statement.

The problem with this array literal syntax is that it works only when you are declaring a variable of array type. Sometimes you need to do something with an array value (such as pass it to a method) but are going to use the array only once, so you don't want to bother assigning it to a variable. In Java 1.1 and later, there is an array literal syntax that supports this kind of anonymous arrays (so called because they are not assigned to variables, so they don't have names). This kind of array literal looks as follows:

```
// Call a method, passing an anonymous array literal that contains two strings
String response = askQuestion("Do you want to quit?",
                              new String[] {"Yes", "No"});

// Call another method with an anonymous array (of anonymous objects)
double d = computeAreaOfTriangle(new Point[] { new Point(1,2),
                                               new Point(3,4),
                                               new Point(3,2) });
```

With this syntax, you use the `new` keyword and specify the type of the array, but the length of the array is not explicitly specified.

It is important to understand that the Java Virtual Machine architecture does not support any kind of efficient array initialization. In other words, array literals are created and initialized when the program is run, not when the program is compiled. Consider the following array literal:

```
int[] perfectNumbers = {6, 28};
```

This is compiled into Java byte codes that are equivalent to:

```
int[] perfectNumbers = new int[2];
perfectNumbers[0] = 6;
perfectNumbers[1] = 28;
```

Thus, if you want to include a large amount of data in a Java program, it may not be a good idea to include that data literally in an array, since the Java compiler has to create lots of Java byte codes to initialize the array, and then the Java interpreter has to laboriously execute all that initialization code. In cases like this, it is better to store your data in an external file and read it into the program at runtime.

The fact that Java does all array initialization explicitly at runtime has an important corollary, however. It means that the elements of an array literal can be arbitrary expressions that are computed at runtime, rather than constant expressions that are resolved by the compiler. For example:

```
Point[] points = { circle1.getCenterPoint(), circle2.getCenterPoint() };
```

Multidimensional Arrays

As we've seen, an array type is simply the element type followed by a pair of square brackets. An array of `char` is `char[]`, and an array of arrays of `char` is `char[][]`. When the elements of an array are themselves arrays, we say that the array is *multidimensional*. In order to work with multidimensional arrays, there are a few additional details you must understand.

Imagine that you want to use a multidimensional array to represent a multiplication table:

```
int[][] products;      // A multiplication table
```

Each of the pairs of square brackets represents one dimension, so this is a two-dimensional array. To access a single `int` element of this two-dimensional array, you must specify two index values, one for each dimension. Assuming that this array was actually initialized as a multiplication table, the `int` value stored at any given element would be the product of the two indexes. That is, `products[2][4]` would be 8, and `products[3][7]` would be 21.

To create a new multidimensional array, use the `new` keyword and specify the size of both dimensions of the array. For example:

```
int[][] products = new int[10][10];
```

In some languages, an array like this would be created as a single block of 100 `int` values. Java does not work this way. This line of code does three things:

- Declares a variable named `products` to hold an array of arrays of `int`.
- Creates a 10-element array to hold 10 arrays of `int`.
- Creates 10 more arrays, each of which is a 10-element array of `int`. It assigns each of these 10 new arrays to the elements of the initial array. The default value of every `int` element of each of these 10 new arrays is 0.

To put this another way, the previous single line of code is equivalent to the following code:

```
int[][] products = new int[10][];    // An array to hold ten int[] values.
for(int i = 0; i < 10; i++)          // Loop ten times...
  products[i] = new int[10];         // ...and create ten arrays.
```

The new keyword performs this additional initialization automatically for you. It works with arrays with more than two dimensions as well:

```
float[][][] globalTemperatureData = new float[360][180][100];
```

When using new with multidimensional arrays, you do not have to specify a size for all dimensions of the array, only the leftmost dimension or dimensions. For example, the following two lines are legal:

```
float[][][] globalTemperatureData = new float[360][][];
float[][][] globalTemperatureData = new float[360][180][];
```

The first line creates a single-dimensional array, where each element of the array can hold a float[][]. The second line creates a two-dimensional array, where each element of the array is a float[]. If you specify a size for only some of the dimensions of an array, however, those dimensions must be the leftmost ones. The following lines are not legal:

```
float[][][] globalTemperatureData = new float[360][][100];  // Error!
float[][][] globalTemperatureData = new float[][180][100];  // Error!
```

Like a one-dimensional array, a multidimensional array can be initialized using an array literal. Simply use nested sets of curly braces to nest arrays within arrays. For example, we can declare, create, and initialize a 5×5 multiplication table like this:

```
int[][] products = { {0, 0, 0, 0, 0},
                     {0, 1, 2, 3, 4},
                     {0, 2, 4, 6, 8},
                     {0, 3, 6, 9, 12},
                     {0, 4, 8, 12, 16} };
```

Or, if you want to use a multidimensional array without declaring a variable, you can use the anonymous initializer syntax:

```
boolean response = bilingualQuestion(question, new String[][] {
                                                 { "Yes", "No" },
                                                 { "Oui", "Non" }});
```

When you create a multidimensional array using the new keyword, you always get a *rectangular* array: one in which all the array values for a given dimension have the same size. This is perfect for rectangular data structures, such as matrixes. However, because multidimensional arrays are implemented as arrays of arrays in Java, instead of as a single rectangular block of elements, you are in no way constrained to use rectangular arrays. For example, since our multiplication table is symmetrical about the diagonal from top left to bottom right, we can represent the same information in a nonrectangular array with fewer elements:

```
int[][] products = { {0},
                     {0, 1},
                     {0, 2, 4},
```

```
        {0, 3, 6, 9},
        {0, 4, 8, 12, 16} };
```

When working with multidimensional arrays, you'll often find yourself using nested loops to create or initialize them. For example, you can create and initialize a large triangular multiplication table as follows:

```
int[][] products = new int[12][];          // An array of 12 arrays of int.
for(int row = 0; row < 12; row++) {        // For each element of that array,
  products[row] = new int[row+1];          // allocate an array of int.
  for(int col = 0; col < row+1; col++)     // For each element of the int[],
    products[row][col] = row * col;        // initialize it to the product.
}
```

Reference Types

Now that we have discussed the syntax for working with objects and arrays, we can return to the issue of why classes and array types are known as reference types. As we saw in Table 2-2, all the Java primitive types have well-defined standard sizes, so all primitive values can be stored in a fixed amount of memory (between one and eight bytes, depending on the type). But classes and array types are composite types; objects and arrays contain other values, so they do not have a standard size, and they often require quite a bit more memory than eight bytes. For this reason, Java does not manipulate objects and arrays directly. Instead, it manipulates *references* to objects and arrays. Because Java handles objects and arrays by reference, classes and array types are known as reference types. In contrast, Java handles values of the primitive types directly, or by value.

A reference to an object or an array is simply some fixed-size value that refers to the object or array in some way.* When you assign an object or array to a variable, you are actually setting the variable to hold a reference to that object or array. Similarly, when you pass an object or array to a method, what really happens is that the method is given a reference to the object or array through which it can manipulate the object or array.

C and C++ programmers should note that Java does not support the & address-of operator or the * and -> dereference operators. In Java, primitive types are always handled exclusively by value, and objects and arrays are always handled exclusively by reference. Furthermore, unlike pointers in C and C++, references in Java are entirely opaque: they cannot be converted to or from integers, and they cannot be incremented or decremented.

Although references are an important part of how Java works, Java programs cannot manipulate references in any way. Despite this, there are significant differences between the behavior of primitive types and reference types in two important areas: the way values are copied and the way they are compared for equality.

---

* Typically, a reference is the memory address at which the object or array is stored. However, since Java references are opaque and cannot be manipulated in any way, this is an implementation detail

Copying Objects and Arrays

Consider the following code that manipulate a primitive `int` value:

```
int x = 42;
int y = x;
```

After these lines execute, the variable `y` contains a copy of the value held in the variable `x`. Inside the Java VM, there are two independent copies of the 32-bit integer 42.

Now think about what happens if we run the same basic code but use a reference type instead of a primitive type:

```
Point p = new Point(1.0, 2.0);
Point q = p;
```

After this code runs, the variable `q` holds a copy of the reference held in the variable `p`. There is still only one copy of the `Point` object in the VM, but there are now two copies of the reference to that object. This has some important implications. Suppose the two previous lines of code are followed by this code:

```
System.out.println(p.x);  // Print out the X coordinate of p: 1.0
q.x = 13.0;               // Now change the X coordinate of q
System.out.println(p.x);  // Print out p.x again; this time it is 13.0
```

Since the variables `p` and `q` hold references to the same object, either variable can be used to make changes to the object, and those changes are visible through the other variable as well.

This behavior is not specific to objects; the same thing happens with arrays, as illustrated by the following code:

```
char[] greet = { 'h','e','l','l','o' };  // greet holds an array reference
char[] cuss = greet;                     // cuss holds the same reference
cuss[4] = '!';                           // Use reference to change an element
System.out.println(greet);               // Prints "hell!"
```

A similar difference in behavior between primitive types and reference types occurs when arguments are passed to methods. Consider the following method:

```
void changePrimitive(int x) {
  while(x > 0)
    System.out.println(x--);
}
```

When this method is invoked, the method is given a copy of the argument used to invoke the method in the parameter `x`. The code in the method uses `x` as a loop counter and decrements it to zero. Since `x` is a primitive type, the method has its own private copy of this value, so this is a perfectly reasonable thing to do.

On the other hand, consider what happens if we modify the method so that the parameter is a reference type:

```
void changeReference(Point p) {
  while(p.x > 0)
    System.out.println(p.x--);
}
```

When this method is invoked, it is passed a private copy of a reference to a Point object and can use this reference to change the Point object. Consider the following:

```
Point q = new Point(3.0, 4.5);  // A point with an X coordinate of 3
changeReference(q);             // Prints 3,2,1 and modifies the Point
System.out.println(q.x);        // The X coordinate of q is now 0!
```

When the changeReference() method is invoked, it is passed a copy of the reference held in variable q. Now both the variable q and the method parameter p hold references to the same object. The method can use its reference to change the contents of the object. Note, however, that it cannot change the contents of the variable q. In other words, the method can change the Point object beyond recognition, but it cannot change the fact that the variable q refers to that object.

The title of this section is "Copying Objects and Arrays," but, so far, we've only seen copies of references to objects and arrays, not copies of the objects and arrays themselves. To make an actual copy of an object or an array, you must use the special clone() method (inherited by all objects from java.lang.Object):

```
Point p = new Point(1,2);    // p refers to one object
Point q = (Point) p.clone(); // q refers to a copy of that object
q.y = 42;                    // Modify the copied object, but not the original

int[] data = {1,2,3,4,5};           // An array
int[] copy = (int[]) data.clone();  // A copy of the array
```

Note that a cast is necessary to coerce the return value of the clone() method to the correct type. The reason for this will become clear later in this chapter. There are a couple of points you should be aware of when using clone(). First, not all objects can be cloned. Java only allows an object to be cloned if the object's class has explicitly declared itself to be cloneable by implementing the Cloneable interface. (We haven't discussed interfaces or how they are implemented yet; that is covered in Chapter 3.) The definition of Point that we showed earlier does not actually implement this interface, so our Point type, as implemented, is not cloneable. Note, however, that arrays are always cloneable. If you call the clone() method for a non-cloneable object, it throws a CloneNotSupportedException, so when you use the clone() method, you may want to use it within a try block to catch this exception.

The second thing you need to understand about clone() is that, by default, it is implemented to create a shallow copy of an object or array. The copied object or array contains copies of all the primitive values and references in the original object or array. In other words, any references in the object or array are copied, not cloned; clone() does not recursively make copies of the objects or arrays referred to by those references. A class may need to override this shallow copy behavior by defining its own version of the clone() method that explicitly performs a deeper copy where needed. To understand the shallow copy behavior of clone(), consider cloning a two-dimensional array of arrays:

```
int[][] data = {{1,2,3}, {4,5}};       // An array of 2 references
int[][] copy = (int[][]) data.clone(); // Copy the 2 refs to a new array
copy[0][0] = 99;                       // This changes data[0][0] too!
copy[1] = new int[] {7,8,9};           // This does not change data[1]
```

If you want to make a deep copy of this multidimensional array, you have to copy each dimension explicitly:

```
int[][] data = {{1,2,3}, {4,5}};       // An array of 2 references
int[][] copy = new int[data.length][]; // A new array to hold copied arrays
for(int i = 0; i < data.length; i++)
  copy[i] = (int[]) data[i].clone();
```

Comparing Objects and Arrays

We've seen that primitive types and reference types differ significantly in the way they are assigned to variables, passed to methods, and copied. The types also differ in the way they are compared for equality. When used with primitive values, the equality operator (==) simply tests whether two values are identical (i.e., whether they have exactly the same bits). With reference types, however, == compares references, not actual objects or arrays. In other words, == tests whether two references refer to the same object or array; it does not test whether two objects or arrays have the same content. For example:

```
String letter = "o";
String s = "hello";                       // These two String objects
String t = "hell" + letter;               // contain exactly the same text.
if (s == t) System.out.println("equal"); // But they are not equal!

byte[] a = { 1, 2, 3 };                   // An array.
byte[] b = (byte[]) a.clone();            // A copy with identical content.
if (a == b) System.out.println("equal"); // But they are not equal!
```

When working with reference types, there are two kinds of equality: equality of reference and equality of object. It is important to distinguish between these two kinds of equality. One way to do this is to use the word "equals" when talking about equality of references and the word "equivalent" when talking about two distinct object or arrays that have the same contents. Unfortunately, the designers of Java didn't use this nomenclature, as the method for testing whether one object is equivalent to another is named `equals()`. To test two objects for equivalence, pass one of them to the `equals()` method of the other:

```
String letter = "o";
String s = "hello";                       // These two String objects
String t = "hell" + letter;               // contain exactly the same text.
if (s.equals(t))                          // And the equals() method
  System.out.println("equivalent");      // tells us so.
```

All objects inherit an `equals()` method (from `Object`, but the default implementation simply uses == to test for equality of references, not equivalence of content. A class that wants to allow objects to be compared for equivalence can define its own version of the `equals()` method. Our `Point` class does not do this, but the `String` class does, as indicated by the code above. You can call the `equals()` method on an array, but it is the same as using the == operator, because arrays always inherit the default `equals()` method that compares references rather than array content. Starting in Java 1.2, you can compare arrays for equivalence with the convenience method `java.util.Arrays.equals()`. Prior to Java 1.2, however, you must loop through the elements of the arrays and compare them yourself.

The null Reference

We've seen the `null` keyword in our discussions of objects and arrays. Now that we have described references, it is worth revisiting `null` to point out that it is a special value that is a reference to nothing, or an absence of a reference. The default value for all reference types is `null`. The `null` value is unique in that it can be assigned to a variable of any reference type whatsoever.

Terminology: Pass by Value

I've said that Java handles arrays and objects "by reference." Don't confuse this with the phrase "pass by reference."* "Pass by reference" is a term used to describe the method-calling conventions of some programming languages. In a pass-by-reference language, values—even primitive values—are not passed directly to methods. Instead, methods are always passed references to values. Thus, if the method modifies its parameters, those modifications are visible when the method returns, even for primitive types.

Java does *not* do this; it is a "pass by value" language. However, when a reference type is involved, the value that is passed is a reference. But this is not the same as pass-by-reference. If Java were a pass-by-reference language, when a reference type was passed to a method, it would be passed as a reference to the reference.

Memory Allocation and Garbage Collection

As we've already noted, objects and arrays are composite values that can contain a number of other values and may require a substantial amount of memory. When you use the `new` keyword to create a new object or array or use an object or array literal in your program, Java automatically creates the object for you, allocating whatever amount of memory is necessary. You don't need to do anything to make this happen.

In addition, Java also automatically reclaims that memory for reuse when it is no longer needed. It does this through a process called *garbage collection.* An object is considered garbage when there are no longer any references to it stored in any variables, the fields of any objects, or the elements of any arrays. For example:

```
Point p = new Point(1,2);          // Create an object
double d = p.distanceFromOrigin(); // Use it for something
p = new Point(2,3);                // Create a new object
```

After the Java interpreter executes the third line, a reference to the new `Point` object has replaced the reference to the first one. There are now no remaining references to the first object, so it is garbage. At some point, the garbage collector will discover this and reclaim the memory used by the object.

C programmers, who are used to using `malloc()` and `free()` to manage memory, and C++ programmers, who are used to explicitly deleting their objects with `delete`, may find it a little hard to relinquish control and trust the garbage

* Unfortunately, previous editions of this book may have contributed to the confusion!

collector. Even though it seems like magic, it really works! There is a slight performance penalty due to the use of garbage collection, and Java programs may sometimes slow down noticeably while the garbage collector is actively reclaiming memory. However, having garbage collection built into the language dramatically reduces the occurrence of memory leaks and related bugs and almost always improves programmer productivity.

Reference Type Conversions

When we discussed primitive types earlier in this chapter, we saw that values of certain types can be converted to values of other types. Widening conversions are performed automatically by the Java interpreter, as necessary. Narrowing conversions, however, can result in lost data, so the interpreter does not perform them unless explicitly directed to do so with a cast.

Java does not allow any kind of conversion from primitive types to reference types or vice versa. Java does allow widening and narrowing conversions among certain reference types, however. As we've seen, there are an infinite number of potential reference types. In order to understand the conversions that can be performed among these types, you need to understand that the types form a hierarchy, usually called the *class hierarchy*.

Every Java class *extends* some other class, known as its *superclass*. A class inherits the fields and methods of its superclass and then defines its own additional fields and methods. There is a special class named `Object` that serves as the root of the class hierarchy in Java. It does not extend any class, but all other Java classes extend `Object` or some other class that has `Object` as one of its ancestors. The `Object` class defines a number of special methods that are inherited (or overridden) by all classes. These include the `toString()`, `clone()`, and `equals()` methods described earlier.

The predefined `String` class and the `Point` class we defined earlier in this chapter both extend `Object`. Thus, we can say that all `String` objects are also `Object` objects. We can also say that all `Point` objects are `Object` objects. The opposite is not true, however. We cannot say that every `Object` is a `String` because, as we've just seen, some `Object` objects are `Point` objects.

With this simple understanding of the class hierarchy, we can return to the rules of reference type conversion:

- An object cannot be converted to an unrelated type. The Java compiler does not allow you to convert a `String` to a `Point`, for example, even if you use a cast operator.
- An object can be converted to the type of a superclass. This is a widening conversion, so no cast is required. For example, a `String` value can be assigned to a variable of type `Object` or passed to a method where an `Object` parameter is expected. Note that no conversion is actually performed; the object is simply treated as if it were an instance of the superclass.

- An object can be converted to the type of a subclass, but this is a narrowing conversion and requires a cast. The Java compiler provisionally allows this kind of conversion, but the Java interpreter checks at runtime to make sure it is valid. Only cast an object to the type of a subclass if you are sure, based on the logic of your program, that the object is actually an instance of the subclass. If it is not, the interpreter throws a `ClassCastException`. For example, if we assign a `String` object to a variable of type `Object`, we can later cast the value of that variable back to type `String`:

```
Object o = "string";    // Widening conversion from String to Object
// Later in the program...
String s = (String) o;  // Narrowing conversion from Object to String
```

- All array types are distinct, so an array of one type cannot be converted to an array of another type, even if the individual elements could be converted. For example, although a `byte` can be widened to an `int`, a `byte[]` cannot be converted to an `int[]`, even with an explicit cast.

- Arrays do not have a type hierarchy, but all arrays are considered instances of `Object`, so any array can be converted to an `Object` value through a widening conversion. A narrowing conversion with a cast can convert such an object value back to an array. For example:

```
Object o = new int[] {1,2,3};  // Widening conversion from array to Object
// Later in the program...
int[] a = (int[]) o;           // Narrowing conversion back to array type
```

Packages and the Java Namespace

A *package* is a named collection of classes (and possibly subpackages). Packages serve to group related classes and define a namespace for the classes they contain.

The Java platform includes packages with names that begin with `java`, `javax`, and `org.omg`. (Sun also defines standard extensions to the Java platform in packages whose names begin with `javax`.) The most fundamental classes of the language are in the package `java.lang`. Various utility classes are in `java.util`. Classes for input and output are in `java.io`, and classes for networking are in `java.net`. Some of these packages contain subpackages. For example, `java.lang` contains two more specialized packages, named `java.lang.reflect` and `java.lang.ref`, and `java.util` contains a subpackage, `java.util.zip`, that contains classes for working with compressed ZIP archives.

Every class has both a simple name, which is the name given to it in its definition, and a fully qualified name, which includes the name of the package of which it is a part. The `String` class, for example, is part of the `java.lang` package, so its fully qualified name is `java.lang.String`.

Defining a Package

To specify the package a class is to be part of, you use a `package` directive. The `package` keyword, if it appears, must be the first token of Java code (i.e., the first thing other than comments and space) in the Java file. The keyword should be

followed by the name of the desired package and a semicolon. Consider a file of Java code that begins with this directive:

```
package com.davidflanagan.jude;
```

All classes defined by this file are part of the package named `com.davidflanagan.jude`.

If no `package` directive appears in a file of Java code, all classes defined in that file are part of a default unnamed package. As we'll see in Chapter 3, classes in the same package have special access to each other. Thus, except when you are writing simple example programs, you should always use the `package` directive to prevent access to your classes from totally unrelated classes that also just happen to be stored in the unnamed package.

Importing Classes and Packages

A class in a package `p` can refer to any other class in `p` by its simple name. And, since the classes in the `java.lang` package are so fundamental to the Java language, any Java code can refer to any class in this package by its simple name. Thus, you can always type `String`, instead of `java.lang.String`. By default, however, you must use the fully qualified name of all other classes. So, if you want to use the `File` class of the `java.io` package, you must type `java.io.File`.

Specifying package names explicitly all the time quickly gets tiring, so Java includes an `import` directive you can use to save some typing. `import` is used to specify classes and packages of classes that can be referred to by their simple names instead of by their fully qualified names. The `import` keyword can be used any number of times in a Java file, but all uses must be at the top of the file, immediately after the `package` directive, if there is one. There can be comments between the `package` directive and the `import` directives, of course, but there cannot be any other Java code.

The `import` directive is available in two forms. To specify a single class that can be referred to by its simple name, follow the `import` keyword with the name of the class and a semicolon:

```
import java.io.File;    // Now we can type File instead of java.io.File
```

To import an entire package of classes, follow `import` with the name of the package, the characters `.*`, and a semicolon. Thus, if you want to use several other classes from the `java.io` package in addition to the `File` class, you can simply import the entire package:

```
import java.io.*;   // Now we can use simple names for all classes in java.io
```

This package `import` syntax does not apply to subpackages. If I import the `java.util` package, I must still refer to the `java.util.zip.ZipInputStream` class by its fully qualified name. If two classes with the same name are both imported from different packages, neither one can be referred to by its simple name; to resolve this naming conflict unambiguously, you must use the fully qualified name of both classes.

Globally Unique Package Names

One of the important functions of packages is to partition the Java namespace and prevent name collisions between classes. It is only their package names that keep the `java.util.List` and `java.awt.List` classes distinct, for example. In order for this to work, however, package names must themselves be distinct. As the developer of Java, Sun controls all package names that begin with `java`, `javax`, and `sun`.

For the rest of us, Sun proposes a package-naming scheme, which, if followed correctly, guarantees globally unique package names. The scheme is to use your Internet domain name, with its elements reversed, as the prefix for all your package names. My web site is *davidflanagan.com*, so all my Java packages begin with `com.davidflanagan`. It is up to me to decide how to partition the namespace below `com.davidflanagan`, but since I own that domain name, no other person or organization who is playing by the rules can define a package with the same name as any of mine.

Java File Structure

This chapter has taken us from the smallest to the largest elements of Java syntax, from individual characters and tokens to operators, expressions, statements, and methods, and on up to classes and packages. From a practical standpoint, the unit of Java program structure you will be dealing with most often is the Java file. A Java file is the smallest unit of Java code that can be compiled by the Java compiler. A Java file consists of:

- An optional `package` directive
- Zero or more `import` directives
- One or more class definitions

These elements can be interspersed with comments, of course, but they must appear in this order. This is all there is to a Java file. All Java statements (except the `package` and `import` directives, which are not true statements) must appear within methods, and all methods must appear within a class definition.

There are a couple of other important restrictions on Java files. First, each file can contain at most one class that is declared `public`. A `public` class is one that is designed for use by other classes in other packages. We'll talk more about `public` and related modifiers in Chapter 3. This restriction on public classes only applies to top-level classes; a class can contain any number of nested or inner classes that are declared `public`, as we'll see in Chapter 3.

The second restriction concerns the filename of a Java file. If a Java file contains a `public` class, the name of the file must be the same as the name of the class, with the extension *.java* appended. Thus, if `Point` is defined as a `public` class, its source code must appear in a file named *Point.java*. Regardless of whether your classes are `public` or not, it is good programming practice to define only one per file and to give the file the same name as the class.

When a Java file is compiled, each of the classes it defines is compiled into a separate *class file* that contains Java byte codes to be interpreted by the Java Virtual Machine. A class file has the same name as the class it defines, with the extension *.class* appended. Thus, if the file *Point.java* defines a class named `Point`, a Java compiler compiles it to a file named *Point.class*. On most systems, class files are stored in directories that correspond to their package names. Thus, the class `com.davidflanagan.jude.DataFile` is defined by the class file *com/davidflanagan/jude/DataFile.class*.

The Java interpreter knows where the class files for the standard system classes are located and can load them as needed. When the interpreter runs a program that wants to use a class named `com.davidflanagan.jude.DataFile`, it knows that the code for that class is located in a directory named *com/davidflanagan/jude* and, by default, it "looks" in the current directory for a subdirectory of that name. In order to tell the interpreter to look in locations other than the current directory, you must use the `-classpath` option when invoking the interpreter or set the `CLASSPATH` environment variable. For details, see the documentation for the Java interpreter, *java*, in Chapter 8.

Defining and Running Java Programs

A Java program consists of a set of interacting class definitions. But not every Java class or Java file defines a program. To create a program, you must define a class that has a special method with the following signature:

```
public static void main(String args[])
```

This `main()` method is the main entry point for your program. It is where the Java interpreter starts running. This method is passed an array of strings and returns no value. When `main()` returns, the Java interpreter exits (unless `main()` has created separate threads, in which case the interpreter waits for all those threads to exit).

To run a Java program, you run the Java interpreter, *java*, specifying the fully qualified name of the class that contains the `main()` method. Note that you specify the name of the class, *not* the name of the class file that contains the class. Any additional arguments you specify on the command line are passed to the `main()` method as its `String[]` parameter. You may also need to specify the `-classpath` option to tell the interpreter to look for the classes needed by the program. Consider the following command:

```
C:\> java -classpath /usr/local/Jude com.davidflanagan.jude.Jude datafile.jude
```

java is the command to run the Java interpreter. *-classpath /usr/local/Jude* tells the interpreter where to look for *.class* files. `com.davidflanagan.jude.Jude` is the name of the program to run (i.e., the name of the class that defines the `main()` method). Finally, *datafile.jude* is a string that is passed to that `main()` method as the single element of an array of `String` objects.

In Java 1.2, there is an easier way to run programs. If a program and all its auxiliary classes (except those that are part of the Java platform) have been properly bundled in a Java archive (JAR) file, you can run the program simply by specifying the name of the JAR file:

```
C:\> java -jar /usr/local/Jude/jude.jar datafile.jude
```

Some operating systems make JAR files automatically executable. On those systems, you can simply say:

```
C:\> /usr/local/Jude/jude.jar datafile.jude
```

See Chapter 8 for details.

Differences Between C and Java

If you are a C or C++ programmer, you should have found much of the syntax of Java—particularly at the level of operators and statements—to be familiar. Because Java and C are so similar in some ways, it is important for C and C++ programmers to understand where the similarities end. There are a number of important differences between C and Java, which are summarized in the following list:

No preprocessor
: Java does not include a preprocessor and does not define any analogs of the `#define`, `#include`, and `#ifdef` directives. Constant definitions are replaced with `static final` fields in Java. (See the `java.lang.Math.PI` field for an example.) Macro definitions are not available in Java, but advanced compiler technology and inlining has made them less useful. Java does not require an `#include` directive because Java has no header files. Java class files contain both the class API and the class implementation, and the compiler reads API information from class files as necessary. Java lacks any form of conditional compilation, but its cross-platform portability means that this feature is very rarely needed.

No global variables
: Java defines a very clean namespace. Packages contain classes, classes contain fields and methods, and methods contain local variables. But there are no global variables in Java, and, thus, there is no possibility of namespace collisions among those variables.

Well-defined primitive type sizes
: All the primitive types in Java have well-defined sizes. In C, the size of `short`, `int`, and `long` types is platform-dependent, which hampers portability.

No pointers
: Java classes and arrays are reference types, and references to objects and arrays are akin to pointers in C. Unlike C pointers, however, references in Java are entirely opaque. There is no way to convert a reference to a primitive type, and a reference cannot be incremented or decremented. There is no address-of operator like &, dereference operator like * or ->, or `sizeof` operator. Pointers are a notorious source of bugs. Eliminating them simplifies the language and makes Java programs more robust and secure.

Garbage collection
: The Java Virtual Machine performs garbage collection so that Java programmers do not have to explicitly manage the memory used by all objects and arrays. This feature eliminates another entire category of common bugs and all but eliminates memory leaks from Java programs.

No goto statement

Java doesn't support a `goto` statement. Use of `goto` except in certain well-defined circumstances is regarded as poor programming practice. Java adds exception handling and labeled `break` and `continue` statements to the flow-control statements offered by C. These are a good substitute for `goto`.

Variable declarations anywhere

C requires local variable declarations to be made at the beginning of a method or block, while Java allows them anywhere in a method or block. Many programmers prefer to keep all their variable declarations grouped together at the top of a method, however.

Forward references

The Java compiler is smarter than the C compiler, in that it allows methods to be invoked before they are defined. This eliminates the need to declare functions in a header file before defining them in a program file, as is done in C.

Method overloading

Java programs can define multiple methods with the same name, as long as the methods have different parameter lists.

No struct and union types

Java doesn't support C `struct` and `union` types. A Java `class` can be thought of as an enhanced `struct`, however.

No enumerated types

Java doesn't support the `enum` keyword used in C to define types that consist of fixed sets of named values. This is surprising for a strongly typed language like Java, but there are ways to simulate this feature with object constants.

No bitfields

Java doesn't support the (infrequently used) ability of C to specify the number of individual bits occupied by fields of a `struct`.

No typedef

Java doesn't support the `typedef` keyword used in C to define aliases for type names. Java's lack of pointers makes its type-naming scheme simpler and more consistent than C's, however, so many of the common uses of `typedef` are not really necessary in Java.

No method pointers

C allows you to store the address of a function in a variable and pass this function pointer to other functions. You cannot do this with Java methods, but you can often achieve similar results by passing an object that implements a particular interface. Also, a Java method can be represented and invoked through a `java.lang.reflect.Method` object.

No variable-length argument lists

Java doesn't allow you to define methods such as C's `printf()` that take a variable number of arguments. Method overloading allows you to simulate C varargs functions for simple cases, but there's no general replacement for this feature.

CHAPTER 3

Object-Oriented Programming in Java

Java is an object-oriented programming language. As we discussed in Chapter 2, *Java Syntax from the Ground Up*, all Java programs use objects, and every Java program is defined as a class. The previous chapter explained the basic syntax of the Java programming language, including data types, operators, and expressions, and even showed how to define simple classes and work with objects. This chapter continues where that one left off, explaining the details of object-oriented programming in Java.

If you do not have any object-oriented (OO) programming background, don't worry; this chapter does not assume any prior experience. If you do have experience with OO programming, however, be careful. The term "object-oriented" has different meanings in different languages. Don't assume that Java works the same way as your favorite OO language. This is particularly true for C++ programmers. We saw in the last chapter that close analogies can be drawn between Java and C. The same is not true for Java and C++, however. Java uses object-oriented programming concepts that are familiar to C++ programmers and even borrows C++ syntax in a number of places, but the similarities between Java and C++ are not nearly as strong as those between Java and C. Don't let your experience with C++ lull you into a false familiarity with Java.

The Members of a Class

As we discussed in Chapter 2, a class is a collection of data, stored in named fields, and code, organized into named methods, that operates on that data. The fields and methods are called *members* of a class. In Java 1.1 and later, classes can also contain other classes. These member classes, or inner classes, are an advanced feature that is discussed later in the chapter. For now, we are going to discuss only fields and methods. The members of a class come in two distinct types: class, or static, members are associated with the class itself, while instance members are associated with individual instances of the class (i.e., with objects). Ignoring member classes for now, this gives us four types of members:

- Class fields
- Class methods
- Instance fields
- Instance methods

The simple class definition for the class `Circle`, shown in Example 3-1, contains all four types of members.

Example 3–1: A Simple Class and its Members

```
public class Circle {
  // A class field
  public static final double PI= 3.14159;     // A useful constant

  // A class method: just compute a value based on the arguments
  public static double radiansToDegrees(double rads) {
    return rads * 180 / PI;
  }

  // An instance field
  public double r;                  // The radius of the circle

  // Two instance methods: they operate on the instance fields of an object
  public double area() {            // Compute the area of the circle
    return PI * r * r;
  }
  public double circumference() {   // Compute the circumference of the circle
    return 2 * PI * r;
  }
}
```


Class Fields

A *class field* is associated with the class in which it is defined, rather than with an instance of the class. The following line declares a class field:

```
public static final double PI = 3.14159;
```

This line declares a field of type `double` named `PI` and assigns it a value of 3.14159. As you can see, a field declaration looks quite a bit like the local variable declarations we discussed in Chapter 2. The difference, of course, is that variables are defined within methods, while fields are members of classes.

The `static` modifier says that the field is a class field. Class fields are sometimes called static fields because of this `static` modifier. The `final` modifier says that the value of the field does not change. Since the field `PI` represents a constant, we declare it `final` so that it cannot be changed. It is a convention in Java (and many other languages) that constants are named with capital letters, which is why our field is named `PI`, not `pi`. Defining constants like this is a common use for class fields, meaning that the `static` and `final` modifiers are often used together. Not all class fields are constants, however. In other words, a field can be declared `static` without declaring it `final`. Finally, the `public` modifier says that anyone can use the field. This is a visibility modifier, and we'll discuss it and related modifiers in more detail later in this chapter.

The key point to understand about a static field is that there is only a single copy of it. This field is associated with the class itself, not with instances of the class. If you look at the various methods of the `Circle` class, you'll see that they use this field. From inside the `Circle` class, the field can be referred to simply as `PI`. Outside the class, however, both class and field names are required to uniquely specify the field. Methods that are not part of `Circle` access this field as `Circle.PI`.

A class field is essentially a global variable. The names of class fields are qualified by the unique names of the classes that contain them, however. Thus, Java does not suffer from the name collisions that can affect other languages when different modules of code define global variables with the same name.

Class Methods

As with class fields, *class methods* are declared with the `static` modifier:

```
public static double radiansToDegrees(double rads) { return rads * 180 / PI; }
```

This line declares a class method named `radiansToDegrees()`. It has a single parameter of type `double` and returns a `double` value. The body of the method is quite short; it performs a simple computation and returns the result.

Like class fields, class methods are associated with a class, rather than with an object. When invoking a class method from code that exists outside the class, you must specify both the name of the class and the method. For example:

```
// How many degrees is 2.0 radians?
double d = Circle.radiansToDegrees(2.0);
```

If you want to invoke a class method from inside the class in which it is defined, you don't have to specify the class name. However, it is often good style to specify the class name anyway, to make it clear that a class method is being invoked.

Note that the body of our `Circle.radiansToDegrees()` method uses the class field `PI`. A class method can use any class fields and class methods of its own class (or of any other class). But it cannot use any instance fields or instance methods because class methods are not associated with an instance of the class. In other words, although the `radiansToDegrees()` method is defined in the `Circle` class, it does not use any `Circle` objects. The instance fields and instance methods of the class are associated with `Circle` objects, not with the class itself. Since a class method is not associated with an instance of its class, it cannot use any instance methods or fields.

As we discussed earlier, a class field is essentially a global variable. In a similar way, a class method is a global method, or global function. Although `radiansToDegrees()` does not operate on `Circle` objects, it is defined within the `Circle` class because it is a utility method that is sometimes useful when working with circles. In many non-object-oriented programming languages, all methods, or functions, are global. You can write complex Java programs using only class methods. This is not object-oriented programming, however, and does not take advantage of the power of the Java language. To do true object-oriented programming, we need to add instance fields and instance methods to our repertoire.

Instance Fields

Any field declared without the `static` modifier is an *instance field*:

```
public double r;    // The radius of the circle
```

Instance fields are associated with instances of the class, rather than with the class itself. Thus, every `Circle` object we create has its own copy of the `double` field `r`. In our example, `r` represents the radius of a circle. Thus, each `Circle` object can have a radius independent of all other `Circle` objects.

Inside a class definition, instance fields are referred to by name alone. You can see an example of this if you look at the method body of the `circumference()` instance method. In code outside the class, the name of an instance method must be prepended by a reference to the object that contains it. For example, if we have a `Circle` object named `c`, we can refer to its instance field `r` as `c.r`:

```
Circle c = new Circle(); // Create a new Circle object; store it in variable c
c.r = 2.0;               // Assign a value to its instance field r
Circle d = new Circle(); // Create a different Circle object
d.r = c.r * 2;           // Make this one twice as big
```

Instance fields are key to object-oriented programming. Instance fields define an object; the values of those fields make one object distinct from another.

Instance Methods

Any method not declared with the `static` keyword is an instance method. An *instance method* operates on an instance of a class (an object) instead of operating on the class itself. It is with instance methods that object-oriented programming starts to get interesting. The `Circle` class defined in Example 3-1 contains two instance methods, `area()` and `circumference()`, that compute and return the area and circumference of the circle represented by a given `Circle` object.

To use an instance method from outside the class in which it is defined, we must prepend a reference to the instance that is to be operated on. For example:

```
Circle c = new Circle();   // Create a Circle object; store in variable c
c.r = 2.0;                 // Set an instance field of the object
double a = c.area();       // Invoke an instance method of the object
```

If you're new to object-oriented programming, that last line of code may look a little strange. I did not write:

```
a = area(c);
```

Instead, I wrote:

```
a = c.area();
```

This is why it is called object-oriented programming; the object is the focus here, not the function call. This small syntactic difference is perhaps the single most important feature of the object-oriented paradigm.

The point here is that we don't have to pass an argument to `c.area()`. The object we are operating on, `c`, is implicit in the syntax. Take a look at Example 3-1 again.

You'll notice the same thing in the signature of the `area()` method: it doesn't have a parameter. Now look at the body of the `area()` method: it uses the instance field `r`. Because the `area()` method is part of the same class that defines this instance field, the method can use the unqualified name `r`. It is understood that this refers to the radius of whatever `Circle` instance invokes the method.

Another important thing to notice about the bodies of the `area()` and `circumference()` methods is that they both use the class field `PI`. We saw earlier that class methods can use only class fields and class methods, not instance fields or methods. Instance methods are not restricted in this way: they can use any member of a class, whether it is declared `static` or not.

How instance methods work

Consider this line of code again:

```
a = c.area();
```

What's going on here? How can a method that has no parameters know what data to operate on? In fact, the `area()` method does have a parameter. All instance methods are implemented with an implicit parameter not shown in the method signature. The implicit argument is named `this`; it holds a reference to the object through which the method is invoked. In our example, that object is a `Circle`.

The implicit `this` parameter is not shown in method signatures because it is usually not needed; whenever a Java method accesses the fields in its class, it is implied that it is accessing fields in the object referred to by the `this` parameter. The same is true when an instance method invokes another instance method in the same class. I said earlier that to invoke an instance method you must prepend a reference to the object to be operated on. When an instance method is invoked within another instance method in the same class, however, you don't need to specify an object. In this case, it is implicit that the method is being invoked on the `this` object.

You can use the `this` keyword explicitly when you want to make it clear that a method is accessing its own fields and/or methods. For example, we can rewrite the `area()` method to use `this` explicitly to refer to instance fields:

```
public double area() {
  return Circle.PI * this.r * this.r;
}
```

This code also uses the class name explicitly to refer to class field `PI`. In a method this simple, it is not necessary to be explicit. In more complicated cases, however, you may find that it increases the clarity of your code to use an explicit `this` where it is not strictly required.

There are some cases in which the `this` keyword *is* required, however. For example, when a method parameter or local variable in a method has the same name as one of the fields of the class, you must use `this` to refer to the field, since the field name used alone refers to the method parameter or local variable. For example, we can add the following method to the `Circle` class:

```
public void setRadius(double r) {
  this.r = r;       // Assign the argument (r) to the field (this.r)
                    // Note that we cannot just say r = r
}
```

Finally, note that while instance methods can use the `this` keyword, class methods cannot. This is because class methods are not associated with objects.

Instance methods or class methods?

Instance methods are one of the key features of object-oriented programming. That doesn't mean, however, that you should shun class methods. There are many cases in which is is perfectly reasonable to define class methods. When working with the `Circle` class, for example, you might find there are many times you want to compute the area of a circle with a given radius, but don't want to bother creating a `Circle` object to represent that circle. In this case, a class method is more convenient:

```
public static double area(double r) { return PI * r * r; }
```


It is perfectly legal for a class to define more than one method with the same name, as long as the methods have different parameters. Since this version of the `area()` method is a class method, it does not have an implicit `this` parameter and must have a parameter that specifies the radius of the circle. This parameter keeps it distinct from the instance method of the same name.

As another example of the choice between instance methods and class methods, consider defining a method named `bigger()` that examines two `Circle` objects and returns whichever has the larger radius. We can write `bigger()` as an instance method as follows:

```
// Compare the implicit "this" circle to the "that" circle passed
// explicitly as an argument and return the bigger one.
public Circle bigger(Circle that) {
  if (this.r > that.r) return this;
  else return that;
}
```

We can also implement `bigger()` as a class method as follows:

```
// Compare circle a to circle b and return the one with the larger radius
public static Circle bigger(Circle a, Circle b) {
  if (a.r > b.r) return a;
  else return b;
}
```

Given two `Circle` objects, `x` and `y`, we can use either the instance method or the class method to determine which is bigger. The invocation syntax differs significantly for the two methods, however:

```
Circle biggest = x.bigger(y);          // Instance method: also y.bigger(x)
Circle biggest = Circle.bigger(x, y);  // Static method
```

Neither option is the correct choice. The instance method is more formally object-oriented, but its invocation syntax suffers from a kind of asymmetry. In a case like this, the choice between an instance method and a class method is simply a design

decision. Depending on the circumstances, one or the other will likely be the more natural choice.

A Mystery Solved

As we saw in Chapters 1 and 2, the way to display textual output to the terminal in Java is with a method named `System.out.println()`. Those chapters never explained why this method has such an long, awkward name or what those two periods are doing in it. Now that you understand class and instance fields and class and instance methods, it is easier to understand what is going on. Here's the story: `System` is a class. It has a class field named `out`. The field `System.out` refers to an object. The object `System.out` has an instance method named `println()`. Mystery solved! If you want to explore this in more detail, you can look up the `java.lang.System` class in Chapter 12, *The java.lang Package*. The class synopsis there tells you that the field `out` is of type `java.io.PrintStream`, which you can look up in Chapter 11, *The java.io Package*.

Creating and Initializing Objects

Take another look at how we've been creating `Circle` objects:

```
Circle c = new Circle();
```

What are those parentheses doing there? They make it look like we're calling a method. In fact, that is exactly what we're doing. Every class in Java has at least one *constructor*, which is a method that has the same name as the class and whose purpose is to perform any necessary initialization for a new object. Since we didn't explicitly define a constructor for our `Circle` class in Example 3-1, Java gave us a default constructor that takes no arguments and performs no special initialization.

Here's how a constructor works. The `new` operator creates a new, but uninitialized, instance of the class. The constructor method is then called, with the new object passed implicitly (a `this` reference, as we saw earlier), and whatever arguments that are specified between parentheses passed explicitly. The constructor can use these arguments to do whatever initialization is necessary.

Defining a Constructor

There is some obvious initialization we could do for our circle objects, so let's define a constructor. Example 3-2 shows a new definition for `Circle` that contains a constructor that lets us specify the radius of a new `Circle` object. The constructor also uses the `this` reference to distinguish between a method parameter and an instance field that have the same name.

Example 3-2: A Constructor for the Circle Class

```
public class Circle {
    public static final double PI = 3.14159;  // A constant
    public double r;   // An instance field that holds the radius of the circle
```

Example 3-2: A Constructor for the Circle Class (continued)

```
    // The constructor method: initialize the radius field
    public Circle(double r) { this.r = r; }

    // The instance methods: compute values based on the radius
    public double circumference() { return 2 * PI * r; }
    public double area() { return PI * r*r; }
}
```

When we relied on the default constructor supplied by the compiler, we had to write code like this to initialize the radius explicitly:

```
Circle c = new Circle();
c.r = 0.25;
```

With this new constructor, the initialization becomes part of the object creation step:

```
Circle c = new Circle(0.25);
```


Here are some important notes about naming, declaring, and writing constructors:

- The constructor name is always the same as the class name.
- Unlike all other methods, a constructor is declared without a return type, not even `void`.
- The body of a constructor should initialize the `this` object.
- A constructor should not return `this` or any other value.

Defining Multiple Constructors

Sometimes you want to initialize an object in a number of different ways, depending on what is most convenient in a particular circumstance. For example, we might want to initialize the radius of a circle to a specified value or a reasonable default value. Since our `Circle` class has only a single instance field, there aren't too many ways we can initialize it, of course. But in more complex classes, it is often convenient to define a variety of constructors. Here's how we can define two constructors for `Circle`:

```
public Circle() { r = 1.0; }
public Circle(double r) { this.r = r; }
```

It is perfectly legal to define multiple constructors for a class, as long as each constructor has a different parameter list. The compiler determines which constructor you wish based on the number and type of arguments you supply. This is simply an example of method overloading, which we discussed in Chapter 2.

Invoking One Constructor from Another

There is a specialized use of the `this` keyword that arises when a class has multiple constructors; it can be used from a constructor to invoke one of the other constructors of the same class. In other words, we can rewrite the two previous `Circle` constructors as follows:

```
// This is the basic constructor: initialize the radius
public Circle(double r) { this.r = r; }
// This constructor uses this() to invoke the constructor above
public Circle() { this(1.0); }
```

The `this()` syntax is a method invocation that calls one of the other constructors of the class. The particular constructor that is invoked is determined by the number and type of arguments, of course. This is a useful technique when a number of constructors share a significant amount of initialization code, as it avoids repetition of that code. This would be a more impressive example, of course, if the one-parameter version of the `Circle()` constructor did more initialization than it does.

There is an important restriction on using `this()`: it can appear only as the first statement in a constructor. It may, of course, be followed by any additional initialization a particular version of the constructor needs to do. The reason for this restriction involves the automatic invocation of superclass constructor methods, which we'll explore later in this chapter.

Field Defaults and Initializers

Not every field of a class requires initialization. Unlike local variables, which have no default value and cannot be used until explicitly initialized, the fields of a class are automatically initialized to the default values shown in Table 2-2. Essentially, every field of a primitive type is initialized to a default value of `false` or zero, as appropriate. All fields of reference type are, by default, initialized to `null`. These default values are guaranteed by Java. If the default value of a field is appropriate, you can simply rely on it without explicitly initializing the field. This default initialization applies to both instance fields and class fields.

As we've seen, the syntax for declaring a field of a class is a lot like the syntax for declaring a local variable. Both class and instance field declarations can be followed by an equals sign and an initial value, as in:

```
public static final double PI = 3.14159;
public double r = 1.0;
```

As we discussed in Chapter 2, a variable declaration is a statement that appears within a Java method; the variable initialization is performed when the statement is executed. Field declarations, however, are not part of any method, so they cannot be executed as statements are. Instead, the Java compiler generates instance-field initialization code automatically and puts it in the constructor or constructors for the class. The initialization code is inserted into a constructor in the order it appears in the source code, which means that a field initializer can use the initial values of fields declared before it. Consider the following code excerpt, which shows a constructor and two instance fields of a hypothetical class:

```
public class TestClass {
  ... public int len = 10;
  public int[] table = new int[len];

  public TestClass() {
    for(int i = 0; i < len; i++) table[i] = i;
  }
```

```
    // Rest of the class is omitted...
}
```

In this case, the code generated for the constructor is actually equivalent to the following:

```
public TestClass() {
  len = 10;
  table = new int[len];
  for(int i = 0; i < len; i++) table[i] = i;
}
```

If a constructor begins with a `this()` call to another constructor, the field initialization code does not appear in the first constructor. Instead, the initialization is handled in the constructor invoked by the `this()` call.

So, if instance fields are initialized in constructor methods, where are class fields initialized? These fields are associated with the class, even if no instances of the class are ever created, so they need to be initialized even before a constructor is called. To support this, the Java compiler generates a class initialization method automatically for every class. Class fields are initialized in the body of this method, which is guaranteed to be invoked exactly once before the class is first used (often when the class is first loaded). As with instance field initialization, class field initialization expressions are inserted into the class initialization method in the order they appear in the source code. This means that the initialization expression for a class field can use the class fields declared before it. The class initialization method is an internal method that is hidden from Java programmers. If you disassemble the byte codes in a Java class file, however, you'll see the class initialization code in a method named `<clinit>`.

Initializer blocks

So far, we've seen that objects can be initialized through the initialization expressions for their fields and by arbitrary code in their constructor methods. A class has a class initialization method, which is like a constructor, but we cannot explicitly define the body of this method as we can for a constructor. Java does allow us to write arbitrary code for the initialization of class fields, however, with a construct known as a *static initializer*. A static initializer is simply the keyword `static` followed by a block of code in curly braces. A static initializer can appear in a class definition anywhere a field or method definition can appear. For example, consider the following code that performs some nontrivial initialization for two class fields:

```
// We can draw the outline of a circle using trigonometric functions
// Trigonometry is slow, though, so we precompute a bunch of values
public class TrigCircle {
  // Here are our static lookup tables and their own simple initializers
  private static final NUMPTS = 500;
  private static double sines[] = new double[NUMPTS];
  private static double cosines[] = new double[NUMPTS];

  // Here's a static initializer that fills in the arrays
  static {
    double x = 0.0, delta_x;
    delta_x = (Circle.PI/2)/(NUMPTS-1);
```

```
    for(int i = 0, x = 0.0; i < NUMPTS; i++, x += delta_x) {
      sines[i] = Math.sin(x);
      cosines[i] = Math.cos(x);
    }
  }
  // The rest of the class is omitted... }
```

A class can have any number of static initializers. The body of each initializer block is incorporated into the class initialization method, along with any static field initialization expressions. A static initializer is like a class method in that it cannot use the `this` keyword or any instance fields or instance methods of the class.

In Java 1.1 and later, classes are also allowed to have instance initializers. An instance initializer is like a static initializer, except that it initializes an object, not a class. A class can have any number of instance initializers, and they can appear anywhere a field or method definition can appear. The body of each instance initializer is inserted at the beginning of every constructor for the class, along with any field initialization expressions. An instance initializer looks just like a static initializer, except that it doesn't use the `static` keyword. In other words, an instance initializer is just a block of arbitrary Java code that appears within curly braces.

Instance initializers can initialize arrays or other fields that require complex initialization. They are sometimes useful because they locate the initialization code right next to the field, instead of separating it off in a constructor method. For example:

```
private static finale int NUMPTS = 100;
private int[] data = new int[NUMPTS];
{ for(int i = 0; i < NUMPTS; i++) data[i] = i; }
```

In practice, however, this use of instance initializers is fairly rare. Instance initializers were introduced in Java to support anonymous inner classes, and that is their main utility (we'll discuss anonymous inner classes later in this chapter).

Destroying and Finalizing Objects

Now that we've seen how new objects are created and initialized in Java, we need to study the other end of the object life cycle and examine how objects are finalized and destroyed. *Finalization* is the opposite of initialization.

As I mentioned in Chapter 2, the memory occupied by an object is automatically reclaimed when the object is no longer needed. This is done through a process known as *garbage collection.* Garbage collection is not some newfangled technique; it has been around for years in languages such as Lisp. It just takes some getting used to for programmers accustomed to such languages as C and C++, in which you must call the `free()` method or the `delete` operator to reclaim memory. The fact that you don't need to remember to destroy every object you create is one of the features that makes Java a pleasant language to work with. It is also one of the features that makes programs written in Java less prone to bugs than those written in languages that don't support automatic garbage collection.

Garbage Collection

The Java interpreter knows exactly what objects and arrays it has allocated. It can also figure out which local variables refer to which objects and arrays, and which objects and arrays refer to which other objects and arrays. Thus, the interpreter is able to determine when an allocated object is no longer referred to by any other object or variable. When the interpreter finds such an object, it knows it can destroy the object safely and does so. The garbage collector can also detect and destroy cycles of objects that refer to each other, but are not referenced by any other active objects. Any such cycles are also reclaimed.

The Java garbage collector runs as a low-priority thread, so it does most of its work when nothing else is going on, such as during idle time while waiting for user input. The only time the garbage collector must run while something high-priority is going on (i.e., the only time it will actually slow down the system) is when available memory has become dangerously low. This doesn't happen very often because the low-priority thread cleans things up in the background.

This scheme may sound slow and wasteful of memory. Actually though, modern garbage collectors can be surprisingly efficient. Garbage collection will never be as efficient as well-written, explicit memory allocation and deallocation. But it does make programming a lot easier and less prone to bugs. And for most real-world programs, rapid development, lack of bugs, and easy maintenance are more important features than raw speed or memory efficiency.

Memory Leaks in Java

The fact that Java supports garbage collection dramatically reduces the incidence of a class of bugs known as *memory leaks*. A memory leak occurs when memory is allocated and never reclaimed. At first glance, it might seem that garbage collection prevents all memory leaks because it reclaims all unused objects. A memory leak can still occur in Java, however, if a valid (but unused) reference to an unused object is left hanging around. For example, when a method runs for a long time (or forever), the local variables in that method can retain object references much longer than they are actually required. The following code illustrates:

```
public static void main(String argso[]) {
  int big_array[] = new int[100000];

  // Do some computations with big_array and get a result.
  int result = compute(big_array);

  // We no longer need big_array. It will get garbage collected when there
  // are no more references to it. Since big_array is a local variable,
  // it refers to the array until this method returns. But this method
  // doesn't return. So we've got to explicitly get rid of the reference
  // ourselves, so the garbage collector knows it can reclaim the array.
  big_array = null;

  // Loop forever, handling the user's input
  for(;;) handle_input(result);
}
```

Memory leaks can also occur when you use a hashtable or similar data structure to associate one object with another. Even when neither object is required anymore, the association remains in the hashtable, preventing the objects from being reclaimed until the hashtable itself is reclaimed. If the hashtable has a substantially longer lifetime than the objects it holds, this can cause memory leaks.

Object Finalization

A *finalizer* in Java is the opposite of a constructor. While a constructor method performs initialization for an object, a finalizer method performs finalization for the object. Garbage collection automatically frees up the memory resources used by objects, but objects can hold other kinds of resources, such as open files and network connections. The garbage collector cannot free these resources for you, so you need to write a finalizer method for any object that needs to perform such tasks as closing files, terminating network connections, deleting temporary files, and so on.

A finalizer is an instance method that takes no arguments and returns no value. There can be only one finalizer per class, and it must be named `finalize()`.* A finalizer can throw any kind of exception or error, but when a finalizer is automatically invoked by the garbage collector, any exception or error it throws is ignored and serves only to cause the finalizer method to return. Finalizer methods are typically declared `protected` (which we have not discussed yet), but can also be declared `public`. An example finalizer looks like this:

```
protected void finalize() throws Throwable {
  // Invoke the finalizer of our superclass
  // We haven't discussed superclasses or this syntax yet
  super.finalize();

  // Delete a temporary file we were using
  // If the file doesn't exist or tempfile is null, this can throw
  // an exception, but that exception is ignored.
  tempfile.delete();
}
```

Here are some important points about finalizers:

- If an object has a finalizer, the finalizer method is invoked sometime after the object becomes unused (or unreachable), but before the garbage collector reclaims the object.
- Java makes no guarantees about when garbage collection will occur or in what order objects will be collected. Therefore, Java can make no guarantees about when (or even whether) a finalizer will be invoked, in what order finalizers will be invoked, or what thread will execute finalizers.

* C++ programmers should note that although Java constructor methods are named like C++ constructors, Java finalization methods are not named like C++ destructor methods. As we will see, they do not behave quite like C++ destructor methods, either.

- The Java interpreter can exit without garbage collecting all outstanding objects, so some finalizers may never be invoked. In this case, though, any outstanding resources are usually freed by the operating system. In Java 1.1, the `Runtime` method `runFinalizersOnExit()` can force the virtual machine to run finalizers before exiting. Unfortunately, however, this method can cause deadlock and is inherently unsafe; it has been deprecated as of Java 1.2. In Java 1.3, the `Runtime` method `addShutdownHook()` can safely execute arbitrary code before the Java interpreter exits.
- After a finalizer is invoked, objects are not freed right away. This is because a finalizer method can resurrect an object by storing the `this` pointer somewhere so that the object once again has references. Thus, after `finalize()` is called, the garbage collector must once again determine that the object is unreferenced before it can garbage-collect it. However, even if an object is resurrected, the finalizer method is never invoked more than once. Resurrecting an object is never a useful thing to do—just a strange quirk of object finalization. As of Java 1.2, the `java.lang.ref.PhantomReference` class can implement an alternative to finalization that does not allow resurrection.

In practice, it is relatively rare for an application-level class to require a `finalize()` method. Finalizer methods are more useful, however, when writing Java classes that interface to native platform code with `native` methods. In this case, the native implementation can allocate memory or other resources that are not under the control of the Java garbage collector and need to be reclaimed explicitly by a `native finalize()` method.

While Java supports both class and instance initialization through static initializers and constructors, it provides only a facility for instance finalization. The original Java specification called for a `classFinalize()` method that could finalize a class when the class itself became unused and was unloaded from the VM. This feature was never implemented, however, and because it has proved to be unnecessary, class finalization has been removed from the language specification.

Subclasses and Inheritance

The `Circle` defined earlier is a simple class that distinguishes circle objects only by their radii. Suppose, instead, that we want to represent circles that have both a size and a position. For example, a circle of radius 1.0 centered at point 0,0 in the Cartesian plane is different from the circle of radius 1.0 centered at point 1,2. To do this, we need a new class, which we'll call `PlaneCircle`. We'd like to add the ability to represent the position of a circle without losing any of the existing functionality of the `Circle` class. This is done by defining `PlaneCircle` as a subclass of `Circle`, so that `PlaneCircle` inherits the fields and methods of its superclass, `Circle`. The ability to add functionality to a class by subclassing, or extending, it is central to the object-oriented programming paradigm.

Extending a Class

Example 3-3 shows how we can implement `PlaneCircle` as a subclass of the `Circle` class.

Example 3-3: Extending the Circle Class

```
public class PlaneCircle extends Circle {
  // We automatically inherit the fields and methods of Circle,
  // so we only have to put the new stuff here.
  // New instance fields that store the center point of the circle
  public double cx, cy;

  // A new constructor method to initialize the new fields
  // It uses a special syntax to invoke the Circle() constructor
  public PlaneCircle(double r, double x, double y) {
    super(r);       // Invoke the constructor of the superclass, Circle()
    this.cx = x;    // Initialize the instance field cx
    this.cy = y;    // Initialize the instance field cy
  }

  // The area() and circumference() methods are inherited from Circle
  // A new instance method that checks whether a point is inside the circle
  // Note that it uses the inherited instance field r
  public boolean isInside(double x, double y) {
    double dx = x - cx, dy = y - cy;            // Distance from center
    double distance = Math.sqrt(dx*dx + dy*dy); // Pythagorean theorem
    return (distance < r);                      // Returns true or false
  }
}
```

Note the use of the keyword `extends` in the first line of Example 3-3. This keyword tells Java that `PlaneCircle` extends, or subclasses, `Circle`, meaning that it inherits the fields and methods of that class.* The definition of the `isInside()` method shows field inheritance; this method uses the field `r` (defined by the `Circle` class) as if it were defined right in `PlaneCircle` itself. `PlaneCircle` also inherits the methods of `Circle`. Thus, if we have a `PlaneCircle` object referenced by variable `pc`, we can say:

```
double ratio = pc.circumference() / pc.area();
```

This works just as if the `area()` and `circumference()` methods were defined in `PlaneCircle` itself.

Another feature of subclassing is that every `PlaneCircle` object is also a perfectly legal `Circle` object. Thus, if `pc` refers to a `PlaneCircle` object, we can assign it to a `Circle` variable and forget all about its extra positioning capabilities:

```
PlaneCircle pc = new PlaneCircle(1.0, 0.0, 0.0);  // Unit circle at the origin
Circle c = pc;     // Assigned to a Circle variable without casting
```

This assignment of a `PlaneCircle` object to a `Circle` variable can be done without a cast. As we discussed in Chapter 2, this is a widening conversion and is always legal. The value held in the `Circle` variable `c` is still a valid `PlaneCircle` object,

* C++ programmers should note that `extends` is the Java equivalent of : in C++; both are used to indicate the superclass of a class.

but the compiler cannot know this for sure, so it doesn't allow us to do the opposite (narrowing) conversion without a cast:

```
// Narrowing conversions require a cast (and a runtime check by the VM)
PlaneCircle pc2 = (PlaneCircle) c;
boolean origininside = ((PlaneCircle) c).isInside(0.0, 0.0);
```

Final classes

When a class is declared with the `final` modifier, it means that it cannot be extended or subclassed. `java.lang.System` is an example of a `final` class. Declaring a class `final` prevents unwanted extensions to the class, and it also allows the compiler to make some optimizations when invoking the methods of a class. We'll explore this in more detail later in this chapter, when we talk about method overriding.

Superclasses, Object, and the Class Hierarchy

In our example, `PlaneCircle` is a subclass of `Circle`. We can also say that `Circle` is the superclass of `PlaneCircle`. The superclass of a class is specified in its `extends` clause:

```
public class PlaneCircle extends Circle { ... }
```

Every class you define has a superclass. If you do not specify the superclass with an `extends` clause, the superclass is the class `java.lang.Object`. `Object` is a special class for a couple of reasons:

- It is the only class in Java that does not have a superclass.
- All Java classes inherit the methods of `Object`.

Because every class has a superclass, classes in Java form a class hierarchy, which can be represented as a tree with `Object` at its root. Figure 3-1 shows a class hierarchy diagram that includes our `Circle` and `PlaneCircle` classes, as well as some of the standard classes from the Java API. Every API quick-reference chapter in Part II includes a class-hierarchy diagram for the classes it documents.

Subclass Constructors

Look again at the `PlaneCircle()` constructor method of Example 3-3:

```
public PlaneCircle(double r, double x, double y) {
  super(r);       // Invoke the constructor of the superclass, Circle()
  this.cx = x;    // Initialize the instance field cx
  this.cy = y;    // Initialize the instance field cy
}
```

This constructor explicitly initializes the `cx` and `cy` fields newly defined by `PlaneCircle`, but it relies on the superclass `Circle()` constructor to initialize the inherited fields of the class. To invoke the superclass constructor, our constructor calls `super()`. `super` is a reserved word in Java. One of its uses is to invoke the constructor method of a superclass from within the constructor method of a subclass. This use is analogous to the use of `this()` to invoke one constructor method

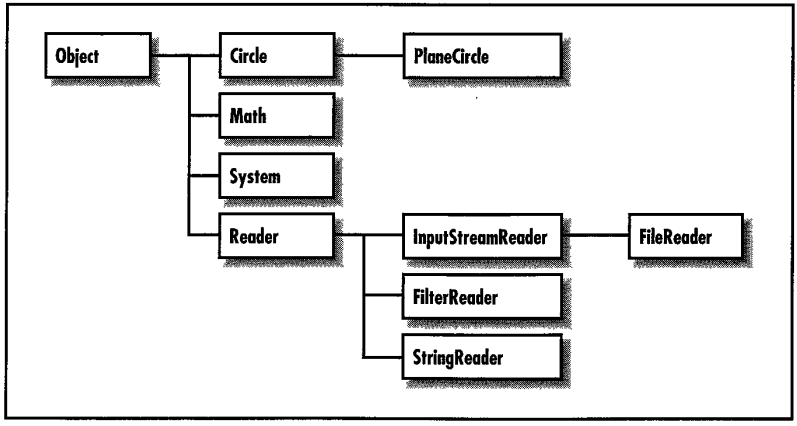


Figure 3–1: A class hierarchy diagram

of a class from within another constructor method of the same class. Using `super()` to invoke a constructor is subject to the same restrictions as using `this()` to invoke a constructor:

- `super()` can be used in this way only within a constructor method.
- The call to the superclass constructor must appear as the first statement within the constructor method, even before local variable declarations.

The arguments passed to `super()` must match the parameters of the superclass constructor. If the superclass defines more than one constructor, `super()` can be used to invoke any one of them, depending on the arguments passed.

Constructor Chaining and the Default Constructor

Java guarantees that the constructor method of a class is called whenever an instance of that class is created. It also guarantees that the constructor is called whenever an instance of any subclass is created. In order to guarantee this second point, Java must ensure that every constructor method calls its superclass constructor method. Thus, if the first statement in a constructor does not explicitly invoke another constructor with `this()` or `super()`, Java implicitly inserts the call `super()`; that is, it calls the superclass constructor with no arguments. If the superclass does not have a constructor that takes no arguments, this implicit invocation causes a compilation error.

Consider what happens when we create a new instance of the `PlaneCircle` class. First, the `PlaneCircle` constructor is invoked. This constructor explicitly calls `super(r)` to invoke a `Circle` constructor, and that `Circle()` constructor implicitly calls `super()` to invoke the constructor of its superclass, `Object`. The body of the `Object` constructor runs first. When it returns, the body of the `Circle()` constructor runs. Finally, when the call to `super(r)` returns, the remaining statements of the `PlaneCircle()` constructor are executed.

What all this means is that constructor calls are chained; any time an object is created, a sequence of constructor methods is invoked, from subclass to superclass on up to `Object` at the root of the class hierarchy. Because a superclass constructor is always invoked as the first statement of its subclass constructor, the body of the `Object` constructor always runs first, followed by the constructor of its subclass and on down the class hierarchy to the class that is being instantiated. There is an important implication here; when a constructor is invoked, it can count on the fields of its superclass to be initialized.

The default constructor

There is one missing piece in the previous description of constructor chaining. If a constructor does not invoke a superclass constructor, Java does so implicitly. But what if a class is declared without a constructor? In this case, Java implicitly adds a constructor to the class. This default constructor does nothing but invoke the superclass constructor. For example, if we don't declare a constructor for the `PlaneCircle` class, Java implicitly inserts this constructor:

```
public PlaneCircle() { super(); }
```

If the superclass, `Circle`, doesn't declare a no-argument constructor, the `super()` call in this automatically inserted default constructor for `PlaneCircle()` causes a compilation error. In general, if a class does not define a no-argument constructor, all its subclasses must define constructors that explicitly invoke the superclass constructor with the necessary arguments.

If a class does not declare any constructors, it is given a no-argument constructor by default. Classes declared `public` are given `public` constructors. All other classes are given a default constructor that is declared without any visibility modifier: such a constructor has default visibility. (The notion of visibility is explained later in this chapter.) If you are creating a `public` class that should not be publicly instantiated, you should declare at least one non-`public` constructor to prevent the insertion of a default `public` constructor. Classes that should never be instantiated (such as `java.lang.Math` or `java.lang.System`) should define a `private` constructor. Such a constructor can never be invoked from outside of the class, but it prevents the automatic insertion of the default constructor.

Finalizer chaining?

You might assume that, since Java chains constructor methods, it also automatically chains the finalizer methods for an object. In other words, you might assume that the finalizer method of a class automatically invokes the finalizer of its superclass, and so on. In fact, Java does *not* do this. When you write a `finalize()` method, you must explicitly invoke the superclass finalizer. (You should do this even if you know that the superclass does not have a finalizer because a future implementation of the superclass might add a finalizer.)

As we saw in our example finalizer earlier in the chapter, you can invoke a superclass method with a special syntax that uses the `super` keyword:

```
// Invoke the finalizer of our superclass. super.finalize();
```

We'll discuss this syntax in more detail when we consider method overriding. In practice, the need for finalizer methods, and thus finalizer chaining, rarely arises.

Shadowing Superclass Fields

For the sake of example, imagine that our `PlaneCircle` class needs to know the distance between the center of the circle and the origin (0,0). We can add another instance field to hold this value:

```
public double r;
```

Adding the following line to the constructor computes the value of the field:

```
this.r = Math.sqrt(cx*cx + cy*cy);  // Pythagorean Theorem
```

But wait, this new field `r` has the same name as the radius field `r` in the `Circle` superclass. When this happens, we say that the field `r` of `PlaneCircle` *shadows* the field `r` of `Circle`. (This is a contrived example, of course: the new field should really be called `distanceFromOrigin`. Although you should attempt to avoid it, subclass fields do sometimes shadow fields of their superclass.)

With this new definition of `PlaneCircle`, the expressions `r` and `this.r` both refer to the field of `PlaneCircle`. How, then, can we refer to the field `r` of `Circle` that holds the radius of the circle? There is a special syntax for this that uses the `super` keyword:

```
r       // Refers to the PlaneCircle field
this.r  // Refers to the PlaneCircle field
super.r // Refers to the Circle field
```

Another way to refer to a shadowed field is to cast `this` (or any instance of the class) to the appropriate superclass and then access the field:

```
((Circle) this).r   // Refers to field r of the Circle class
```

This casting technique is particularly useful when you need to refer to a shadowed field defined in a class that is not the immediate superclass. Suppose, for example, that classes `A`, `B`, and `C` all define a field named `x` and that `C` is a subclass of `B`, which is a subclass of `A`. Then, in the methods of class `C`, you can refer to these different fields as follows:

```
x               // Field x in class C
this.x          // Field x in class C
super.x         // Field x in class B
((B)this).x     // Field x in class B
((A)this).x     // Field x in class A
super.super.x   // Illegal; does not refer to x in class A
```

You cannot refer to a shadowed field `x` in the superclass of a superclass with `super.super.x`. This is not legal syntax.

Similarly, if you have an instance `c` of class `C`, you can refer to the three fields named `x` like this:

```
c.x             // Field x of class C
((B)c).x        // Field x of class B
((A)c).x        // Field x of class A
```

So far, we've been discussing instance fields. Class fields can also be shadowed. You can use the same `super` syntax to refer to the shadowed value of the field, but this is never necessary since you can always refer to a class field by prepending the name of the desired class. Suppose that the implementer of `PlaneCircle` decides that the `Circle.PI` field does not express π to enough decimal places. She can define her own class field `PI`:

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

Now, code in `PlaneCircle` can use this more accurate value with the expressions `PI` or `PlaneCircle.PI`. It can also refer to the old, less accurate value with the expressions `super.PI` and `Circle.PI`. Note, however, that the `area()` and `circumference()` methods inherited by `PlaneCircle` are defined in the `Circle` class, so they use the value `Circle.PI`, even though that value is shadowed now by `PlaneCircle.PI`.

Overriding Superclass Methods

When a class defines an instance method using the same name, return type, and parameters as a method in its superclass, that method *overrides* the method of the superclass. When the method is invoked for an object of the class, it is the new definition of the method that is called, not the superclass's old definition.

Method overriding is an important and useful technique in object-oriented programming. `PlaneCircle` does not override either of the methods defined by `Circle`, but suppose we define another subclass of `Circle`, named `Ellipse`.* In this case, it is important for `Ellipse` to override the `area()` and `circumference()` methods of `Circle`, since the formulas used to compute the area and circumference of a circle do not work for ellipses.

The upcoming discussion of method overriding considers only instance methods. Class methods behave quite differently, and there isn't much to say. Like fields, class methods can be shadowed by a subclass, but not overridden. As I noted earlier in this chapter, it is good programming style to always prefix a class method invocation with the name of the class in which it is defined. If you consider the class name part of the class method name, the two methods have different names, so nothing is actually shadowed at all. It is, however, illegal for a class method to shadow an instance method.

Before we go any further with the discussion of method overriding, you need to be sure you understand the difference between method overriding and method overloading. As we discussed in Chapter 2, method overloading refers to the practice of defining multiple methods (in the same class) that have the same name, but different parameter lists. This is very different from method overriding, so don't get them confused.

---

* Mathematical purists may argue that since all circles are ellipses, `Ellipse` should be the superclass and `Circle` the subclass. A pragmatic engineer might counterargue that circles can be represented with fewer instance fields, so `Circle` objects should not be burdened by inheriting unnecessary fields from `Ellipse`. In any case, this is a useful example here.

Overriding is not shadowing

Although Java treats the fields and methods of a class analogously in many ways, method overriding is not like field shadowing at all. You can refer to shadowed fields simply by casting an object to an instance of the appropriate superclass, but you cannot invoke overridden instance methods with this technique. The following code illustrates this crucial difference:

```
class A {                           // Define a class named A
  int i = 1;                        // An instance field
  int f() { return i; }             // An instance method
  static char g() { return 'A'; }   // A class method
}

class B extends A {                 // Define a subclass of A
  int i = 2;                        // Shadows field i in class A
  int f() { return -i; }            // Overrides instance method f in class A
  static char g() { return 'B'; }   // Shadows class method g() in class A
}

public class OverrideTest {
  public static void main(String args[]) {
    B b = new B();                // Creates a new object of type B
    System.out.println(b.i);      // Refers to B.i; prints 2
    System.out.println(b.f());    // Refers to B.f(); prints -2
    System.out.println(b.g());    // Refers to B.g(); prints B
    System.out.println(B.g());    // This is a better way to invoke B.g()

    A a = (A) b;                  // Casts b to an instance of class A
    System.out.println(a.i);      // Now refers to A.i; prints 1
    System.out.println(a.f());    // Still refers to B.f(); prints -2
    System.out.println(a.g());    // Refers to A.g(); prints A
    System.out.println(A.g());    // This is a better way to invoke A.g()
  }
}
```

While this difference between method overriding and field shadowing may seem surprising at first, a little thought makes the purpose clear. Suppose we have a bunch of `Circle` and `Ellipse` objects we are manipulating. To keep track of the circles and ellipses, we store them in an array of type `Circle[]`. (We can do this because `Ellipse` is a subclass of `Circle`, so all `Ellipse` objects are legal `Circle` objects.) When we loop through the elements of this array, we don't have to know or care whether the element is actually a `Circle` or an `Ellipse`. What we do care about very much, however, is that the correct value is computed when we invoke the `area()` method of any element of the array. In other words, we don't want to use the formula for the area of a circle when the object is actually an ellipse! Seen in this context, it is not surprising at all that method overriding is handled differently by Java than field shadowing.

Dynamic method lookup

If we have a `Circle[]` array that holds `Circle` and `Ellipse` objects, how does the compiler know whether to call the `area()` method of the `Circle` class or the `Ellipse` class for any given item in the array? In fact, the compiler does not know this because it cannot know it. The compiler knows that it does not know,

however, and produces code that uses dynamic method lookup at runtime. When the interpreter runs the code, it looks up the appropriate `area()` method to call for each of the objects in the array. That is, when the interpreter interprets the expression `o.area()`, it checks the actual type of the object referred to by the variable `o` and then finds the `area()` method that is appropriate for that type. It does not simply use the `area()` method that is statically associated with the type of the variable `o`. This process of dynamic method lookup is sometimes also called virtual method invocation.*

Final methods and static method lookup

Virtual method invocation is fast, but method invocation is faster when no dynamic lookup is necessary at runtime. Fortunately, there are a number of situations in which Java does not need to use dynamic method lookup. In particular, if a method is declared with the `final` modifier, it means that the method definition is the final one; it cannot be overridden by any subclasses. If a method cannot be overridden, the compiler knows that there is only one version of the method, and dynamic method lookup is not necessary.† In addition, all methods of a `final` class are themselves implicitly final and cannot be overridden. As we'll discuss later in this chapter, `private` methods are not inherited by subclasses and, therefore, cannot be overridden (i.e., all `private` methods are implicitly `final`). Finally, class methods behave like fields (i.e., they can be shadowed by subclasses but not overridden). Taken together, this means that all methods of a class that is declared `final`, as well as all methods that are `final`, `private`, or `static`, are invoked without dynamic method lookup. These methods are also candidates for inlining at runtime by a just-in-time compiler (JIT) or similar optimization tool.

Invoking an overridden method

We've seen the important differences between method overriding and field shadowing. Nevertheless, the Java syntax for invoking an overridden method is quite similar to the syntax for accessing a shadowed field: both use the `super` keyword. The following code illustrates:

```
class A {
  int i = 1;                // An instance field shadowed by subclass B
  int f() { return i; }     // An instance method overridden by subclass B
}
class B extends A {
  int i;                    // This field shadows i in A
  int f() {                 // This method overrides f() in A
    i = super.i + 1;        // It can retrieve A.i like this
    return super.f() + i;   // It can invoke A.f() like this
  }
}
```

* C++ programmers should note that dynamic method lookup is what C++ does for `virtual` functions. An important difference between Java and C++ is that Java does not have a `virtual` keyword. In Java, methods are virtual by default.

† In this sense, the `final` modifier is the opposite of the `virtual` modifier in C++. All non-`final` methods in Java are `virtual`.

Recall that when you use `super` to refer to a shadowed field, it is the same as casting `this` to the superclass type and accessing the field through that. Using `super` to invoke an overridden method, however, is not the same as casting `this`. In other words, in the previous code, the expression `super.f()` is not the same as `((A)this).f()`.

When the interpreter invokes an instance method with this `super` syntax, a modified form of dynamic method lookup is performed. The first step, as in regular dynamic method lookup, is to determine the actual class of the object through which the method is invoked. Normally, the dynamic search for an appropriate method definition would begin with this class. When a method is invoked with the `super` syntax, however, the search begins at the superclass of the class. If the superclass implements the method directly, that version of the method is invoked. If the superclass inherits the method, the inherited version of the method is invoked.

Note that the `super` keyword invokes the most immediately overridden version of a method. Suppose class `A` has a subclass `B` that has a subclass `C`, and all three classes define the same method `f()`. Then the method `C.f()` can invoke the method `B.f()`, which it overrides directly, with `super.f()`. But there is no way for `C.f()` to invoke `A.f()` directly: `super.super.f()` is not legal Java syntax. Of course, if `C.f()` invokes `B.f()`, it is reasonable to suppose that `B.f()` might also invoke `A.f()`. This kind of chaining is relatively common when working with overridden methods: it is a way of augmenting the behavior of a method without replacing the method entirely. We saw this technique in the the example `finalize()` method shown earlier in the chapter: that method invoked `super.finalize()` to run its superclass finalization method.

Don't confuse the use of `super` to invoke an overridden method with the `super()` method call used in constructor methods to invoke a superclass constructor. Although they both use the same keyword, these are two entirely different syntaxes. In particular, you can use `super` to invoke an overridden method anywhere in the overriding method, while you can use `super()` only to invoke a superclass constructor as the very first statement of a constructor.

It is also important to remember that `super` can be used only to invoke an overridden method from within the method that overrides it. Given an `Ellipse` object `e`, there is no way for a program that uses an object (with or without the `super` syntax) to invoke the `area()` method defined by the `Circle` class on this object.

I've already explained that class methods can shadow class methods in superclasses, but cannot override them. The preferred way to invoke class methods is to include the name of the class in the invocation. If you do not do this, however, you can use the `super` syntax to invoke a shadowed class method, just as you would invoke an instance method or refer to a shadowed field.

Data Hiding and Encapsulation

We started this chapter by describing a class as "a collection of data and methods." One of the important object-oriented techniques we haven't discussed so far is hiding the data within the class and making it available only through the methods.

This technique is known as *encapsulation* because it seals the data (and internal methods) safely inside the "capsule" of the class, where it can be accessed only by trusted users (i.e., by the methods of the class).

Why would you want to do this? The most important reason is to hide the internal implementation details of your class. If you prevent programmers from relying on those details, you can safely modify the implementation without worrying that you will break existing code that uses the class.

Another reason for encapsulation is to protect your class against accidental or willful stupidity. A class often contains a number of interdependent fields that must be in a consistent state. If you allow a programmer (including yourself) to manipulate those fields directly, he may change one field without changing important related fields, thus leaving the class in an inconsistent state. If, instead, he has to call a method to change the field, that method can be sure to do everything necessary to keep the state consistent. Similarly, if a class defines certain methods for internal use only, hiding these methods prevents users of the class from calling them.

Here's another way to think about encapsulation: when all the data for a class is hidden, the methods define the only possible operations that can be performed on objects of that class. Once you have carefully tested and debugged your methods, you can be confident that the class will work as expected. On the other hand, if all the fields of the class can be directly manipulated, the number of possibilities you have to test becomes unmanageable.

There are other reasons to hide fields and methods of a class, as well:

- Internal fields and methods that are visible outside the class just clutter up the API. Keeping visible fields to a minimum keeps your class tidy and therefore easier to use and understand.
- If a field or method is visible to the users of your class, you have to document it. Save yourself time and effort by hiding it instead.

Access Control

All the fields and methods of a class can always be used within the body of the class itself. Java defines access control rules that restrict members of a class from being used outside the class. In an number of examples in this chapter, you've seen the `public` modifier used in field and method declarations. This `public` keyword, along with `protected` and `private`, are *access control modifiers*; they specify the access rules for the field or method.

Access to packages

A package is always accessible to code defined within the package. Whether it is accessible to code from other packages depends on the way the package is deployed on the host system. When the class files that comprise a package are stored in a directory, for example, a user must have read access to the directory and the files within it in order to have access to the package. Package access is not part of the Java language itself. Access control is usually done at the level of classes and members of classes instead.

Access to classes

By default, top-level classes are accessible within the package in which they are defined. However, if a top-level class is declared `public`, it is accessible everywhere (or everywhere that the package itself is accessible). The reason that we've restricted these statements to top-level classes is that, as we'll see later in this chapter, classes can also be defined as members of other classes. Because these inner classes are members of a class, they obey the member access-control rules.

Access to members

As I've already said, the members of a class are always accessible within the body of the class. By default, members are also accessible throughout the package in which the class is defined. This implies that classes placed in the same package should trust each other with their internal implementation details.* This default level of access is often called *package access*. It is only one of four possible levels of access. The other three levels of access are defined by the `public`, `protected`, and `private` modifiers. Here is some example code that uses these modifiers:

```
public class Laundromat {      // People can use this class.
  private Laundry[] dirty;     // They cannot use this internal field,
  public void wash() { ... }   // but they can use these public methods
  public void dry() { ... }    // to manipulate the internal field.
}
```

Here are the access rules that apply to members of a class:

- If a member of a class is declared with the `public` modifier, it means that the member is accessible anywhere the containing class is accessible. This is the least restrictive type of access control.
- If a member of a class is declared `private`, the member is never accessible, except within the class itself. This is the most restrictive type of access control.
- If a member of a class is declared `protected`, it is accessible to all classes within the package (the same as the default package accessibility) and also accessible within the body of any subclass of the class, regardless of the package in which that subclass is defined. This is more restrictive than `public` access, but less restrictive than package access.
- If a member of a class is not declared with any of these modifiers, it has the default package access: it is accessible to code within all classes that are defined in the same package, but inaccessible outside of the package.

`protected` access requires a little more elaboration. Suppose that the field `r` of our `Circle` class had been declared `protected` and that our `PlaneCircle` class had been defined in a different package. In this case, every `PlaneCircle` object inherits the field `r`, and the `PlaneCircle` code can use that field as it currently does. Now suppose that `PlaneCircle` defines the following method to compare the size of a `PlaneCircle` object to the size of some other `Circle` object:

---

* C++ programmers might say that all classes within a package are `friend`-ly to each other.

```
// Return true if this object is bigger than the specified circle
public boolean isBigger(Circle c) {
  return (this.r > c.r);  // If r is protected, c.r is illegal access!
}
```

In this scenario, this method does not compile. The expression `this.r` is perfectly legal, since it accesses a protected field inherited by `PlaneCircle`. Accessing `c.r` is not legal, however, since it is attempting to access a protected field it does not inherit. To make this method legal, we either have to declare `PlaneCircle` in the same package as `Circle` or change the type of the `isBigger()` parameter to be a `PlaneCircle` instead of a `Circle`.

Access control and inheritance

The Java specification states that a subclass inherits all the instance fields and instance methods of its superclass accessible to it. If the subclass is defined in the same package as the superclass, it inherits all non-`private` instance fields and methods. If the subclass is defined in a different package, however, it inherits all `protected` and `public` instance fields and methods. `private` fields and methods are never inherited; neither are class fields or class methods. Finally, constructors are not inherited; they are chained, as described earlier in this chapter.

The statement that a subclass does not inherit the inaccessible fields and methods of its superclass can be a confusing one. It would seem to imply that when you create an instance of a subclass, no memory is allocated for any `private` fields defined by the superclass. This is not the intent of the statement, however. Every instance of a subclass does, in fact, include a complete instance of the superclass within it, including all inaccessible fields and methods. It is simply a matter of terminology. Because the inaccessible fields cannot be used in the subclass, we say they are not inherited. I stated earlier in this section that the members of a class are always accessible within the body of the class. If this statement is to apply to all members of the class, including inherited members, then we have to define "inherited members" to include only those members that are accessible. If you don't care for this definition, you can think of it this way instead:

- A class inherits *all* instance fields and instance methods (but not constructors) of its superclass.
- The body of a class can always access all the fields and methods it declares itself. It can also access the *accessible* fields and members it inherits from its superclass.

Member access summary

Table 3-1 summarizes the member access rules.

Table 3-1: Class Member Accessibility

	Member Visibility			
Accessible to:	*public*	*protected*	*package*	*private*
Defining class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non-subclass different package	Yes	No	No	No

Here are some simple rules of thumb for using visibility modifiers:

- Use `public` only for methods and constants that form part of the public API of the class. Certain important or frequently used fields can also be `public`, but it is common practice to make fields non-`public` and encapsulate them with `public` accessor methods.
- Use `protected` for fields and methods that aren't required by most programmers using the class, but that may be of interest to anyone creating a subclass as part of a different package. Note that `protected` members are technically part of the exported API of a class. They should be documented and cannot be changed without potentially breaking code that relies on them.
- Use the default package visibility for fields and methods that are internal implementation details, but are used by cooperating classes in the same package. You cannot take real advantage of package visibility unless you use the `package` directive to group your cooperating classes into a package.
- Use `private` for fields and methods that are used only inside the class and should be hidden everywhere else.

If you are not sure whether to use `protected`, package, or `private` accessibility, it is better to start with overly restrictive member access. You can always relax the access restrictions in future versions of your class, if necessary. Doing the reverse is not a good idea because increasing access restrictions is not a backwards-compatible change.

Data Accessor Methods

In the `Circle` example we've been using, we've declared the circle radius to be a `public` field. The `Circle` class is one in which it may well be reasonable to keep that field publicly accessible; it is a simple enough class, with no dependencies between its fields. On the other hand, our current implementation of the class allows a `Circle` object to have a negative radius, and circles with negative radii should simply not exist. As long as the radius is stored in a `public` field, however, any programmer can set the field to any value she wants, no matter how unreasonable. The only solution is to restrict the programmer's direct access to the field and define `public` methods that provide indirect access to the field. Providing `public` methods to read and write a field is not the same as making the field itself `public`. The crucial difference is that methods can perform error checking.

Example 3-4 shows how we might reimplement Circle to prevent circles with negative radii. This version of Circle declares the r field to be protected and defines accessor methods named getRadius() and setRadius() to read and write the field value while enforcing the restriction on negative radius values. Because the r field is protected, it is directly (and more efficiently) accessible to subclasses.

Example 3-4: The Circle Class Using Data Hiding and Encapsulation

```
package shapes;           // Specify a package for the class

public class Circle {     // The class is still public
  // This is a generally useful constant, so we keep it public
  public static final double PI = 3.14159;

  protected double r;     // Radius is hidden, but visible to subclasses

  // A method to enforce the restriction on the radius
  // This is an implementation detail that may be of interest to subclasses
  protected checkRadius(double radius) {
    if (radius < 0.0)
      throw new IllegalArgumentException("radius may not be negative.");
  }

  // The constructor method
  public Circle(double r) {
    checkRadius(r);
    this.r = r;
  }

  // Public data accessor methods
  public double getRadius() { return r; };
  public void setRadius(double r) {
    checkRadius(r);
    this.r = r;
  }

  // Methods to operate on the instance field
  public double area() { return PI * r * r; }
  public double circumference() { return 2 * PI * r; }
}
```

We have defined the Circle class within a package named shapes. Since r is protected, any other classes in the shapes package have direct access to that field and can set it however they like. The assumption here is that all classes within the shapes package were written by the same author or a closely cooperating group of authors, and that the classes all trust each other not to abuse their privileged level of access to each other's implementation details.

Finally, the code that enforces the restriction against negative radius values is itself placed within a protected method, checkRadius(). Although users of the Circle class cannot call this method, subclasses of the class can call it and even override it if they want to change the restrictions on the radius.

Note particularly the getRadius() and setRadius() methods of Example 3-4. It is almost universal in Java that data accessor methods begin with the prefixes "get" and "set." If the field being accessed is of type boolean, however, the get()

method may be replaced with an equivalent method that begins with "is." For example, the accessor method for a `boolean` field named `readable` is typically called `isReadable()` instead of `getReadable()`. In the programming conventions of the JavaBeans component model (covered in Chapter 6, *JavaBeans*), a hidden field with one or more data accessor methods whose names begin with "get," "is," or "set" is called a *property*. An interesting way to study a complex class is to look at the set of properties it defines. Properties are particularly common in the AWT and Swing APIs, which are covered in *Java Foundation Classes in a Nutshell* (O'Reilly).

Abstract Classes and Methods

In Example 3-4, we declared our `Circle` class to be part of a package named `shapes`. Suppose we plan to implement a number of shape classes: `Rectangle`, `Square`, `Ellipse`, `Triangle`, and so on. We can give these shape classes our two basic `area()` and `circumference()` methods. Now, to make it easy to work with an array of shapes, it would be helpful if all our shape classes had a common superclass, `Shape`. If we structure our class hierarchy this way, every shape object, regardless of the actual type of shape it represents, can be assigned to variables, fields, or array elements of type `Shape`. We want the `Shape` class to encapsulate whatever features all our shapes have in common (e.g., the `area()` and `circumference()` methods). But our generic `Shape` class doesn't represent any real kind of shape, so it cannot define useful implementations of the methods. Java handles this situation with *abstract methods*.

Java lets us define a method without implementing it by declaring the method with the `abstract` modifier. An `abstract` method has no body; it simply has a signature definition followed by a semicolon.* Here are the rules about `abstract` methods and the `abstract` classes that contain them:

- Any class with an `abstract` method is automatically `abstract` itself and must be declared as such.
- An `abstract` class cannot be instantiated.
- A subclass of an `abstract` class can be instantiated only if it overrides each of the `abstract` methods of its superclass and provides an implementation (i.e., a method body) for all of them. Such a class is often called a *concrete* subclass, to emphasize the fact that it is not `abstract`.
- If a subclass of an `abstract` class does not implement all the `abstract` methods it inherits, that subclass is itself `abstract`.
- `static`, `private`, and `final` methods cannot be `abstract`, since these types of methods cannot be overridden by a subclass. Similarly, a `final` class cannot contain any `abstract` methods.

* An `abstract` method in Java is something like a pure virtual function in C++ (i.e., a virtual function that is declared = 0). In C++, a class that contains a pure virtual function is called an abstract class and cannot be instantiated. The same is true of Java classes that contain `abstract` methods.

- A class can be declared `abstract` even if it does not actually have any `abstract` methods. Declaring such a class `abstract` indicates that the implementation is somehow incomplete and is meant to serve as a superclass for one or more subclasses that will complete the implementation. Such a class cannot be instantiated.

There is an important feature of the rules of `abstract` methods. If we define the `Shape` class to have `abstract area()` and `circumference()` methods, any subclass of `Shape` is required to provide implementations of these methods so it can be instantiated. In other words, every `Shape` object is guaranteed to have implementations of these methods defined. Example 3-5 shows how this might work. It defines an `abstract Shape` class and two concrete subclasses of it.

Example 3-5: An Abstract Class and Concrete Subclasses

```
public abstract class Shape {
  public abstract double area();           // Abstract methods: note
  public abstract double circumference();  // semicolon instead of body.
}

class Circle extends Shape {
  public static final double PI = 3.14159265358979323846;
  protected double r;                              // Instance data
  public Circle(double r) { this.r = r; }          // Constructor
  public double getRadius() { return r; }          // Accessor
  public double area() { return PI*r*r; }          // Implementations of
  public double circumference() { return 2*PI*r; } // abstract methods.
}

class Rectangle extends Shape {
  protected double w, h;                                  // Instance data
  public Rectangle(double w, double h) {                  // Constructor
    this.w = w;  this.h = h;
  }
  public double getWidth() { return w; }                  // Accessor method
  public double getHeight() { return h; }                 // Another accessor
  public double area() { return w*h; }                    // Implementations of
  public double circumference() { return 2*(w + h); }  // abstract methods.
}
```

Each `abstract` method in `Shape` has a semicolon right after its parentheses. There are no curly braces, and no method body is defined. Using the classes defined in Example 3-5, we can now write code like this:

```
Shape[] shapes = new Shape[3];          // Create an array to hold shapes
shapes[0] = new Circle(2.0);            // Fill in the array
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);

double total_area = 0;
for(int i = 0; i < shapes.length; i++)
    total_area += shapes[i].area();     // Compute the area of the shapes
```

There are two important points to notice here:

- Subclasses of `Shape` can be assigned to elements of an array of `Shape`. No cast is necessary. This is another example of a widening reference type conversion (discussed in Chapter 2).

- You can invoke the `area()` and `circumference()` methods for any `Shape` object, even though the `Shape` class does not define a body for these methods. When you do this, the method to be invoked is found using dynamic method lookup, so the area of a circle is computed using the method defined by `Circle`, and the area of a rectangle is computed using the method defined by `Rectangle`.

Interfaces

Let's extend our `shapes` package further. Suppose we now want to implement a number of shapes that not only know their sizes, but also know the position of their center point in the Cartesian coordinate plane. One way to do this is to define an abstract `CenteredShape` class and then implement various subclasses of it, such as `CenteredCircle`, `CenteredRectangle`, and so on.

But we also want these positionable shape classes to support the `area()` and `circumference()` methods we've already defined, without reimplementing these methods. So, for example, we'd like to define `CenteredCircle` as a subclass of `Circle`, so that it inherits `area()` and `circumference()`. But a class in Java can have only one immediate superclass. If `CenteredCircle` extends `Circle`, it cannot also extend the `abstract CenteredShape` class!*

Java's solution to this problem is called an interface. Although a Java class can extend only a single superclass, it can *implement* any number of interfaces.

Defining an Interface

An interface is a reference type that is closely related to a class. Almost everything you've read so far in this book about classes applies equally to interfaces. Defining an interface is a lot like defining an abstract class, except that the keywords `abstract` and `class` are replaced with the keyword `interface`. When you define an interface, you are creating a new reference type, just as you are when you define a class. As its name implies, an *interface* specifies an interface, or API, for certain functionality. It does not define any implementation of that API, however. There are a number of restrictions that apply to the members of an interface:

- An interface contains no implementation whatsoever. All methods of an interface are implicitly `abstract`, even if the `abstract` modifier is omitted. Interface methods have no implementation; a semicolon appears in place of the method body. Because interfaces can contain only `abstract` methods, and class methods cannot be abstract, the methods of an interface must all be instance methods.
- An interface defines a public API. All methods of an interface are implicitly `public`, even if the `public` modifier is omitted. It is an error to define a `protected` or `private` method in an interface.

---

* C++ allows classes to have more than one superclass, using a technique known as multiple inheritance. Multiple inheritance adds a lot of complexity to a language; Java supports what many believe is a more elegant solution

- Although a class defines data and methods that operate on that data, an interface cannot define instance fields. Fields are an implementation detail, and an interface is a pure specification without any implementation. The only fields allowed in an interface definition are constants that are declared both `static` and `final`.
- An interface cannot be instantiated, so it does not define a constructor.

Example 3-6 shows the definition of an interface named `Centered`. This interface defines the methods a `Shape` subclass should implement if it knows the x,y coordinate of its center point.

Example 3-6: An Interface Definition

```
public interface Centered {
  public void setCenter(double x, double y);
  public double getCenterX();
  public double getCenterY();
}
```


Implementing an Interface

Just as a class uses `extends` to specify its superclass, it can use `implements` to name one or more interfaces it supports. `implements` is a Java keyword that can appear in a class declaration following the `extends` clause. `implements` should be followed by the name or names of the interface(s) the class implements, with multiple names separated by commas.

When a class declares an interface in its `implements` clause, it is saying that it provides an implementation (i.e., a body) for each method of that interface. If a class implements an interface but does not provide an implementation for every interface method, it inherits those unimplemented `abstract` methods from the interface and must itself be declared `abstract`. If a class implements more than one interface, it must implement every method of each interface it implements (or be declared `abstract`).

Example 3-7 shows how we can define a `CenteredRectangle` class that extends our `Rectangle` class and implements the `Centered` interface we defined in Example 3-6.

Example 3-7: Implementing an Interface

```
public class CenteredRectangle extends Rectangle implements Centered {
  // New instance fields
  private double cx, cy;

  // A constructor
  public CenteredRectangle(double cx, double cy, double w, double h) {
    super(w, h);
    this.cx = cx;
    this.cy = cy;
  }

  // We inherit all the methods of Rectangle, but must
  // provide implementations of all the Centered methods.
  public void setCenter(double x, double y) { cx = x; cy = y; }
```

Example 3-7: Implementing an Interface (continued)

```
  public double getCenterX() { return cx; }
  public double getCenterY() { return cy; }
}
```

As I noted earlier, constants can appear in an interface definition. Any class that implements the interface inherits the constants and can use them as if they were defined directly in the class. There is no need to prefix them with the name of the interface or provide any kind of implementation of the constants. When you have a set of constants used by more than one class (e.g., a port number and other protocol constants used by a client and server), it can be convenient to define the necessary constants in an interface that contains no methods. Then, any class that wants to use those constants needs only to declare that it implements the interface. `java.io.ObjectStreamConstants` is just such an interface.

Using Interfaces

Suppose we implement `CenteredCircle` and `CenteredSquare` just as we implemented `CenteredRectangle` in Example 3-7. Since each class extends `Shape`, instances of the classes can be treated as instances of the `Shape` class, as we saw earlier. Since each class implements `Centered`, instances can also be treated as instances of that type. The following code demonstrates both techniques:

```
Shape[] shapes = new Shape[3];          // Create an array to hold shapes

// Create some centered shapes, and store them in the Shape[]
// No cast necessary: these are all widening conversions
shapes[0] = new CenteredCircle(1.0, 1.0, 1.0);
shapes[1] = new CenteredSquare(2.5, 2, 3);
shapes[2] = new CenteredRectangle(2.3, 4.5, 3, 4);

// Compute average area of the shapes and average distance from the origin
double totalArea = 0;
double totalDistance;
for(int i = 0; i < shapes.length; i++) {
  totalArea += shapes[i].area();       // Compute the area of the shapes
  if (shapes[i] instanceof Centered) { // The shape is a Centered shape
    // Note the required cast from Shape to Centered (no cast
    // would be required to go from CenteredSquare to Centered, however).
    Centered c = (Centered) shapes[i]; // Assign it to a Centered variable
    double cx = c.getCenterX();        // Get coordinates of the center
    double cy = c.getCenterY();        // Compute distance from origin
    totalDistance += Math.sqrt(cx*cx + cy*cy);
  }
}
System.out.println("Average area: " + totalArea/shapes.length);
System.out.println("Average distance: " + totalDistance/shapes.length);
```

This example demonstrates that interfaces are data types in Java, just like classes. When a class implements an interface, instances of that class can be assigned to variables of the interface type. Don't interpret this example, however, to imply that you must assign a `CenteredRectangle` object to a `Centered` variable before you can invoke the `setCenter()` method or to a `Shape` variable before you can invoke the `area()` method. `CenteredRectangle` defines `setCenter()` and inherits `area()` from its `Rectangle` superclass, so you can always invoke these methods.

When to Use Interfaces

When defining an abstract type (e.g., `Shape`) that you expect to have many subtypes (e.g., `Circle`, `Rectangle`, `Square`), you are often faced with a choice between interfaces and abstract classes. Since they have similar features, it is not always clear when to use one over the other.

An interface is useful because any class can implement it, even if that class extends some entirely unrelated superclass. But an interface is a pure API specification and contains no implementation. If an interface has numerous methods, it can become tedious to implement the methods over and over, especially when much of the implementation is duplicated by each implementing class.

On the other hand, a class that extends an abstract class cannot extend any other class, which can cause design difficulties in some situations. However, an abstract class does not need to be entirely abstract; it can contain a partial implementation that subclasses can take advantage of. In some cases, numerous subclasses can rely on default method implementations provided by an abstract class.

Another important difference between interfaces and abstract classes has to do with compatibility. If you define an interface as part of a public API and then later add a new method to the interface, you break any classes that implemented the previous version of the interface. If you use an abstract clas, however, you can safely add nonabstract methods to that class without requiring modifications to existing classes that extend the abstract class.

In some situations, it will be clear that an interface or an abstract class is the right design choice. In other cases, a common design pattern is to use both. First, define the type as a totally abstract interface. Then create an abstract class that implements the interface and provides useful default implementations subclasses can take advantage of. For example:

```
// Here is a basic interface. It represents a shape that fits inside
// of a rectangular bounding box. Any class that wants to serve as a
// RectangularShape can implement these methods from scratch.
public interface RectangularShape {
  public void setSize(double width, double height);
  public void setPosition(double x, double y);
  public void translate(double dx, double dy);
  public double area();
  public boolean isInside();
}

// Here is a partial implementation of that interface. Many
// implementations may find this a useful starting point.
public abstract class AbstractRectangularShape implements RectangularShape {
  // The position and size of the shape
  protected double x, y, w, h;

  // Default implementations of some of the interface methods
  public void setSize(double width, double height) { w = width; h = height; }
  public void setPosition(double x, double y) { this.x = x; this.y = y; }
  public void translate (double dx, double dy) { x += dx; y += dy; }
}
```

Implementing Multiple Interfaces

Suppose we want shape objects that can be positioned in terms of not only their center points, but also their upper-left corners. And suppose we also want shapes that can be scaled larger and smaller. Remember that although a class can extend only a single superclass, it can implement any number of interfaces. Assuming we have defined appropriate `UpperRightCornered` and `Scalable` interfaces, we can declare a class as follows:

```
public class SuperDuperSquare extends Shape
       implements Centered, UpperRightCornered, Scalable {
   // class members omitted here.
}
```

When a class implements more than one interface, it simply means that it must provide implementations for all abstract methods in all its interfaces.

Extending Interfaces

Interfaces can have subinterfaces, just as classes can have subclasses. A subinterface inherits all the abstract methods and constants of its superinterface and can define new abstract methods and constants. Interfaces are different from classes in one very important way, however: an interface can have an `extends` clause that lists more than one superinterface. For example, here are some interfaces that extend other interfaces:

```
public interface Positionable extends Centered {
  public setUpperRightCorner(double x, double y);
  public double getUpperRightX();
  public double getUpperRightY();
}
public interface Transformable extends Scalable, Translatable, Rotatable {}
public interface SuperShape implements Positionable, Transformable {}
```

An interface that extends more than one interface inherits all the abstract methods and constants from each of those interfaces and can define its own additional abstract methods and constants. A class that implements such an interface must implement the abstract methods defined directly by the interface, as well as all the abstract methods inherited from all the superinterfaces.

Marker Interfaces

Sometimes it is useful to define an interface that is entirely empty. A class can implement this interface simply by naming it in its `implements` clause without having to implement any methods. In this case, any instances of the class become valid instances of the interface. Java code can check whether an object is an instance of the interface using the `instanceof` operator, so this technique is a useful way to provide additional information about an object. The `Cloneable` interface in `java.lang` is an example of this type of *marker interface*. It defines no methods, but identifies the class as one that allows its internal state to be cloned by the `clone()` method of the `Object` class. As of Java 1.1, `java.io.Serializable` is another such marker interface. Given an arbitrary object, you can determine whether it has a working `clone()` method with code like this:

```
Object o;     // Initialized elsewhere
Object copy;
if (o instanceof Cloneable) copy = o.clone();
else copy = null;
```

Inner Class Overview

The classes and interfaces we have seen so far in this chapter have all been top-level classes (i.e., they are direct members of packages, not nested within any other classes). Starting in Java 1.1, however, there are four other types of classes, loosely known as *inner classes*, that can be defined in a Java program. Used correctly, inner classes are an elegant and powerful feature of the Java language. These four types of classes are summarized here:

Static member classes
: A static member class is a class (or interface) defined as a `static` member of another class. A `static` method is called a class method, so, by analogy, we could call this type of inner class a "class class," but this terminology would obviously be confusing. A static member class behaves much like an ordinary top-level class, except that it can access the `static` members of the class that contains it. Interfaces can be defined as static members of classes.

Member classes
: A member class is also defined as a member of an enclosing class, but is not declared with the `static` modifier. This type of inner class is analogous to an instance method or field. An instance of a member class is always associated with an instance of the enclosing class, and the code of a member class has access to all the fields and methods (both `static` and non-`static`) of its enclosing class. There are several features of Java syntax that exist specifically to work with the enclosing instance of a member class. Interfaces can only be defined as static members of a class, not as non-`static` members.

Local classes
: A local class is a class defined within a block of Java code. Like a local variable, a local class is visible only within that block. Although local classes are not member classes, they are still defined within an enclosing class, so they share many of the features of member classes. Additionally, however, a local class can access any `final` local variables or parameters that are accessible in the scope of the block that defines the class. Interfaces cannot be defined locally.

Anonymous classes
: An anonymous class is a kind of local class that has no name; it combines the syntax for class definition with the syntax for object instantiation. While a local class definition is a Java statement, an anonymous class definition (and instantiation) is a Java expression, so it can appear as part of a larger expression, such as method invocation. Interfaces cannot be defined anonymously.

Java programmers have not reached a consensus on the appropriate names for the various kinds of inner classes. Thus, you may find them referred to by different names in different situations. In particular, static member classes are sometimes called "nested top-level" classes, and the term "nested classes" may refer to all

types of inner classes. The term "inner classes" is itself overloaded and sometimes refers specifically to member classes. On other occasions, "inner classes" refers to member classes, local classes, and anonymous classes, but not static member classes. In this book, I use "inner class" to mean any class other than a standard top-level class and the names shown previously to refer to the individual types of inner classes.

Static Member Classes

A *static member class* (or interface) is much like a regular top-level class (or interface). For convenience, however, it is nested within another class or interface. Example 3-8 shows a helper interface defined as a static member of a containing class. The example also shows how this interface is used both within the class that contains it and by external classes. Note the use of its hierarchical name in the external class.

Example 3-8: Defining and Using a Static Member Interface

```
// A class that implements a stack as a linked list
public class LinkedStack {
  // This static member interface defines how objects are linked
  public static interface Linkable {
    public Linkable getNext();
    public void setNext(Linkable node);
  }

  // The head of the list is a Linkable object
  Linkable head;

  // Method bodies omitted
  public void push(Linkable node) { ... }
  public Object pop() { ... }
}

// This class implements the static member interface
class LinkableInteger implements LinkedStack.Linkable {
  // Here's the node's data and constructor
  int i;
  public LinkableInteger(int i) { this.i = i; }

  // Here are the data and methods required to implement the interface
  LinkedStack.Linkable next;
  public LinkedStack.Linkable getNext() { return next; }
  public void setNext(LinkedStack.Linkable node) { next = node; }
}
```

Features of Static Member Classes

A static member class or interface is defined as a `static` member of a containing class, making it analogous to the class fields and methods that are also declared `static`. Like a class method, a static member class is not associated with any instance of the containing class (i.e., there is no `this` object). A static member class does, however, have access to all the `static` members (including any other static member classes and interfaces) of its containing class. A static member class

can use any other static member without qualifying its name with the name of the containing class.

A static member class has access to all static members of its containing class, including `private` members. The reverse is true as well: the methods of the containing class have access to all members of a static member class, including the `private` members. A static member class even has access to all the members of any other static member classes, including the `private` members of those classes.

Since static member classes are themselves class members, a static member class can be declared with its own access control modifiers. These modifiers have the same meanings for static member classes as they do for other members of a class. In Example 3-8, the `Linkable` interface is declared `public`, so it can be implemented by any class that is interested in being stored on a `LinkedStack`.

Restrictions on Static Member Classes

A static member class cannot have the same name as any of its enclosing classes. In addition, static member classes and interfaces can be defined only within top-level classes and other static member classes and interfaces. This is actually part of a larger prohibition against `static` members of any sort within member, local, and anonymous classes.

New Syntax for Static Member Classes

In code outside of the containing class, a static member class or interface is named by combining the name of the outer class with the name of the inner class (e.g., `LinkedStack.Linkable`). You can use the `import` directive to import a static member class:

```
import LinkedStack.Linkable;  // Import a specific inner class
import LinkedStack.*;         // Import all inner classes of LinkedStack
```

Importing inner classes is not recommended, however, because it obscures the fact that the inner class is tightly associated with its containing class.

Member Classes

A *member class* is a class that is declared as a non-`static` member of a containing class. If a static member class is analogous to a class field or class method, a member class is analogous to an instance field or instance method. Example 3-9 shows how a member class can be defined and used. This example extends the previous `LinkedStack` example to allow enumeration of the elements on the stack by defining an `enumerate()` method that returns an implementation of the `java.util.Enumeration` interface. The implementation of this interface is defined as a member class.

Example 3-9: An Enumeration Implemented as a Member Class

```
public class LinkedStack {
  // Our static member interface; body omitted here...
  public static interface Linkable { ... }
```

Example 3-9: An Enumeration Implemented as a Member Class (continued)

```
  // The head of the list
  private Linkable head;

  // Method bodies omitted here
  public void push(Linkable node) { ... }
  public Linkable pop() { ... }

  // This method returns an Enumeration object for this LinkedStack
public java.util.Enumeration enumerate() { return new Enumerator(); }

  // Here is the implementation of the Enumeration interface,
  // defined as a member class.
  protected class Enumerator implements java.util.Enumeration {
    Linkable current;
    // The constructor uses the private head field of the containing class
public Enumerator() { current = head; }
    public boolean hasMoreElements() {  return (current != null); }
    public Object nextElement() {
      if (current == null) throw new java.util.NoSuchElementException();
      Object value = current;
      current = current.getNext();
      return value;
    }
  }
}
```

Notice how the `Enumerator` class is nested within the `LinkedStack` class. Since `Enumerator` is a helper class used only within `LinkedStack`, there is a real elegance to having it defined so close to where it is used by the containing class.

Features of Member Classes

Like instance fields and instance methods, every member class is associated with an instance of the class within which it is defined (i.e., every instance of a member class is associated with an instance of the containing class). This means that the code of a member class has access to all the instance fields and instance methods (as well as the `static` members) of the containing class, including any that are declared `private`.

This crucial feature is illustrated in Example 3-9. Here is the body of the `LinkedStack.Enumerator()` constructor again:

```
current = head;
```

This single line of code sets the `current` field of the inner class to the value of the `head` field of the containing class. The code works as shown, even though `head` is declared as a `private` field in the containing class.

A member class, like any member of a class, can be assigned one of three visibility levels: `public`, `protected`, or `private`. If none of these visibility modifiers is specified, the default package visibility is used. In Example 3-9, the `Enumerator` class is declared `protected`, so it is inaccessible to code using the `LinkedStack` class, but accessible to any class that subclasses `LinkedStack`.

Restrictions on Member Classes

There are three important restrictions on member classes:

- A member class cannot have the same name as any containing class or package. This is an important rule, and one not shared by fields and methods.
- Member classes cannot contain any `static` fields, methods, or classes (with the exception of constant fields declared both `static` and `final`). `static` fields, methods, and classes are top-level constructs not associated with any particular object, while every member class is associated with an instance of its enclosing class. Defining a `static` top-level member within a non-top-level member class simply promotes confusion and bad programming style, so you are required to define all static members within a top-level or static member class or interface.
- Interfaces cannot be defined as member classes. An interface cannot be instantiated, so there is no object to associate with an instance of the enclosing class. If you declare an interface as a member of a class, the interface is implicitly `static`, making it a static member class.

New Syntax for Member Classes

The most important feature of a member class is that it can access the instance fields and methods in its containing object. We saw this in the `LinkedStack.Enumerator()` constructor of Example 3-9:

```
public Enumerator() { current = head; }
```

In this example, `head` is a field of the `LinkedStack` class, and we assign it to the `current` field of the `Enumerator` class. The current code works, but what if we want to make these references explicit? We could try code like this:

```
public Enumerator() { this.current = this.head; }
```

This code does not compile, however. `this.current` is fine; it is an explicit reference to the `current` field in the newly created `Enumerator` object. It is the `this.head` expression that causes the problem; it refers to a field named `head` in the `Enumerator` object. Since there is no such field, the compiler generates an error. To solve this problem, Java defines a special syntax for explicitly referring to the containing instance of the `this` object. Thus, if we want to be explicit in our constructor, we can use the following syntax:

```
public Enumerator() { this.current = LinkedStack.this.head; }
```

The general syntax is *`classname`*`.this`, where *`classname`* is the name of a containing class. Note that member classes can themselves contain member classes, nested to any depth. Since no member class can have the same name as any containing class, however, the use of the enclosing class name prepended to `this` is a perfectly general way to refer to any containing instance. This syntax is needed only when referring to a member of a containing class that is hidden by a member of the same name in the member class.

Accessing superclass members of the containing class

When a class shadows or overrides a member of its superclass, you can use the keyword `super` to refer to the hidden member. This `super` syntax can be extended to work with member classes as well. On the rare occasion when you need to refer to a shadowed field `f` or an overridden method `m` of a superclass of a containing class `C`, use the following expressions:

```
C.super.f
C.super.m()
```

This syntax was not implemented by Java 1.1 compilers, but it works correctly as of Java 1.2.

Specifying the containing instance

As we've seen, every instance of a member class is associated with an instance of its containing class. Look again at our definition of the `enumerate()` method in Example 3-9:

```
public Enumeration enumerate() { return new Enumerator(); }
```

When a member class constructor is invoked like this, the new instance of the member class is automatically associated with the `this` object. This is what you would expect to happen and exactly what you want to occur in most cases. Occasionally, however, you may want to specify the containing instance explicitly when instantiating a member class. You can do this by preceding the `new` operator with a reference to the containing instance. Thus, the `enumerate()` method shown above is shorthand for the following:

```
public Enumeration enumerate() { return this.new Enumerator(); }
```

Let's pretend we didn't define an `enumerate()` method for `LinkedStack`. In this case, the code to obtain an `Enumerator` object for a given `LinkedStack` object might look like this:

```
LinkedStack stack = new LinkedStack();     // Create an empty stack
Enumeration e = stack.new Enumerator();    // Create an Enumeration for it
```

The containing instance implicitly specifies the name of the containing class; it is a syntax error to explicitly specify that containing class:

```
Enumeration e = stack.new LinkedStack.Enumerator();  // Syntax error
```

There is one other special piece of Java syntax that specifies an enclosing instance for a member class explicitly. Before we consider it, however, let me point out that you should rarely, if ever, need to use this syntax. It is one of the pathological cases that snuck into the language along with all the elegant features of inner classes.

As strange as it may seem, it is possible for a top-level class to extend a member class. This means that the subclass does not have a containing instance, but its superclass does. When the subclass constructor invokes the superclass constructor, it must specify the containing instance. It does this by prepending the containing instance and a period to the `super` keyword. If we had not declared our

Enumerator class to be a protected member of LinkedStack, we could subclass it. Although it is not clear why we would want to do so, we could write code like the following:

```
// A top-level class that extends a member class
class SpecialEnumerator extends LinkedStack.Enumerator {
  // The constructor must explicitly specify a containing instance
  // when invoking the superclass constructor.
  public SpecialEnumerator(LinkedStack s) { s.super(); }
    // Rest of class omitted...
}
```

Scope Versus Inheritance for Member Classes

We've just noted that a top-level class can extend a member class. With the introduction of member classes, there are two separate hierarchies that must be considered for any class. The first is the *class hierarchy*, from superclass to subclass, that defines the fields and methods a member class inherits. The second is the *containment hierarchy*, from containing class to contained class, that defines a set of fields and methods that are in the scope of (and are therefore accessible to) the member class.

The two hierarchies are entirely distinct from each other; it is important that you do not confuse them. This should not be a problem if you refrain from creating naming conflicts, where a field or method in a superclass has the same name as a field or method in a containing class. If such a naming conflict does arise, however, the inherited field or method takes precedence over the field or method of the same name in the containing class. This behavior is logical: when a class inherits a field or method, that field or method effectively becomes part of that class. Therefore, inherited fields and methods are in the scope of the class that inherits them and take precedence over fields and methods by the same name in enclosing scopes.

Because this can be quite confusing, Java does not leave it to chance that you get it right. Whenever there is a naming conflict between an inherited field or method and a field or method in a containing class, Java requires that you *explicitly* specify which one you mean. For example, if a member class B inherits a field named x and is contained within a class A that also defines a field named x, you must use this.x to specify the inherited field and A.this.x to specify the field in the containing class. Any attempt to use the field x without an explicit specification of the desired instance causes a compilation error.

A good way to prevent confusion between the class hierarchy and the containment hierarchy is to avoid deep containment hierarchies. If a class is nested more than two levels deep, it is probably going to cause more confusion than it is worth. Furthermore, if a class has a deep class hierarchy (i.e., it has many superclass ancestors), consider defining it as a top-level class, rather than as a member class.

Local Classes

A *local class* is declared locally within a block of Java code, rather than as a member of a class. Typically, a local class is defined within a method, but it can also be defined within a static initializer or instance initializer of a class. Because all blocks of Java code appear within class definitions, all local classes are nested within containing classes. For this reason, local classes share many of the features of member classes. It is usually more appropriate, however, to think of them as an entirely separate kind of inner class. A local class has approximately the same relationship to a member class as a local variable has to an instance variable of a class.

The defining characteristic of a local class is that it is local to a block of code. Like a local variable, a local class is valid only within the scope defined by its enclosing block. If a member class is used only within a single method of its containing class, for example, there is usually no reason it cannot be coded as a local class, rather than a member class. Example 3-10 shows how we can modify the `enumerate()` method of the `LinkedStack` class so it defines `Enumerator` as a local class instead of a member class. By doing this, we move the definition of the class even closer to where it is used and hopefully improve the clarity of the code even further. For brevity, Example 3-10 shows only the `enumerate()` method, not the entire `LinkedStack` class that contains it.

Example 3-10: Defining and Using a Local Class

```
// This method creates and returns an Enumeration object
public java.util.Enumeration enumerate() {

  // Here's the definition of Enumerator as a local class
  class Enumerator implements java.util.Enumeration {
    Linkable current;
    public Enumerator() { current = head; }
    public boolean hasMoreElements() {  return (current != null); }
    public Object nextElement() {
      if (current == null) throw new java.util.NoSuchElementException();
      Object value = current;
      current = current.getNext();
      return value;
    }
  }

  // Now return an instance of the Enumerator class defined directly above
  return new Enumerator();
}
```

Features of Local Classes

Local classes have the following interesting features:

- Like member classes, local classes are associated with a containing instance, and can access any members, including `private` members, of the containing class.

- In addition to accessing fields defined by the containing class, local classes can access any local variables, method parameters, or exception parameters that are in the scope of the local method definition and declared `final`.

Restrictions on Local Classes

Local classes are subject to the following restrictions:

- A local class is visible only within the block that defines it; it can never be used outside that block.
- Local classes cannot be declared `public`, `protected`, `private`, or `static`. These modifiers are for members of classes; they are not allowed with local variable declarations or local class declarations.
- Like member classes, and for the same reasons, local classes cannot contain `static` fields, methods, or classes. The only exception is for constants that are declared both `static` and `final`.
- Interfaces cannot be defined locally.
- A local class, like a member class, cannot have the same name as any of its enclosing classes.
- As noted earlier, a local class can use the local variables, method parameters, and even exception parameters that are in its scope, but only if those variables or parameters are declared `final`. This is because the lifetime of an instance of a local class can be much longer than the execution of the method in which the class is defined. For this reason, a local class must have a private internal copy of all local variables it uses (these copies are automatically generated by the compiler). The only way to ensure that the local variable and the private copy are always the same is to insist that the local variable is `final`.

New Syntax for Local Classes

In Java 1.0, only fields, methods, and classes can be declared `final`. The addition of local classes in Java 1.1 has required a liberalization in the use of the `final` modifier. It can now be applied to local variables, method parameters, and even the exception parameter of a `catch` statement. The meaning of the `final` modifier remains the same in these new uses: once the local variable or parameter has been assigned a value, that value cannot be changed.

Instances of local classes, like instances of member classes, have an enclosing instance that is implicitly passed to all constructors of the local class. Local classes can use the same `this` syntax as member classes, to refer explicitly to members of enclosing classes. Because local classes are never visible outside the blocks that define them, however, there is never a need to use the `new` and `super` syntax used by member classes to specify the enclosing instance explicitly.

Scope of a Local Class

In discussing member classes, we saw that a member class can access any members inherited from superclasses and any members defined by its containing classes. The same is true for local classes, but local classes can also access `final` local variables and parameters. The following code illustrates the many fields and variables that may be accessible to a local class:

```
class A { protected char a = 'a'; }
class B { protected char b = 'b'; }

public class C extends A {
  private char c = 'c';          // Private fields visible to local class
  public static char d = 'd';
  public void createLocalObject(final char e)
  {
    final char f = 'f';
    int i = 0;                   // i not final; not usable by local class
    class Local extends B
    {
      char g = 'g';
      public void printVars()
      {
        // All of these fields and variables are accessible to this class
        System.out.println(g);  // (this.g) g is a field of this class
        System.out.println(f);  // f is a final local variable
        System.out.println(e);  // e is a final local parameter
        System.out.println(d);  // (C.this.d) d -- field of containing class
        System.out.println(c);  // (C.this.c) c -- field of containing class
        System.out.println(b);  // b is inherited by this class
        System.out.println(a);  // a is inherited by the containing class
      }
    }
    Local l = new Local();       // Create an instance of the local class
    l.printVars();               // and call its printVars() method.
  }
}
```

Local Classes and Local Variable Scope

A local variable is defined within a block of code, which defines its scope. A local variable ceases to exist outside of its scope. Java is a *lexically scoped* language, which means that its concept of scope has to do with the way the source code is written. Any code within the curly braces that define the boundaries of a block can use local variables defined in that block.*

Lexical scoping simply defines a segment of source code within which a variable can be used. It is common, however, to think of a scope as a temporal scope—to think of a local variable as existing from the time the Java interpreter begins executing the block until the time the interpreter exits the block. This is usually a reasonable way to think about local variables and their scope.

---

* This section covers advanced material; first-time readers may want to skip it for now and return to it later

The introduction of local classes confuses the picture, however, because local classes can use local variables, and instances of a local class can have a lifetime much longer than the time it takes the interpreter to execute the block of code. In other words, if you create an instance of a local class, the instance does not automatically go away when the interpreter finishes executing the block that defines the class, as shown in the following code:

```
public class Weird {
  // A static member interface used below
  public static interface IntHolder { public int getValue(); }

  public static void main(String[] args) {
    IntHolder[] holders = new IntHolder[10];   // An array to hold 10 objects
    for(int i = 0; i < 10; i++) {              // Loop to fill the array up
      final int fi = i;                        // A final local variable
      class MyIntHolder implements IntHolder { // A local class
         public int getValue() { return fi; }  // It uses the final variable
      }
      holders[i] = new MyIntHolder();          // Instantiate the local class
    }

    // The local class is now out of scope, so we can't use it. But
    // we've got ten valid instances of that class in our array. The local
    // variable fi is not in our scope here, but it is still in scope for
    // the getValue() method of each of those ten objects. So call getValue()
    // for each object and print it out. This prints the digits 0 to 9.
    for(int i = 0; i < 10; i++) System.out.println(holders[i].getValue());
  }
}
```

The behavior of the previous program is pretty surprising. To make sense of it, remember that the lexical scope of the methods of a local class has nothing to do with when the interpreter enters and exits the block of code that defines the local class. Here's another way to think about it: each instance of a local class has an automatically created private copy of each of the final local variables it uses, so, in effect, it has its own private copy of the scope that existed when it was created.

Anonymous Classes

An *anonymous class* is a local class without a name. An anonymous class is defined and instantiated in a single succinct expression using the `new` operator. While a local class definition is a statement in a block of Java code, an anonymous class definition is an expression, which means that it can be included as part of a larger expression, such as a method call. When a local class is used only once, consider using anonymous class syntax, which places the definition and use of the class in exactly the same place.

Consider Example 3-11, which shows the `Enumeration` class implemented as an anonymous class within the `enumerate()` method of the `LinkedStack` class. Compare it with Example 3-10, which shows the same class implemented as a local class.

Example 3-11: An Enumeration Implemented with an Anonymous Class

```
public java.util.Enumeration enumerate() {
  // The anonymous class is defined as part of the return statement
  return new java.util.Enumeration() {
    Linkable current; = head;
    { current = head; }  // Replace constructor with an instance initializer
    public boolean hasMoreElements() {  return (current != null); }
    public Object nextElement() {
      if (current == null) throw new java.util.NoSuchElementException();
      Object value = current;
      current = current.getNext();
      return value;
    }
  };  // Note the required semicolon: it terminates the return statement
}
```

One common use for an anonymous class is to provide a simple implementation of an adapter class. An *adapter class* is one that defines code that is invoked by some other object. Take, for example, the `list()` method of the `java.io.File` class. This method lists the files in a directory. Before it returns the list, though, it passes the name of each file to a `FilenameFilter` object you must supply. This `FilenameFilter` object accepts or rejects each file. When you implement the `FilenameFilter` interface, you are defining an adapter class for use with the `File.list()` method. Since the body of such a class is typically quite short, it is easy to define an adapter class as an anonymous class. Here's how you can define a `FilenameFilter` class to list only those files whose names end with *.java*:

```
File f = new File("/src");      // The directory to list

// Now call the list() method with a single FilenameFilter argument
// Define and instantiate an anonymous implementation of FilenameFilter
// as part of the method invocation expression.
String[] filelist = f.list(new FilenameFilter() {
  public boolean accept(File f, String s) { return s.endsWith(".java"); }
}); // Don't forget the parenthesis and semicolon that end the method call!
```

As you can see, the syntax for defining an anonymous class and creating an instance of that class uses the `new` keyword, followed by the name of a class and a class body definition in curly braces. If the name following the `new` keyword is the name of a class, the anonymous class is a subclass of the named class. If the name following `new` specifies an interface, as in the two previous examples, the anonymous class implements that interface and extends `Object`. The syntax does not include any way to specify an `extends` clause, an `implements` clause, or a name for the class.

Because an anonymous class has no name, it is not possible to define a constructor for it within the class body. This is one of the basic restrictions on anonymous classes. Any arguments you specify between the parentheses following the superclass name in an anonymous class definition are implicitly passed to the superclass constructor. Anonymous classes are commonly used to subclass simple classes that do not take any constructor arguments, so the parentheses in the anonymous class definition syntax are often empty. In the previous examples, each anonymous class implemented an interface and extended `Object`. Since the `Object()` constructor takes no arguments, the parentheses were empty in those examples.

Features of Anonymous Classes

One of the most elegant things about anonymous classes is that they allow you to define a one-shot class exactly where it is needed. In addition, anonymous classes have a succinct syntax that reduces clutter in your code.

Restrictions on Anonymous Classes

Because an anonymous class is just a type of local class, anonymous classes and local classes share the same restrictions. An anonymous class cannot define any `static` fields, methods, or classes, except for `static final` constants. Interfaces cannot be defined anonymously, since there is no way to implement an interface without a name. Also, like local classes, anonymous classes cannot be `public`, `private`, `protected`, or `static`.

Since an anonymous class has no name, it is not possible to define a constructor for an anonymous class. If your class requires a constructor, you must use a local class instead. However, you can often use an instance initializer as a substitute for a constructor. In fact, instance initializers were introduced into the language for this very purpose.

The syntax for defining an anonymous class combines definition with instantiation. Thus, using an anonymous class instead of a local class is not appropriate if you need to create more than a single instance of the class each time the containing block is executed.

New Syntax for Anonymous Classes

We've already seen examples of the syntax for defining and instantiating an anonymous class. We can express that syntax more formally as:

```
new class-name ( [ argument-list ] ) { class-body }
```

or:

```
new interface-name () { class-body }
```

As I already mentioned, instance initializers are another specialized piece of Java syntax that was introduced to support anonymous classes. As we discussed earlier in the chapter, an instance initializer is a block of initialization code contained within curly braces inside a class definition. The contents of an instance initializer for a class are automatically inserted into all constructors for the class, including any automatically created default constructor. An anonymous class cannot define a constructor, so it gets a default constructor. By using an instance initializer, you can get around the fact that you cannot define a constructor for an anonymous class.

When to Use an Anonymous Class

As we've discussed, an anonymous class behaves just like a local class and is distinguished from a local class merely in the syntax used to define and instantiate it. In your own code, when you have to choose between using an anonymous class

and a local class, the decision often comes down to a matter of style. You should use whichever syntax makes your code clearer. In general, you should consider using an anonymous class instead of a local class if:

- The class has a very short body.
- Only one instance of the class is needed.
- The class is used right after it is defined.
- The name of the class does not make your code any easier to understand.

Anonymous Class Indentation and Formatting

The common indentation and formatting conventions we are familiar with for block-structured languages like Java and C begin to break down somewhat once we start placing anonymous class definitions within arbitrary expressions. Based on their experience with inner classes, the engineers at Sun recommend the following formatting rules:

- The opening curly brace should not be on a line by itself; instead, it should follow the close parenthesis of the `new` operator. Similarly, the `new` operator should, when possible, appear on the same line as the assignment or other expression of which it is a part.
- The body of the anonymous class should be indented relative to the beginning of the line that contains the `new` keyword.
- The closing curly brace of an anonymous class should not be on a line by itself either; it should be followed by whatever tokens are required by the rest of the expression. Often this is a semicolon or a close parenthesis followed by a semicolon. This extra punctuation serves as a flag to the reader that this is not just an ordinary block of code and makes it easier to understand anonymous classes in a code listing.

How Inner Classes Work

The preceding sections have explained the features and behavior of the various types of inner classes. Strictly speaking, that should be all you need to know about inner classes. In practice, however, some programmers find it easier to understand the details of inner classes if they understand how they are implemented.

Inner classes were introduced in Java 1.1. Despite the dramatic changes to the Java language, the introduction of inner classes did not change the Java Virtual Machine or the Java class file format. As far as the Java interpreter is concerned, there is no such thing as an inner class: all classes are normal top-level classes. In order to make an inner class behave as if it is actually defined inside another class, the Java compiler ends up inserting hidden fields, methods, and constructor arguments into the classes it generates. You may want to use the *javap* disassembler to disassemble some of the class files for inner classes so you can see what tricks the compiler has used to make inner classes work. (See Chapter 8, *Java Development Tools*, for information on *javap*.)

Static Member Class Implementation

Recall our first `LinkedStack` example (Example 3-8), which defined a static member interface named `Linkable`. When you compile this `LinkedStack` class, the compiler actually generates two class files. The first one is *LinkedStack.class*, as expected. The second class file, however, is called *LinkedStack$Linkable.class*. The `$` in this name is automatically inserted by the Java compiler. This second class file contains the implementation of the static member interface.

As we discussed earlier, a static member class can access all the `static` members of its containing class. If a static member class does this, the compiler automatically qualifies the member access expression with the name of the containing class. A static member class is even allowed to access the `private static` fields of its containing class. Since the static member class is compiled into an ordinary top-level class, however, there is no way it can directly access the `private` members of its container. Therefore, if a static member class uses a `private` member of its containing class (or vice versa), the compiler automatically generates non-`private` access methods and converts the expressions that access the `private` members into expressions that access these specially generated methods. These methods are given the default package access, which is sufficient, as the member class and its containing class are guaranteed to be in the same package.

Member Class Implementation

A member class is implemented much like a static member class. It is compiled into a separate top-level class file, and the compiler performs various code manipulations to make interclass member access work correctly.

The most significant difference between a member class and a static member class is that each instance of a member class is associated with an instance of the enclosing class. The compiler enforces this association by defining a synthetic field named `this$0` in each member class. This field is used to hold a reference to the enclosing instance. Every member class constructor is given an extra parameter that initializes this field. Every time a member class constructor is invoked, the compiler automatically passes a reference to the enclosing class for this extra parameter.

As we've seen, a member class, like any member of a class, can be declared `public`, `protected`, or `private`, or given the default package visibility. However, as I mentioned earlier, there have been no changes to the Java Virtual Machine to support member classes. Member classes are compiled to class files just like top-level classes, but top-level classes can only have public or package access. Therefore, as far as the Java interpreter is concerned, member classes can only have public or package visibility. This means that a member class declared `protected` is actually treated as a public class, and a member class declared `private` actually has package visibility. This does not mean you should never declare a member class as `protected` or `private`. Although the interpreter cannot enforce these access control modifiers, the modifiers are noted in the class file. This allows any conforming Java compiler to enforce the access modifiers and prevent the member classes from being accessed in unintended ways.

Local and Anonymous Class Implementation

A local class is able to refer to fields and methods in its containing class for exactly the same reason that a member class can; it is passed a hidden reference to the containing class in its constructor and saves that reference away in a `private` field added by the compiler. Also, like member classes, local classes can use `private` fields and methods of their containing class because the compiler inserts any required accessor methods.

What makes local classes different from member classes is that they have the ability to refer to local variables in the scope that defines them. The crucial restriction on this ability, however, is that local classes can only reference local variables and parameters that are declared `final`. The reason for this restriction becomes apparent from the implementation. A local class can use local variables because the compiler automatically gives the class a `private` instance field to hold a copy of each local variable the class uses. The compiler also adds hidden parameters to each local class constructor to initialize these automatically created `private` fields. Thus, a local class does not actually access local variables, but merely its own private copies of them. The only way this can work correctly is if the local variables are declared `final`, so that they are guaranteed not to change. With this guarantee, the local class can be assured that its internal copies of the variables are always in sync with the real local variables.

Since anonymous classes have no names, you may wonder what the class files that represent them are named. This is an implementation detail, but the Java compiler from Sun uses numbers to provide anonymous class names. If you compile the example shown in Example 3-11, you'll find that it produces a file with a name like *LinkedStack$1.class*. This is the class file for the anonymous class.

Modifier Summary

As we've seen, classes, interfaces, and their members can be declared with one or more *modifiers*—keywords such as `public`, `static`, and `final`. This chapter has introduced the `public`, `protected`, and `private` access modifiers, as well as the `abstract`, `final`, and `static` modifiers. In addition to these six, Java defines five other less commonly used modifiers. Table 3-2 lists the Java modifiers, explains what types of Java constructs they can modify, and explains what they do.

Table 3-2: Java Modifiers

Modifier	*Used on*	*Meaning*
`abstract`	class	The class contains unimplemented methods and cannot be instantiated.
	interface	All interfaces are `abstract`. The modifier is optional in interface declarations.

Table 3-2: Java Modifiers (continued)

Modifier	*Used on*	*Meaning*
`abstract`	method	No body is provided for the method; it is provided by a subclass. The signature is followed by a semicolon. The enclosing class must also be `abstract`.
`final`	class	The class cannot be subclassed.
	method	The method cannot be overridden (and is not subject to dynamic method lookup).
	field	The field cannot have its value changed. `static final` fields are compile-time constants.
	variable	A local variable, method parameter, or exception parameter cannot have its value changed (Java 1.1 and later). Useful with local classes.
`native`	method	The method is implemented in some platform-dependent way (often in C). No body is provided; the signature is followed by a semicolon.
none (package)	class	A non-`public` class is accessible only in its package.
	interface	A non-`public` interface is accessible only in its package.
	member	A member that is not `private`, `protected`, or `public` has package visibility and is accessible only within its package.
`private`	member	The member is accessible only within the class that defines it.
`protected`	member	The member is accessible only within the package in which it is defined and within subclasses.
`public`	class	The class is accessible anywhere its package is.
	interface	The interface is accessible anywhere its package is.
	member	The member is accessible anywhere its class is.
`strictfp`	class	All methods of the class are implicitly `strictfp` (Java 1.2 and later).

Table 3-2: Java Modifiers (continued)

Modifier	*Used on*	*Meaning*
`strictfp`	method	All floating-point computation done by the method must be performed in a way that strictly conforms to the IEEE 754 standard. In particular, all values, including intermediate results, must be expressed as IEEE `float` or `double` values and cannot take advantage of any extra precision or range offered by native platform floating-point formats or hardware (Java 1.2 and later). This modifier is rarely used.
`static`	class	An inner class declared `static` is a top-level class, not associated with a member of the containing class (Java 1.1 and later).
	method	A `static` method is a class method. It is not passed an implicit `this` object reference. It can be invoked through the class name.
	field	A `static` field is a class field. There is only one instance of the field, regardless of the number of class instances created. It can be accessed through the class name.
	initializer	The initializer is run when the class is loaded, rather than when an instance is created.
`synchronized`	method	The method makes non-atomic modifications to the class or instance, so care must be taken to ensure that two threads cannot modify the class or instance at the same time. For a `static` method, a lock for the class is acquired before executing the method. For a non-`static` method, a lock for the specific object instance is acquired.
`transient`	field	The field is not part of the persistent state of the object and should not be serialized with the object. Used with object serialization; see `java.io.ObjectOutputStream`.
`volatile`	field	The field can be accessed by unsynchronized threads, so certain optimizations must not be performed on it. This modifier can sometimes be used as an alternative to `synchronized`. This modifier is very rarely used.

C++ Features Not Found in Java

Throughout this chapter, I've noted similarities and differences between Java and C++ in footnotes. Java shares enough concepts and features with C++ to make it an easy language for C++ programmers to pick up. There are several features of C++ that have no parallel in Java, however. In general, Java does not adopt those features of C++ that make the language significantly more complicated.

C++ supports multiple inheritance of method implementations from more than one superclass at a time. While this seems like a useful feature, it actually introduces many complexities to the language. The Java language designers chose to avoid the added complexity by using interfaces instead. Thus, a class in Java can inherit method implementations only from a single superclass, but it can inherit method declarations from any number of interfaces.

C++ supports templates that allow you, for example, to implement a `Stack` class and then instantiate it as `Stack<int>` or `Stack<double>` to produce two separate types: a stack of integers and a stack of floating-point values. Java does not allow this, but efforts are underway to add this feature to the language in a robust and standardized way. Furthermore, the fact that every class in Java is a subclass of `Object` means that every object can be cast to an instance of `Object`. Thus, in Java it is often sufficient to define a data structure (such as a `Stack` class) that operates on `Object` values; the objects can be cast back to their actual types whenever necessary.

C++ allows you to define operators that perform arbitrary operations on instances of your classes. In effect, it allows you to extend the syntax of the language. This is a nifty feature, called operator overloading, that makes for elegant examples. In practice, however, it tends to make code quite difficult to understand. After much debate, the Java language designers decided to omit such operator overloading from the language. Note, though, that the use of the + operator for string concatenation in Java is at least reminiscent of operator overloading.

C++ allows you to define conversion functions for a class that automatically invoke an appropriate constructor method when a value is assigned to a variable of that class. This is simply a syntactic shortcut (similar to overriding the assignment operator) and is not included in Java.

In C++, objects are manipulated by value by default; you must use & to specify a variable or function argument automatically manipulated by reference. In Java, all objects are manipulated by reference, so there is no need for this & syntax.

CHAPTER 4

The Java Platform

Chapter 2, *Java Syntax from the Ground Up*, and Chapter 3, *Object-Oriented Programming in Java*, documented the Java programming language. This chapter switches gears and covers the Java platform, which is the vast collection of predefined classes available to every Java program, regardless of the underlying host system on which it is running. The classes of the Java platform are collected into related groups, known as *packages*. This chapter begins with an overview of the packages of the Java platform that are documented in this book. It then moves on to demonstrate, in the form of short examples, the most useful classes in these packages.

Java Platform Overview

Table 4-1 summarizes the key packages of the Java platform that are covered in this book.

Table 4-1: Key Packages of the Java Platform

Package	*Description*
`java.beans`	The JavaBeans component model for reusable, embeddable software components.
`java.beans.beancontext`	Additional classes that define bean context objects that hold and provide services to the JavaBeans objects they contain.
`java.io`	Classes and interfaces for input and output. Although some of the classes in this package are for working directly with files, most are for working with streams of bytes or characters.
`java.lang`	The core classes of the language, such as `String`, `Math`, `System`, `Thread`, and `Exception`.

Table 4-1: Key Packages of the Java Platform (continued)

Package	*Description*
`java.lang.ref`	Classes that define weak references to objects. A weak reference is one that does not prevent the referent object from being garbage-collected.
`java.lang.reflect`	Classes and interfaces that allow Java programs to reflect on themselves by examining the constructors, methods, and fields of classes.
`java.math`	A small package that contains classes for arbitrary-precision integer and floating-point arithmetic.
`java.net`	Classes and interfaces for networking with other systems.
`java.security`	Classes and interfaces for access control and authentication. Supports cryptographic message digests and digital signatures.
`java.security.acl`	A package that supports access control lists. Deprecated and unused as of Java 1.2.
`java.security.cert`	Classes and interfaces for working with public key certificates.
`java.security.interfaces`	Interfaces used with DSA and RSA public-key encryption.
`java.security.spec`	Classes and interfaces for transparent representations of keys and parameters used in public-key cryptography.
`java.text`	Classes and interfaces for working with text in internationalized applications.
`java.util`	Various utility classes, including the powerful collections framework for working with collections of objects.
`java.util.jar`	Classes for reading and writing JAR files.
`java.util.zip`	Classes for reading and writing ZIP files.
`javax.crypto`	Classes and interfaces for encryption and decryption of data.
`javax.crypto.interfaces`	Interfaces that represent the Diffie-Hellman public/private keys used in the Diffie-Hellman key agreement protocol.
`javax.crypto.spec`	Classes that define transparent representations of keys and parameters used in cryptography.

Table 4-1 does not list all the packages in the Java platform, only those documented in this book. Java also defines numerous packages for graphics and graphical user interface programming and for distributed, or enterprise, computing. The

graphics and GUI packages are `java.awt` and `javax.swing` and their many subpackages. These packages, along with the `java.applet` package, are documented in *Java Foundation Classes in a Nutshell* (O'Reilly). The enterprise packages of Java include `java.rmi`, `java.sql`, `javax.jndi`, `org.omg.CORBA`, `org.omg.CosNaming`, and all of their subpackages. These packages, as well as several standard extensions to the Java platform, are documented in the book *Java Enterprise in a Nutshell* (O'Reilly).

Strings and Characters

Strings of text are a fundamental and commonly used data type. In Java, however, strings are not a primitive type, like `char`, `int`, and `float`. Instead, strings are represented by the `java.lang.String` class, which defines many useful methods for manipulating strings. `String` are *immutable*: once a `String` object has been created, there is no way to modify the string of text it represents. Thus, each method that operates on a string typically returns a new `String` object that holds the modified string.

This code shows some of the basic operations you can perform on strings:

```
// Creating strings
String s = "Now";               // String objects have a special literal syntax
String t = s + " is the time."; // Concatenate strings with + operator
String t1 = s + " " + 23.4;     // + converts other values to strings
t1 = String.valueOf('c');       // Get string corresponding to char value
t1 = String.valueOf(42);        // Get string version of integer or any value
t1 = Object.toString();         // Convert objects to strings with toString()

// String length
int len = t.length();           // Number of characters in the string: 16

// Substrings of a string
String sub = t.substring(4);    // Returns char 4 to end: "is the time."
sub = t.substring(4, 6);        // Returns chars 4 and 5: "is"
sub = t.substring(0, 3);        // Returns chars 0 through 2: "Now"
sub = t.substring(x, y);        // Returns chars between pos x and y-1
int numchars = sub.length();    // Length of substring is always (y-x)

// Extracting characters from a string
char c = t.charAt(2);           // Get the 3rd character of t: w
char[] ca = t.toCharArray();    // Convert string to an array of characters
t.getChars(0, 3, ca, 1);        // Put 1st 4 chars of s into ca at position 2

// Case conversion
String caps = t.toUpperCase();  // Convert to uppercase
String lower = t.toLowerCase(); // Convert to lowercase

// Comparing strings
boolean b1 = t.equals("hello");         // Returns false: strings not equal
boolean b2 = t.equalsIgnoreCase(caps);  // Case-insensitive compare: true
boolean b3 = t.startsWith("Now");       // Returns true
boolean b4 = t.endsWith("time.");       // Returns true
int r1 = s.compareTo("Pow");            // Returns < 0: s comes before "Pow"
int r2 = s.compareTo("Now");            // Returns 0: strings are equal
int r3 = s.compareTo("Mow");            // Returns > 0: s comes after "Mow"
r1 = s.compareToIgnoreCase("pow");      // Returns < 0 (Java 1.2 and later)
```

```
// Searching for characters and substrings
int pos = t.indexOf('i');          // Position of first 'i': 4
pos = t.indexOf('i', pos+1);       // Position of the next 'i': 12
pos = t.indexOf('i', pos+1);       // No more 'i's in string, returns -1
pos = t.lastIndexOf('i');          // Position of last 'i' in string: 12
pos = t.lastIndexOf('i', pos-1);   // Search backwards for 'i' from char 11

pos = t.indexOf("is");             // Search for substring: returns 4
pos = t.indexOf("is", pos+1);      // Only appears once: returns -1
pos = t.lastIndexOf("the ");       // Search backwards for a string
String noun = t.substring(pos+4); // Extract word following "the"

// Replace all instances of one character with another character
String exclaim = t.replace('.', '!');  // Only works with chars, not substrings

// Strip blank space off the beginning and end of a string
String noextraspaces = t.trim();

// Obtain unique instances of strings with intern()
String s1 = s.intern();        // Returns s1 equal to s
String s2 = "Now".intern();    // Returns s2 equal to "Now"
boolean equals = (s1 == s2);   // Now can test for equality with ==
```

Since `String` objects are immutable, you cannot manipulate the characters of a `String` in place. If you need to do this, use a `java.lang.StringBuffer` instead:

```
// Create a string buffer from a string
StringBuffer b = new StringBuffer("Mow");

// Get and set individual characters of the StringBuffer
char c = b.charAt(0);          // Returns 'M': just like String.charAt()
b.setCharAt(0, 'N');           // b holds "Now": can't do that with a String!

// Append to a StringBuffer
b.append(' ');                 // Append a character
b.append("is the time.");      // Append a string
b.append(23);                  // Append an integer or any other value

// Insert Strings or other values into a StringBuffer
b.insert(6, "n't");            // b now holds: "Now isn't the time.23"

// Replace a range of characters with a string (Java 1.2 and later)
b.replace(4, 9, "is");         // Back to "Now is the time.23"

// Delete characters
b.delete(16, 18);              // Delete a range: "Now is the time"
b.deleteCharAt(2);             // Delete 2nd character: "No is the time"
b.setLength(5);                // Truncate by setting the length: "No is"

// Other useful operations
b.reverse();                   // Reverse characters: "si oN"
String s = b.toString();       // Convert back to an immutable string
s = b.substring(1,2);          // Or take a substring: "i"
b.setLength(0);                // Erase buffer; now it is ready for reuse
```

In addition to the `String` and `StringBuffer` classes, there are a number of other Java classes that operate on strings. One notable class is `java.util.StringTokenizer`, which you can use to break a string of text into its component words:

```
String s = "Now is the time";
java.util.StringTokenizer st = new java.util.StringTokenizer(s);
while(st.hasMoreTokens()) {
  System.out.println(st.nextToken());
}
```

You can even use this class to tokenize words that are delimited by characters other than spaces:

```
String s = "a:b:c:d";
java.util.StringTokenizer st = new java.util.StringTokenizer(s, ":");
```

As you know, individual characters are represented in Java by the primitive `char` type. The Java platform also defines a `Character` class, which defines useful class methods for checking the type of a character and converting the case of a character. For example:

```
char[] text;  // An array of characters, initialized somewhere else
int p = 0;    // Our current position in the array of characters
// Skip leading whitespace
while((p < text.length) && Character.isWhitespace(text[p])) p++;
// Capitalize the first word of text
while((p < text.length) && Character.isLetter(text[p])) {
  text[p] = Character.toUpperCase(text[p]);
  p++;
}
```

The `compareTo()` and `equals()` methods of the `String` class allow you to compare strings. `compareTo()` bases its comparison on the character order defined by the Unicode encoding, while `equals()` defines string equality as strict character-by-character equality. These are not always the right methods to use, however. In some languages, the character ordering imposed by the Unicode standard does not match the dictionary ordering used when alphabetizing strings. In Spanish, for example, the letters "ch" are considered a single letter that comes after "c" and before "d." When comparing human-readable strings in an internationalized application, you should use the `java.text.Collator` class instead:

```
import java.text.*;

// Compare two strings; results depend on where the program is run
// Return values of Collator.compare() have same meanings as String.compareTo()
Collator c = Collator.getInstance();       // Get Collator for current locale
int result = c.compare("chica", "coche");  // Use it to compare two strings
```

Numbers and Math

Java provides the `byte`, `short`, `int`, `long`, `float`, and `double` primitive types for representing numbers. The `java.lang` package includes the corresponding `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` classes, each of which is a subclass of `Number`. These classes can be useful as object wrappers around their primitive types, and they also define some useful constants:

```
// Integral range constants: Integer, Long, and Character also define these
Byte.MIN_VALUE      // The smallest (most negative) byte value
Byte.MAX_VALUE      // The largest byte value
Short.MIN_VALUE     // The most negative short value
```

```
Short.MAX_VALUE     // The largest short value

// Floating-point range constants: Double also defines these
Float.MIN_VALUE     // Smallest (closest to zero) positive float value
Float.MAX_VALUE     // Largest positive float value

// Other useful constants
Math.PI             // 3.14159265358979323846
Math.E              // 2.7182818284590452354
```

A Java program that operates on numbers must get its input values from somewhere. Often, such a program reads a textual representation of a number and must convert it to a numeric representation. The various `Number` subclasses define useful conversion methods:

```
String s = "-42";
byte b = Byte.parseByte(s);             // s as a byte
short sh = Short.parseShort(sh);        // s as a short
int i = Integer.parseInt(s);            // s as an int
long l = Long.parseLong(s);             // s as a long
float f = Float.parseFloat(s);          // s as a float (Java 1.2 and later)
f = Float.valueOf(s).floatValue();      // s as a float (prior to Java 1.2)
double d = Double.parseDouble(s);       // s as a double (Java 1.2 and later)
d = Double.valueOf(s).doubleValue();    // s as a double (prior to Java 1.2)

// The integer conversion routines handle numbers in other bases
byte b = Byte.parseByte("1011", 2);      // 1011 in binary is 11 in decimal
short sh = Short.parseShort("ff", 16);   // ff in base 16 is 255 in decimal

// The valueOf() method can handle arbitrary bases
int i = Integer.valueOf("egg", 17).intValue();   // Base 17!

// The decode() method handles octal, decimal, or hexadecimal, depending
// on the numeric prefix of the string
short sh = Short.decode("0377").byteValue();   // Leading 0 means base 8
int i = Integer.decode("0xff").shortValue();   // Leading 0x means base 16
long l = Long.decode("255").intValue();        // Other numbers mean base 10

// Integer class can convert numbers to strings
String decimal = Integer.toString(42);
String binary = Integer.toBinaryString(42);
String octal = Integer.toOctalString(42);
String hex = Integer.toHexString(42);
String base36 = Integer.toString(42, 36);
```

Numeric values are often printed differently in different countries. For example, many European languages use a comma to separate the integral part of a floating-point value from the fractional part (instead of a decimal point). Formatting differences can diverge even further when displaying numbers that represent monetary values. When converting numbers to strings for display, therefore, it is best to use the `java.text.NumberFormat` class to perform the conversion in a locale-specific way:

```
import java.text.*;

// Use NumberFormat to format and parse numbers for the current locale
NumberFormat nf = NumberFormat.getNumberInstance();  // Get a NumberFormat
System.out.println(nf.format(9876543.21)); // Format number for current locale
try {
```

```
  Number n = nf.parse("1.234.567,89");     // Parse strings according to locale
} catch (ParseException e) { /* Handle exception */ }

// Monetary values are sometimes formatted differently than other numbers
NumberFormat moneyFmt = NumberFormat.getCurrencyInstance();
System.out.println(moneyFmt.format(1234.56)); // Prints $1,234.56 in U.S.
```

The Math class defines a number of methods that provide trigonometric, logarithmic, exponential, and rounding operations, among others. This class is primarily useful with floating-point values. For the trigonometric functions, angles are expressed in radians. The logarithm and exponentiation functions are base *e*, not base 10. Here are some examples:

```
double d = Math.toRadians(27);       // Convert 27 degrees to radians
d = Math.cos(d);                     // Take the cosine
d = Math.sqrt(d);                    // Take the square root
d = Math.log(d);                     // Take the natural logarithm
d = Math.exp(d);                     // Do the inverse: e to the power d
d = Math.pow(10, d);                 // Raise 10 to this power
d = Math.atan(d);                    // Compute the arc tangent
d = Math.toDegrees(d);               // Convert back to degrees
double up = Math.ceil(d);            // Round to ceiling
double down = Math.floor(d);         // Round to floor
long nearest = Math.round(d);        // Round to nearest
```

The Math class also defines a rudimentary method for generating pseudo-random numbers, but the java.util.Random class is more flexible. If you need *very* random pseudo-random numbers, you can use the java.security.SecureRandom class:

```
// A simple random number
double r = Math.random();     // Returns d such that: 0.0 <= d < 1.0

// Create a new Random object, seeding with the current time
java.util.Random generator = new java.util.Random(System.currentTimeMillis());
double d = generator.nextDouble();   // 0.0 <= d < 1.0
float f = generator.nextFloat();     // 0.0 <= d < 1.0
long l = generator.nextLong();       // Chosen from the entire range of long
int i = generator.nextInt();         // Chosen from the entire range of int
i = generator.nextInt(limit);        // 0 <= i < limit (Java 1.2 and later)
boolean b = generator.nextBoolean(); // true or false (Java 1.2 and later)
d = generator.nextGaussian();        // Mean value: 0.0; std. deviation: 1.0
byte[] randomBytes = new byte[128];
generator.nextBytes(randomBytes);    // Fill in array with random bytes

// For cryptographic strength random numbers, use the SecureRandom subclass
java.security.SecureRandom generator2 = new java.security.SecureRandom();
// Have the generator generate its own 16-byte seed; takes a *long* time
generator2.setSeed(generator2.generateSeed(16));  // Extra random 16-byte seed
// Then use SecureRandom like any other Random object
generator2.nextBytes(randomBytes);   // Generate more random bytes
```

The java.math package contains the BigInteger and BigDecimal classes. These classes allow you to work with arbitrary-size and arbitrary-precision integers and floating-point values. For example:

```
import java.math.*;

// Compute the factorial of 1000
```

```
BigInteger total = BigInteger.valueOf(1);
for(int i = 2; i <= 1000; i++)
  total = total.multiply(BigInteger.valueOf(i));
System.out.println(total.toString());
```

Dates and Times

Java uses several different classes for working with dates and times. The `java.util.Date` class represents an instant in time (precise down to the millisecond). This class is nothing more than a wrapper around a `long` value that holds the number of milliseconds since midnight GMT, January 1, 1970. Here are two ways to determine the current time:

```
long t0 = System.currentTimeMillis();      // Current time in milliseconds
java.util.Date now = new java.util.Date(); // Basically the same thing
long t1 = now.getTime();                   // Convert a Date to a long value
```

The `Date` class has a number of interesting-sounding methods, but almost all of them have been deprecated in favor of methods of the `java.util.Calendar` and `java.text.DateFormat` classes. To print a date or a time, use the `DateFormat` class, which automatically handles locale-specific conventions for date and time formatting. `DateFormat` even works correctly in locales that use a calendar other than the common era (Gregorian) calendar in use in much of the world:


```
import java.util.Date;
import java.text.*;

// Display today's date using a default format for the current locale
DateFormat defaultDate = DateFormat.getDateInstance();
System.out.println(defaultDate.format(new Date()));

// Display the current time using a short time format for the current locale
DateFormat shortTime = DateFormat.getTimeInstance(DateFormat.SHORT);
System.out.println(shortTime.format(new Date()));

// Display date and time using a long format for both
DateFormat longTimestamp =
  DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL);
System.out.println(longTimestamp.format(new Date()));

// Use SimpleDateFormat to define your own formatting template
// See java.text.SimpleDateFormat for the template syntax
DateFormat myformat = new SimpleDateFormat("yyyy.MM.dd");
System.out.println(myformat.format(new Date()));
try {   // DateFormat can parse dates too
  Date leapday = myformat.parse("2000.02.29");
}
catch (ParseException e) { /* Handle parsing exception */ }
```

The `Date` class and its millisecond representation allow only a very simple form of date arithmetic:

```
long now = System.currentTimeMillis();         // The current time
long anHourFromNow = now + (60 * 60 * 1000);   // Add 3,600,000 milliseconds
```

To perform more sophisticated date and time arithmetic and manipulate dates in ways humans (rather than computers) typically care about, use the `java.util.Calendar` class:

```
import java.util.*;

// Get a Calendar for current locale and time zone
Calendar cal = Calendar.getInstance();

// Figure out what day of the year today is
cal.setTime(new Date());                        // Set to the current time
int dayOfYear = cal.get(Calendar.DAY_OF_YEAR);  // What day of the year is it?

// What day of the week does the leap day in the year 2000 occur on?
cal.set(2000, Calendar.FEBRUARY, 29);           // Set year, month, day fields
int dayOfWeek = cal.get(Calendar.DAY_OF_WEEK);  // Query a different field

// What day of the month is the 3rd Thursday of May, 2001?
cal.set(Calendar.YEAR, 2001);                     // Set the year
cal.set(Calendar.MONTH, Calendar.MAY);            // Set the month
cal.set(Calendar.DAY_OF_WEEK, Calendar.THURSDAY); // Set the day of week
cal.set(Calendar.DAY_OF_WEEK_IN_MONTH, 3);        // Set the week
int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);  // Query the day in month

// Get a Date object that represents 30 days from now
Date today = new Date();                  // Current date
cal.setTime(today);                       // Set it in the Calendar object
cal.add(Calendar.DATE, 30);               // Add 30 days
Date expiration = cal.getTime();          // Retrieve the resulting date
```

Arrays

The `java.lang.System` class defines an `arraycopy()` method that is useful for copying specified elements in one array to a specified position in a second array. The second array must be the same type as the first, and it can even be the same array:

```
char[] text = "Now is the time".toCharArray();
char[] copy = new char[100];
// Copy 10 characters from element 4 of text into copy, starting at copy[0]
System.arraycopy(text, 4, copy, 0, 10);

// Move some of the text to later elements, making room for insertions
System.arraycopy(copy, 3, copy, 6, 7);
```

In Java 1.2 and later, the `java.util.Arrays` class defines useful array-manipulation methods, including methods for sorting and searching arrays:

```
import java.util.Arrays;

int[] intarray = new int[] { 10, 5, 7, -3 }; // An array of integers
Arrays.sort(intarray);                       // Sort it in place
int pos = Arrays.binarySearch(intarray, 7);  // Value 7 is found at index 2
pos = Arrays.binarySearch(intarray, 12);     // Not found: negative return value

// Arrays of objects can be sorted and searched too
String[] strarray = new String[] { "now", "is", "the", "time" };
Arrays.sort(strarray);      // { "is", "now", "the", "time" }
```

```
// Arrays.equals() compares all elements of two arrays
String[] clone = (String[]) strarray.clone();
boolean b1 = Arrays.equals(strarray, clone);  // Yes, they're equal

// Arrays.fill() initializes array elements
byte[] data = new byte[100];          // An empty array; elements set to 0
Arrays.fill(data, (byte) -1);         // Set them all to -1
Arrays.fill(data, 5, 10, (byte) -2);  // Set elements 5, 6, 7, 8, 9 to -2
```

Arrays can be treated and manipulated as objects in Java. Given an arbitrary object `o`, you can use code such as the following to find out if the object is an array and, if so, what type of array it is:

```
Class type = o.getClass();
if (type.isArray()) {
  Class elementType = type.getComponentType();
}
```

Collections

The Java collection framework is a set of important utility classes and interfaces in the `java.util` package for working with collections of objects. The collection framework defines two fundamental types of collections. A `Collection` is a group of objects, while a `Map` is a set of mappings, or associations, between objects. A `Set` is a type of `Collection` in which there are no duplicates, and a `List` is a `Collection` in which the elements are ordered. `Collection`, `Set`, `List`, and `Map` are all interfaces, but the `java.util` package also defines various concrete implementations (see Chapter 23, *The java.util Package*). Other important interfaces are `Iterator` and `ListIterator`, which allow you to loop through the objects in a collection. The collection framework is new as of Java 1.2, but prior to that release you can use `Vector` and `Hashtable`, which are approximately the same as `ArrayList` and `HashMap`.

The following code demonstrates how you might create and perform basic manipulations on sets, lists, and maps:

```
import java.util.*;

Set s = new HashSet();            // Implementation based on a hash table
s.add("test");                    // Add a String object to the set
boolean b = s.contains("test2"); // Check whether a set contains an object
s.remove("test");                 // Remove a member from a set

Set ss = new TreeSet();           // TreeSet implements SortedSet
ss.add("b");                      // Add some elements
ss.add("a");
// Now iterate through the elements (in sorted order) and print them
for(Iterator i = ss.iterator(); i.hasNext();)
  System.out.println(i.next());

List l = new LinkedList();        // LinkedList implements a doubly linked list
l = new ArrayList();              // ArrayList is more efficient, usually
Vector v = new Vector();          // Vector is an alternative in Java 1.1/1.0
l.addAll(ss);                     // Append some elements to it
l.addAll(1, ss);                  // Insert the elements again at index 1
Object o = l.get(1);              // Get the second element
```

```
l.set(3, "new element");         // Set the fourth element
l.add("test");                   // Append a new element to the end
l.add(0, "test2");               // Insert a new element at the start
l.remove(1);                     // Remove the second element
l.remove("a");                   // Remove the element "a"
l.removeAll(ss);                 // Remove elements from this set
if (!l.isEmpty())                // If list is not empty,
  System.out.println(l.size());  // print out the number of elements in it
boolean b1 = l.contains("a");    // Does it contain this value?
boolean b2 = l.containsAll(ss);  // Does it contain all these values?
List sublist = l.subList(1,3);   // A sublist of the 2nd and 3rd elements
Object[] elements = l.toArray(); // Convert it to an array
l.clear();                       // Delete all elements

Map m = new HashMap();           // Hashtable an alternative in Java 1.1/1.0
m.put("key", new Integer(42));   // Associate a value object with a key object
Object value = m.get("key");     // Look up the value associated with a key
m.remove("key");                 // Remove the association from the Map
Set keys = m.keySet();           // Get the set of keys held by the Map
```

Arrays of objects and collections serve similar purposes. It is possible to convert from one to the other:

```
Object[] members = set.toArray();         // Get set elements as an array
Object[] items = list.toArray();          // Get list elements as an array
Object[] keys = map.keySet().toArray();   // Get map key objects as an array
Object[] values = map.values().toArray(); // Get map value objects as an array

List l = Arrays.asList(a);                // View array as an ungrowable list
List l = new ArrayList(Arrays.asList(a)); // Make a growable copy of it
```

Just as the `java.util.Arrays` class defined methods to operate on arrays, the `java.util.Collections` class defines methods to operate on collections. Most notable are methods to sort and search the elements of collections:

```
Collections.sort(list);
int pos = Collections.binarySearch(list, "key"); // list must be sorted first
```

Here are some other interesting `Collections` methods:

```
Collections.copy(list1, list2); // Copy list2 into list1, overwriting list1
Collections.fill(list, o);      // Fill list with Object o
Collections.max(c);             // Find the largest element in Collection c
Collections.min(c);             // Find the smallest element in Collection c
Collections.reverse(list);      // Reverse list
Collections.shuffle(list);      // Mix up list

Set s = Collections.singleton(o); // Return an immutable set with one element o
List ul = Collections.unmodifiableList(list); // Immutable wrapper for list
Map sm = Collections.synchronizedMap(map);    // Synchronized wrapper for map
```

One particularly useful collection class is `java.util.Properties`. `Properties` is a subclass of `Hashtable` that predates the collections framework of Java 1.2, making it a legacy collection. A `Properties` object maintains a mapping between string keys and string values, and defines methods that allow the mappings to be written to and read from a simple-format text file. This makes the `Properties` class ideal for configuration and user preference files. The `Properties` class is also used for the system properties returned by `System.getProperty()`:

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

String homedir = System.getProperty("user.home");  // Get a system property
Properties sysprops = System.getProperties();      // Get all system properties

// Print the names of all defined system properties
for(Enumeration e = sysprops.propertyNames(); e.hasMoreElements();)
  System.out.println(e.nextElement());

sysprops.list(System.out);  // Here's an even easier way to list the properties

// Read properties from a configuration file
Properties options = new Properties();             // Empty properties list
File configfile = new File(homedir, ".config");    // The configuration file
try {
  options.load(new FileInputStream(configfile));   // Load props from the file
} catch (IOException e) { /* Handle exception here */ }

// Query a property ("color"), specifying a default ("gray") if undefined
String color = options.getProperty("color", "gray");

// Set a property named "color" to the value "green"
options.setProperty("color", "green");

// Store the contents of the Properties object back into a file
try {
  options.store(new FileOutputStream(configfile),  // Output stream
                "MyApp Config File");              // File header comment text
} catch (IOException e) { /* Handle exception */ }
```


Types, Reflection, and Dynamic Loading

The `java.lang.Class` class represents data types in Java and, along with the classes in the `java.lang.reflect` package, gives Java programs the capability of introspection (or self-reflection); a Java class can look at itself, or any other class, and determine its superclass, what methods it defines, and so on. There are several ways you can obtain a `Class` object in Java:

```
// Obtain the Class of an arbitrary object o
Class c = o.getClass();

// Obtain a Class object for primitive types with various predefined constants
c = Void.TYPE;          // The special "no-return-value" type
c = Byte.TYPE;          // Class object that represents a byte
c = Integer.TYPE;       // Class object that represents an int
c = Double.TYPE;        // etc. See also Short, Character, Long, Float.

// Express a class literal as a type name followed by ".class"
c = int.class;          // Same as Integer.TYPE
c = String.class;       // Same as "dummystring".getClass()
c = byte[].class;       // Type of byte arrays
c = Class[][].class;    // Type of array of arrays of Class objects
```

Once you have a `Class` object, you can perform some interesting reflective operations with it:

```
import java.lang.reflect.*;

Object o;                    // Some unknown object to investigate
Class c = o.getClass();      // Get its type

// If it is an array, figure out its base type
while (c.isArray()) c = c.getComponentType();

// If c is not a primitive type, print its class hierarchy
if (!c.isPrimitive()) {
  for(Class s = c; s != null; s = s.getSuperclass())
    System.out.println(s.getName() + " extends");
}

// Try to create a new instance of c; this requires a no-arg constructor
Object newobj = null;
try { newobj = c.newInstance(); }
catch (Exception e) {
  // Handle InstantiationException, IllegalAccessException
}

// See if the class has a method named setText that takes a single String
// If so, call it with a string argument
try {
  Method m = c.getMethod("setText", new Class[] { String.class });
  m.invoke(newobj, new Object[] { "My Label" });
} catch(Exception e) { /* Handle exceptions here */ }
```

`Class` also provides a simple mechanism for dynamic class loading in Java. For more complete control over dynamic class loading, however, you should use a `java.lang.ClassLoader` object, typically a `java.net.URLClassLoader`. This technique is useful, for example, when you want to load a class that is named in a configuration file instead of being hardcoded into your program:

```
// Dynamically load a class specified by name in a config file
String classname =                    // Look up the name of the class
  config.getProperties("filterclass",  // The property name
                       "com.davidflangan.filters.Default"); // A default

try {
  Class c = Class.forName(classname);  // Dynamically load the class
  Object o = c.newInstance();          // Dynamically instantiate it
} catch (Exception e) { /* Handle exceptions */ }

// If the class to be loaded is not in the classpath, create a custom
// class loader to load it.
// Use the config file again to specify the custom path
import java.net.URLClassLoader;
String classdir = config.getProperties("classpath");
try {
  ClassLoader loader = new URLClassLoader(new URL[] { new URL(classdir) });
  Class c = loader.loadClass(classname);
}
catch (Exception e) { /* Handle exceptions */ }
```

Threads

Java makes it easy to define and work with multiple threads of execution within a program. `java.lang.Thread` is the fundamental thread class in the Java API. There are two ways to define a thread. One is to subclass `Thread`, override the `run()` method, and then instantiate your `Thread` subclass. The other is to define a class that implements the `Runnable` method (i.e., define a `run()` method) and then pass an instance of this `Runnable` object to the `Thread()` constructor. In either case, the result is a `Thread` object, where the `run()` method is the body of the thread. When you call the `start()` method of the `Thread` object, the interpreter creates a new thread to execute the `run()` method. This new thread continues to run until the `run()` method exits, at which point it ceases to exist. Meanwhile, the original thread continues running itself, starting with the statement following the `start()` method. The following code demonstrates:

```
final List list;  // Some long unsorted list of objects; initialized elsewhere

/** A Thread class for sorting a List in the background */
class BackgroundSorter extends Thread {
  List l;
  public BackgroundSorter(List l) { this.l = l; }    // Constructor
  public void run() { Collections.sort(l); }         // Thread body
}

// Create a BackgroundSorter thread
Thread sorter = new BackgroundSorter(list);
// Start it running; the new thread runs the run() method above, while
// the original thread continues with whatever statement comes next.
sorter.start();

// Here's another way to define a similar thread
Thread t = new Thread(new Runnable() {          // Create a new thread
  public void run() { Collections.sort(list); } // to sort the list of objects.
});
t.start();                                      // Start it running
```

Threads can run at different priority levels. A thread at a given priority level does not run unless there are no higher-priority threads waiting to run. Here is some code you can use when working with thread priorities:

```
// Set a thread t to lower-than-normal priority
t.setPriority(Thread.NORM_PRIORITY-1);

// Set a thread to lower priority than the current thread
t.setPriority(Thread.currentThread().getPriority() - 1);

// Threads that don't pause for I/O should explicitly yield the CPU
// to give other threads with the same priority a chance to run.
Thread t = new Thread(new Runnable() {
  public void run() {
    for(int i = 0; i < data.length; i++) {  // Loop through a bunch of data
      process(data[i]);                     // Process it
      if ((i % 10) == 0)                    // But after every 10 iterations,
        Thread.yield();                     // pause to let other threads run.
    }
  }
});
```

Often, threads are used to perform some kind of repetitive task at a fixed interval. This is particularly true when doing graphical programming that involves animation or similar effects:

```
public class Clock extends Thread {
  java.text.DateFormat f =      // How to format the time for this locale
    java.text.DateFormat.getTimeInstance(java.text.DateFormat.MEDIUM);
  boolean keepRunning = true;

  public Clock() {          // The constructor
    setDaemon(true);        // Daemon thread: interpreter can exit while it runs
    start();                // This thread starts itself
  }

  public void run() {       // The body of the thread
    while(keepRunning) {    // This thread runs until asked to stop
      String time = f.format(new java.util.Date()); // Current time
      System.out.println(time);                     // Print the time
      try { Thread.sleep(1000); }                   // Wait 1000 milliseconds
      catch (InterruptedException e) {}             // Ignore this exception
    }
  }

  // Ask the thread to stop running
  public void pleaseStop() { keepRunning = false; }
}
```

Notice the `pleaseStop()` method in the previous example. You can forcefully terminate a thread by calling its `stop()` method, but this method has been deprecated because a thread that is forcefully stopped can leave objects it is manipulating in an inconsistent state. If you need a thread that can be stopped, you should define a method such as `pleaseStop()` that stops the thread in a controlled way.

In Java 1.3, the `java.util.Timer` and `java.util.TimerTask` classes make it even easier to run repetitive tasks. Here is some code that behaves much like the previous `Clock` class:

```
import java.util.*;

// How to format the time for this locale
final java.text.DateFormat timeFmt =
  java.text.DateFormat.getTimeInstance(java.text.DateFormat.MEDIUM);
// Define the time-display task
TimerTask displayTime = new TimerTask() {
  public void run() { System.out.println(timeFmt.format(new Date())); }
};
// Create a timer object to run the task (and possibly others)
Timer timer = new Timer();
// Now schedule that task to be run every 1000 milliseconds, starting now
Timer.schedule(displayTime, 0, 1000);
// To stop the time-display task
displayTime.cancel();
```

Sometimes one thread needs to stop and wait for another thread to complete. You can accomplish this with the `join()` method:

```
List list;  // A long list of objects to be sorted; initialized elsewhere

// Define a thread to sort the list: lower its priority, so it only runs
// when the current thread is waiting for I/O, and then start it running.
Thread sorter = new BackgroundSorter(list);              // Defined earlier
sorter.setPriority(Thread.currentThread.getPriority()-1); // Lower priority
sorter.start();                                          // Start sorting

// Meanwhile, in this original thread, read data from a file
byte[] data = readData();  // Method defined elsewhere

// Before we can proceed, we need the list to be fully sorted, so
// we've got to wait for the sorter thread to exit, if it hasn't already.
sorter.join();
```

When using multiple threads, you must be very careful if you allow more than one thread to access the same data structure. Consider what would happen if one thread was trying to loop through the elements of a `List` while another thread was sorting those elements. Preventing this problem is called *thread synchronization* and is one of the central problems of multithreaded computing. The basic technique for preventing two threads from accessing the same object at the same time is to require a thread to obtain a lock on the object before the thread can modify it. While any one thread holds the lock, another thread that requests the lock has to wait until the first thread is done and releases the lock. Every Java object has the fundamental ability to provide such a locking capability.

The easiest way to keep objects thread-safe is to declare any sensitive methods `synchronized`. A thread must obtain a lock on an object before it can execute any of its `synchronized` methods, which means that no other thread can execute any other `synchronized` method at the same time. (If a `static` method is declared `synchronized`, the thread must obtain a lock on the class, and this works in the same manner.) To do finer-grained locking, you can specify `synchronized` blocks of code that hold a lock on a specified object for a short time:

```
// This method swaps two array elements in a synchronized block
public static void swap(Object[] array, int index1, int index2) {
  synchronized(array) {
    Object tmp = array[index1];
    array[index1] = array[index2];
    array[index2] = tmp;
  }
}

// The Collection, Set, List, and Map implementations in java.util do
// not have synchronized methods (except for the legacy implementations
// Vector and Hashtable). When working with multiple threads, you can
// obtain synchronized wrapper objects.
List synclist = Collections.synchronizedList(list);
Map syncmap = Collections.synchronizedMap(map);
```

When you are synchronizing threads, you must be careful to avoid *deadlock*, which occurs when two threads end up waiting for each other to release a lock they need. Since neither can proceed, neither one can release the lock it holds, and they both stop running:

```
// When two threads try to lock two objects, deadlock can occur unless
// they always request the locks in the same order.
final Object resource1 = new Object();   // Here are two objects to lock
final Object resource2 = new Object();
Thread t1 = new Thread(new Runnable() {  // Locks resource1 then resource2
  public void run() {
    synchronized(resource1) {
      synchronized(resource2) { compute(); }
    }
  }
});

Thread t2 = new Thread(new Runnable() {  // Locks resource2 then resource1
  public void run() {
    synchronized(resource2) {
      synchronized(resource1) { compute(); }
    }
  }
});

t1.start();  // Locks resource1
t2.start();  // Locks resource2 and now neither thread can progress!
```

Sometimes a thread needs to stop running and wait until some kind of event occurs, at which point it is told to continue running. This is done with the `wait()` and `notify()` methods. These aren't methods of the `Thread` class, however; they are methods of `Object`. Just as every Java object has a lock associated with it, every object can maintain a list of waiting threads. When a thread calls the `wait()` method of an object, it is added to the list of waiting threads for that object and stops running. When another thread calls the `notify()` method of the same object, the object wakes up one of the waiting threads and allows it to continue running:

```
/**
 * A queue. One thread calls push() to put an object on the queue.
 * Another calls pop() to get an object off the queue. If there is no
 * data, pop() waits until there is some, using wait()/notify().
 * wait() and notify() must be used within a synchronized method or
 * block.
 */
import java.util.*;

public class Queue {
  LinkedList q = new LinkedList();  // Where objects are stored
  public synchronized void push(Object o) {
    q.add(o);      // Append the object to the end of the list
    this.notify(); // Tell waiting threads that data is ready
  }
  public synchronized Object pop() {
    while(q.size() == 0) {
      try { this.wait(); }
      catch (InterruptedException e) { /* Ignore this exception */ }
    }
    return q.remove(0);
  }
}
```

Files and Directories

The `java.io.File` class represents a file or a directory and defines a number of important methods for manipulating files and directories. Note, however, that none of these methods allow you to read the contents of a file; that is the job of `java.io.FileInputStream`, which is just one of the many types of input/output streams used in Java and discussed in the next section. Here are some things you can do with `File`:

```
import java.io.*;

// Get the name of the user's home directory and represent it with a File
File homedir = new File(System.getProperty("user.home"));

// Create a File object to represent a file in that directory
File f = new File(homedir, ".configfile");

// Find out how big a file is and when it was last modified
long filelength = f.length();
Date lastModified = new java.util.Date(f.lastModified());

// If the file exists, is not a directory, and is readable,
// move it into a newly created directory.
if (f.exists() && f.isFile() && f.canRead()) {       // Check config file
  File configdir = new File(homedir, ".configdir");  // A new config directory
  configdir.mkdir();                                 // Create that directory
  f.renameTo(new File(configdir, ".config"));        // Move the file into it
}

// List all files in the home directory
String[] allfiles = homedir.list();

// List all files that have a ".java" suffix
String[] sourcecode = homedir.list(new FilenameFilter() {
  public boolean accept(File d, String name) { return name.endsWith(".java"); }
});
```

The `File` class provides some important additional functionality as of Java 1.2:

```
// List all filesystem root directories; on Windows, this gives us
// File objects for all drive letters (Java 1.2 and later).
File[] rootdirs = File.listRoots();

// Atomically, create a lock file, then delete it (Java 1.2 and later)
File lock = new File(configdir, ".lock");
if (lock.createNewFile()) {
  // We successfully created the file, so do something
  ...
  // Then delete the lock file
  lock.delete();
}
else {
  // We didn't create the file; someone else has a lock
  System.err.println("Can't create lock file; exiting.");
  System.exit(0);
}

// Create a temporary file to use during processing (Java 1.2 and later)
File temp = File.createTempFile("app", ".tmp");  // Filename prefix and suffix
```

```
// Make sure file gets deleted when we're done with it (Java 1.2 and later)
temp.deleteOnExit();
```

The `java.io` package also defines a `RandomAccessFile` class that allows you to read binary data from arbitrary locations in a file. This can be a useful thing to do in certain situations, but most applications read files sequentially, using the stream classes described in the next section. Here is a short example of using `RandomAccessFile`:

```
// Open a file for read/write ("rw") access
File datafile = new File(configdir, "datafile");
RandomAccessFile f = new RandomAccessFile(datafile, "rw");
f.seek(100);                // Move to byte 100 of the file
byte[] data = new byte[100]; // Create a buffer to hold data
f.read(data);               // Read 100 bytes from the file
int i = f.readInt();        // Read a 4-byte integer from the file
f.seek(100);                // Move back to byte 100
f.writeInt(i);              // Write the integer first
f.write(data);              // Then write the 100 bytes
f.close();                  // Close file when done with it
```

Input and Output Streams

The `java.io` package defines a large number of classes for reading and writing streaming, or sequential, data. The `InputStream` and `OutputStream` classes are for reading and writing streams of bytes, while the `Reader` and `Writer` classes are for reading and writing streams of characters. Streams can be nested, meaning you might read characters from a `FilterReader` object that reads and processes characters from an underlying `Reader` stream. This underlying `Reader` stream might read bytes from an `InputStream` and convert them to characters.

There are a number of common operations you can perform with streams. One is to read lines of input the user types at the console:

```
import java.io.*;

BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
System.out.print("What is your name: ");
String name = null;
try {
  name = console.readLine();
}
catch (IOException e) { name = "<" + e + ">"; }  // This should never happen
System.out.println("Hello " + name);
```

Reading lines of text from a file is a similar operation. The following code reads an entire text file and quits when it reaches the end:

```
String filename = System.getProperty("user.home") + File.separator + ".cshrc";
try {
  BufferedReader in = new BufferedReader(new FileReader(filename));
  String line;
  while((line = in.readLine()) != null) {  // Read line, check for end-of-file
    System.out.println(line);              // Print the line
  }
  in.close();    // Always close a stream when you are done with it
}
```

```
catch (IOException e) {
  // Handle FileNotFoundException, etc. here
}
```

Throughout this book, you've seen the use of the `System.out.println()` method to display text on the console. `System.out` simply refers to an output stream. You can print text to any output stream using similar techniques. The following code shows how to output text to a file:

```
try {
  File f = new File(homedir, ".config");
  PrintWriter out = new PrintWriter(new FileWriter(f));
  out.println("## Automatically generated config file. DO NOT EDIT!");
  out.close();  // We're done writing
}
catch (IOException e) { /* Handle exceptions */ }
```

Not all files contain text, however. The following lines of code treat a file as a stream of bytes and read the bytes into a large array:

```
try {
  File f;                              // File to read; initialized elsewhere
  int filesize = (int) f.length();     // Figure out the file size
  byte[] data = new byte[filesize];    // Create an array that is big enough
  // Create a stream to read the file
  DataInputStream in = new DataInputStream(new FileInputStream(f));
  in.readFully(data);  // Read file contents into array
  in.close();
}
catch (IOException e) { /* Handle exceptions */ }
```


Various other packages of the Java platform define specialized stream classes that operate on streaming data in some useful way. The following code shows how to use stream classes from `java.util.zip` to compute a checksum of data and then compress the data while writing it to a file:

```
import java.io.*;
import java.util.zip.*;

try {
  File f;                            // File to write to; initialized elsewhere
  byte[] data;                       // Data to write; initialized elsewhere
  Checksum check = new Adler32();    // An object to compute a simple checksum

  // Create a stream that writes bytes to the file f
  FileOutputStream fos = new FileOutputStream(f);
  // Create a stream that compresses bytes and writes them to fos
  GZIPOutputStream gzos = new GZIPOutputStream(fos);
  // Create a stream that computes a checksum on the bytes it writes to gzos
  CheckedOutputStream cos = new CheckedOutputStream(gzos, check);

  cos.write(data);             // Now write the data to the nested streams
  cos.close();                 // Close down the nested chain of streams
  long sum = check.getValue(); // Obtain the computed checksum
}
catch (IOException e) { /* Handle exceptions */ }
```

The java.util.zip package also contains a ZipFile class that gives you random access to the entries of a ZIP archive and allows you to read those entries through a stream:

```
import java.io.*;
import java.util.zip.*;

String filename;     // File to read; initialized elsewhere
String entryname;    // Entry to read from the ZIP file; initialized elsewhere
ZipFile zipfile = new ZipFile(filename);        // Open the ZIP file
ZipEntry entry = zipfile.getEntry(entryname);   // Get one entry
InputStream in = zipfile.getInputStream(entry); // A stream to read the entry
BufferedInputStream bis = new BufferedInputStream(in);  // Improves efficiency
// Now read bytes from bis...
// Print out contents of the ZIP file
for(java.util.Enumeration e = zipfile.entries(); e.hasMoreElements();) {
  ZipEntry zipentry = (ZipEntry) e.nextElement();
  System.out.println(zipentry.getName());
}
```

If you need to compute a cryptographic-strength checksum (also knows as a message digest), use one of the stream classes of the java.security package. For example:

```
import java.io.*;
import java.security.*;
import java.util.*;

File f;          // File to read and compute digest on; initialized elsewhere
List text = new ArrayList();  // We'll store the lines of text here

// Get an object that can compute an SHA message digest
MessageDigest digester = MessageDigest.getInstance("SHA");
// A stream to read bytes from the file f
FileInputStream fis = new FileInputStream(f);
// A stream that reads bytes from fis and computes an SHA message digest
DigestInputStream dis = new DigestInputStream(fis, digester);
// A stream that reads bytes from dis and converts them to characters
InputStreamReader isr = new InputStreamReader(dis);
// A stream that can read a line at a time
BufferedReader br = new BufferedReader(isr);
// Now read lines from the stream
for(String line; (line = br.readLine()) != null; text.add(line)) ;
// Close the streams
br.close();
// Get the message digest
byte[] digest = digester.digest();
```

So far, we've used a variety of stream classes to manipulate streaming data, but the data itself ultimately comes from a file or is written to the console. The java.io package defines other stream classes that can read data from and write data to arrays of bytes or strings of text:

```
import java.io.*;

// Set up a stream that uses a byte array as its destination
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(baos);
out.writeUTF("hello");            // Write some string data out as bytes
```

```
out.writeDouble(Math.PI);         // Write a floating-point value out as bytes
byte[] data = baos.toByteArray(); // Get the array of bytes we've written
out.close();                      // Close the streams

// Set up a stream to read characters from a string
Reader in = new StringReader("Now is the time!");
// Read characters from it until we reach the end
int c;
while((c = in.read()) != -1) System.out.print((char) c);
```

Other classes that operate this way include `ByteArrayInputStream`, `StringWriter`, `CharArrayReader`, and `CharArrayWriter`.

`PipedInputStream` and `PipedOutputStream` and their character-based counterparts, `PipedReader` and `PipedWriter`, are another interesting set of streams defined by `java.io`. These streams are used in pairs by two threads that want to communicate. One thread writes bytes to a `PipedOutputStream` or characters to a `PipedWriter`, and another thread reads bytes or characters from the corresponding `PipedInputStream` or `PipedReader`:

```
// A pair of connected piped I/O streams forms a pipe. One thread writes
// bytes to the PipedOutputStream, and another thread reads them from the
// corresponding PipedInputStream. Or use PipedWriter/PipedReader for chars.
final PipedOutputStream writeEndOfPipe = new PipedOutputStream();
final PipedInputStream readEndOfPipe = new PipedInputStream(writeEndOfPipe);

// This thread reads bytes from the pipe and discards them
Thread devnull = new Thread(new Runnable() {
  public void run() {
    try { while(readEndOfPipe.read() != -1); }
    catch (IOException e) {}  // ignore it
  }
});
devnull.start();
```

One of the most important features of the `java.io` package is the ability to *serialize* objects: to convert an object into a stream of bytes that can later be deserialized back into a copy of the original object. The following code shows how to use serialization to save an object to a file and later read it back:

```
Object o;  // The object we are serializing; it must implement Serializable
File f;    // The file we are saving it to

try {
  // Serialize the object
  ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(f));
  oos.writeObject(o);
  oos.close();

  // Read the object back in:
  ObjectInputStream ois = new ObjectInputStream(new FileInputStream(f));
  Object copy = ois.readObject();
  ois.close();
}
catch (IOException e) { /* Handle input/output exceptions */ }
catch (ClassNotFoundException cnfe) { /* readObject() can throw this */ }
```

The previous example serializes to a file, but remember, you can write serialized objects to any type of stream. Thus, you can write an object to a byte array, then

read it back from the byte array, creating a deep copy of the object. You can write the object's bytes to a compression stream or even write the bytes to a stream connected across a network to another program!

Networking

The `java.net` package defines a number of classes that make writing networked applications surprisingly easy. The easiest class to use is `URL`, which represents a uniform resource locator. Different Java implementations may support different sets of URL protocols, but, at a minimum, you can rely on support for the `http:`, `ftp:`, and `file:` protocols. Here are some ways you can use the `URL` class:

```
import java.net.*;
import java.io.*;

// Create some URL objects
URL url=null, url2=null, url3=null;
try {
  url = new URL("http://www.oreilly.com");        // An absolute URL
  url2 = new URL(url, "catalog/books/javanut3/"); // A relative URL
  url3 = new URL("http:", "www.oreilly.com", "index.html");
} catch (MalformedURLException e) { /* Ignore this exception */ }

// Read the content of a URL from an input stream:
InputStream in = url.openStream();

// For more control over the reading process, get a URLConnection object
URLConnection conn = url.openConnection();

// Now get some information about the URL
String type = conn.getContentType();
String encoding = conn.getContentEncoding();
java.util.Date lastModified = new java.util.Date(conn.getLastModified());
int len = conn.getContentLength();

// If necessary, read the contents of the URL using this stream
InputStream in = conn.getInputStream();
```

Sometimes you need more control over your networked application than is possible with the `URL` class. In this case, you can use a `Socket` to communicate directly with a server. For example:

```
import java.net.*;
import java.io.*;

// Here's a simple client program that connects to a web server,
// requests a document, and reads the document from the server.
String hostname = "java.oreilly.com";  // The server to connect to
int port = 80;                         // Standard port for HTTP
String filename = "/index.html";       // The file to read from the server
Socket s = new Socket(hostname, port); // Connect to the server

// Get I/O streams we can use to talk to the server
InputStream sin = s.getInputStream();
BufferedReader fromServer = new BufferedReader(new InputStreamReader(sin));
OutputStream sout = s.getOutputStream();
PrintWriter toServer = new PrintWriter(new OutputStreamWriter(sout));
```

```
// Request the file from the server, using the HTTP protocol
toServer.print("GET " + filename + " HTTP/1.0\n\n");
toServer.flush();

// Now read the server's response, assume it is a text file, and print it out
for(String l = null; (l = fromServer.readLine()) != null; )
  System.out.println(l);

// Close everything down when we're done
toServer.close();
fromServer.close();
s.close();
```

A client application uses a `Socket` to communicate with a server. The server does the same thing: it uses a `Socket` object to communicate with each of its clients. However, the server has an additional task, in that it must be able to recognize and accept client connection requests. This is done with the `ServerSocket` class. The following code shows how you might use a `Server Socket`. The code implements a simple HTTP server that responds to all requests by sending back (or mirroring) the exact contents of the HTTP request. A dummy server like this is useful when debugging HTTP clients:

```
import java.io.*;
import java.net.*;

public class HttpMirror {
  public static void main(String[] args) {
    try {
      int port = Integer.parseInt(args[0]);        // The port to listen on
      ServerSocket ss = new ServerSocket(port);    // Create a socket to listen
      for(;;) {                                    // Loop forever
        Socket client = ss.accept();               // Wait for a connection
        ClientThread t = new ClientThread(client); // A thread to handle it
        t.start();                                 // Start the thread running
      }                                            // Loop again
    }
    catch (Exception e) {
      System.err.println(e.getMessage());
      System.err.println("Usage: java HttpMirror <port>");
    }
  }

  static class ClientThread extends Thread {
    Socket client;
    ClientThread(Socket client) { this.client = client; }
    public void run() {
      try {
        // Get streams to talk to the client
        BufferedReader in =
          new BufferedReader(new InputStreamReader(client.getInputStream()));
        PrintWriter out =
          new PrintWriter(new OutputStreamWriter(client.getOutputStream()));

        // Send an HTTP response header to the client
        out.print("HTTP/1.0 200\nContent-Type: text/plain\n\n");

        // Read the HTTP request from the client and send it right back
        // Stop when we read the blank line from the client that marks
        // the end of the request and its headers.
```

```
        String line;
        while((line = in.readLine()) != null) {
          if (line.length() == 0) break;
          out.println(line);
        }

        out.close();
        in.close();
        client.close();
      }
      catch (IOException e) { /* Ignore exceptions */ }
    }
  }
}
```

Note how elegantly both the URL and Socket classes use the input/output streams that we saw earlier in the chapter. This is one of the features that makes the Java networking classes so powerful.

Both URL and Socket perform networking on top of a stream-based network connection. Setting up and maintaining a stream across a network takes work at the network level, however. Sometimes you need a low-level way to speed a packet of data across a network, but you don't care about maintaining a stream. If, in addition, you don't need a guarantee that your data will get there or that the packets of data will arrive in the order you sent them, you may be interested in the DatagramSocket and DatagramPacket classes:

```
import java.net.*;

// Send a message to another computer via a datagram
try {
  String hostname = "host.domain.org";      // The computer to send the data to
  InetAddress address =                     // Convert the DNS hostname
    InetAddress.getByName(hostname);        // to a lower-level IP address.
  int port = 1234;                          // The port to connect to
  String message = "The eagle has landed."; // The message to send
  byte[] data = message.getBytes();         // Convert string to bytes
  DatagramSocket s = new DatagramSocket();  // Socket to send message with
  DatagramPacket p =                        // Create the packet to send
    new DatagramPacket(data, data.length, address, port);
  s.send(p);                                // Now send it!
  s.close();                                // Always close sockets when done
}
catch (UnknownHostException e) {}  // Thrown by InetAddress.getByName()
catch (SocketException e) {}       // Thrown by new DatagramSocket()
catch (java.io.IOException e) {}   // Thrown by DatagramSocket.send()

// Here's how the other computer can receive the datagram
try {
  byte[] buffer = new byte[4096];               // Buffer to hold data
  DatagramSocket s = new DatagramSocket(1234);  // Socket to receive it through
  DatagramPacket p =
    new DatagramPacket(buffer, buffer.length);  // The packet to receive it
  s.receive(p);                                 // Wait for a packet to arrive
  String msg =                                  // Convert the bytes from the
    new String(buffer, 0, p.getLength());       // packet back to a string.
  s.close();                                    // Always close the socket
}
```

```
catch (SocketException e) {}       // Thrown by new DatagramSocket()
catch (java.io.IOException e) {}   // Thrown by DatagramSocket.receive()
```

Processes

Earlier in the chapter, we saw how easy it is to create and manipulate multiple threads of execution running within the same Java interpreter. Java also has a `java.lang.Process` class that represents a program running externally to the interpreter. A Java program can communicate with an external process using streams in the same way that it might communicate with a server running on some other computer on the network. Using a `Process` is always platform-dependent and is rarely portable, but it is sometimes a useful thing to do:

```
// Maximize portability by looking up the name of the command to execute
// in a configuration file.
java.util.Properties config;
String cmd = config.getProperty("sysloadcmd");
if (cmd != null) {
  // Execute the command; Process p represents the running command
  Process p = Runtime.getRuntime().exec(cmd);          // Start the command
  InputStream pin = p.getInputStream();                // Read bytes from it
  InputStreamReader cin = new InputStreamReader(pin); // Convert them to chars
  BufferedReader in = new BufferedReader(cin);         // Read lines of chars
  String load = in.readLine();                         // Get the command output
  in.close();                                          // Close the stream
}
```


Security

The `java.security` package defines quite a few classes related to the Java access-control architecture, which is discussed in more detail in Chapter 5, *Java Security*. These classes allow Java programs to run untrusted code in a restricted environment from which it can do no harm. While these are important classes, you rarely need to use them.

The more interesting classes are the ones used for authentication. A *message digest* is a value, also known as cryptographic checksum or secure hash, that is computed over a sequence of bytes. The length of the digest is typically much smaller than the length of the data for which it is computed, but any change, no matter how small, in the input bytes, produces a change in the digest. When transmitting data (a message), you can transmit a message digest along with it. Then, the recipient of the message can recompute the message digest on the received data and, by comparing the computed digest to the received digest, determine whether the message or the digest was corrupted or tampered with during transmission. We saw a way to compute a message digest earlier in the chapter when we discussed streams. A similar technique can be used to compute a message digest for non-streaming binary data:

```
import java.security.*;

// Obtain an object to compute message digests using the "Secure Hash
// Algorithm"; this method can throw NoSuchAlgorithmException.
MessageDigest md = MessageDigest.getInstance("SHA");
```

```
byte[] data, data1, data2, secret;  // Some byte arrays initialized elsewhere

// Create a digest for a single array of bytes
byte[] digest = md.digest(data);

// Create a digest for several chunks of data
md.reset();           // Optional: automatically called by digest()
md.update(data1);     // Process the first chunk of data
md.update(data2);     // Process the second chunk of data
digest = md.digest(); // Compute the digest

// Create a keyed digest that can be verified if you know the secret bytes
md.update(data);            // The data to be transmitted with the digest
digest = md.digest(secret); // Add the secret bytes and compute the digest

// Verify a digest like this
byte[] receivedData, receivedDigest;  // The data and the digest we received
byte[] verifyDigest = md.digest(receivedData);  // Digest the received data
// Compare computed digest to the received digest
boolean verified =  java.util.Arrays.equals(receivedDigest, verifyDigest);
```

A *digital signature* combines a message-digest algorithm with public-key cryptography. The sender of a message, Alice, can compute a digest for a message and then encrypt that digest with her private key. She then sends the message and the encrypted digest to a recipient, Bob. Bob knows Alice's public key (it is public, after all), so he can use it to decrypt the digest and verify that the message has not been tampered with. In performing this verification, Bob also learns that the digest was encrypted with Alice's private key, since he was able to decrypt the digest successfully using Alice's public key. As Alice is the only one who knows her private key, the message must have come from Alice. A digital signature is called such because, like a pen-and-paper signature, it serves to authenticate the origin of a document or message. Unlike a pen-and-paper signature, however, a digital signature is very difficult, if not impossible, to forge, and it cannot simply be cut and pasted onto another document.

Java makes creating digital signatures easy. In order to create a digital signature, however, you need a `java.security.PrivateKey` object. Assuming that a keystore exists on your system (see the *keytool* documentation in Chapter 8, *Java Development Tools*), you can get one with code like the following:

```
// Here is some basic data we need
File homedir = new File(System.getProperty("user.home"));
File keyfile = new File(homedir, ".keystore"); // Or read from config file
String filepass = "KeyStore password"          // Password for entire file
String signer = "david";                       // Read from config file
String password = "No one can guess this!";    // Better to prompt for this
PrivateKey key;  // This is the key we want to look up from the keystore

try {
  // Obtain a KeyStore object and then load data into it
  KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
  keystore.load(new BufferedInputStream(new FileInputStream(keyfile)),
                filepass.toCharArray());
  // Now ask for the desired key
  key = (PrivateKey) keystore.getKey(signer, password.toCharArray());
}
catch (Exception e) { /* Handle various exception types here */ }
```

Once you have a PrivateKey object, you create a digital signature with a java.security.Signature object:

```
PrivateKey key;              // Initialized as shown previously
byte[] data;                 // The data to be signed
Signature s =                // Obtain object to create and verify signatures
  Signature.getInstance("SHA1withDSA");  // Can throw NoSuchAlgorithmException
s.initSign(key);             // Initialize it; can throw InvalidKeyException
s.update(data);              // Data to sign; can throw SignatureException
/* s.update(data2); */       // Call multiple times to specify all data
byte[] signature = s.sign(); // Compute signature
```

A Signature object can verify a digital signature:

```
byte[] data;        // The signed data; initialized elsewhere
byte[] signature;   // The signature to be verified; initialized elsewhere
String signername;  // Who created the signature; initialized elsewhere
KeyStore keystore;  // Where certificates stored; initialize as shown earlier

// Look for a public key certificate for the signer
java.security.cert.Certificate cert = keystore.getCertificate(signername);
PublicKey publickey = cert.getPublicKey();  // Get the public key from it

Signature s = Signature.getInstance("SHA1withDSA"); // Or some other algorithm
s.initVerify(publickey);                            // Setup for verification
s.update(data);                                     // Specify signed data
boolean verified = s.verify(signature);             // Verify signature data
```

The java.security.SignedObject class is a convenient utility for wrapping a digital signature around an object. The SignedObject can then be serialized and transmitted to a recipient, who can deserialize it and use the verify() method to verify the signature:

```
Serializable o;  // The object to be signed; must be Serializable
PrivateKey k;    // The key to sign with; initialized elsewhere
Signature s = Signature.getInstance("SHA1withDSA"); // Signature "engine"
SignedObject so = new SignedObject(o, k, s);        // Create the SignedObject

// The SignedObject encapsulates the object o; it can now be serialized
// and transmitted to a recipient.

// Here's how the recipient verifies the SignedObject
SignedObject so;           // The deserialized SignedObject
Object o;                  // The original object to extract from it
PublicKey pk;              // The key to verify with
Signature s = Signature.getInstance("SHA1withDSA"); // Verification "engine"
if (so.verify(pk,s))       // If the signature is valid,
  o = so.getObject();      // retrieve the encapsulated object.
```

Cryptography

The java.security package includes cryptography-based classes, but it does not contain classes for actual encryption and decryption. That is the job of the javax.crypto package. This package supports symmetric-key cryptography, in which the same key is used for both encryption and decryption and must be known by both the sender and the receiver of encrypted data. The SecretKey interface represents an encryption key; the first step of any cryptographic

operation is to obtain an appropriate SecretKey. Unfortunately, the *keytool* program supplied with the Java SDK cannot generate and store secret keys, so a program must handle these tasks itself. Here is some code that shows various ways to work with SecretKey objects:

```
import javax.crypto.*;
import javax.crypto.spec.*;

// Generate encryption keys with a KeyGenerator object
KeyGenerator desGen = KeyGenerator.getInstance("DES");       // DES algorithm
SecretKey desKey = desGen.generateKey();                     // Generate a key
KeyGenerator desEdeGen = KeyGenerator.getInstance("DESede"); // Triple DES
SecretKey desEdeKey = desEdeGen.generateKey();               // Generate a key

// SecretKey is an opaque representation of a key. Use SecretKeyFactory to
// convert to a transparent representation that can be manipulated: saved
// to a file, securely transmitted to a receiving party, etc.
SecretKeyFactory desFactory = SecretKeyFactory.getInstance("DES");
DESKeySpec desSpec = (DESKeySpec)
  desFactory.getKeySpec(desKey, javax.crypto.spec.DESKeySpec.class);
byte[] rawDesKey = desSpec.getKey();
// Do the same for a DESede key
SecretKeyFactory desEdeFactory = SecretKeyFactory.getInstance("DESede");
DESedeKeySpec desEdeSpec = (DESedeKeySpec)
  desEdeFactory.getKeySpec(desEdeKey, javax.crypto.spec.DESedeKeySpec.class);
byte[] rawDesEdeKey = desEdeSpec.getKey();

// Convert the raw bytes of a key back to a SecretKey object
DESedeKeySpec keyspec = new DESedeKeySpec(rawDesEdeKey);
SecretKey k = desEdeFactory.generateSecret(keyspec);

// For DES and DESede keys, there is an even easier way to create keys
// SecretKeySpec implements SecretKey, so use it to represent these keys
byte[] desKeyData = new byte[8];        // Read 8 bytes of data from a file
byte[] tripleDesKeyData = new byte[24]; // Read 24 bytes of data from a file
SecretKey myDesKey = new SecretKeySpec(desKeyData, "DES");
SecretKey myTripleDesKey = new SecretKeySpec(tripleDesKeyData, "DESede");
```

Once you have obtained an appropriate SecretKey object, the central class for encryption and decryption is Cipher. Use it like this:

```
SecretKey key;     // Obtain a SecretKey as shown earlier
byte[] plaintext;  // The data to encrypt; initialized elsewhere

// Obtain an object to perform encryption or decryption
Cipher cipher = Cipher.getInstance("DESede");  // Triple-DES encryption
// Initialize the cipher object for encryption
cipher.init(Cipher.ENCRYPT_MODE, key);
// Now encrypt data
byte[] ciphertext = cipher.doFinal(plaintext);

// If we had multiple chunks of data to encrypt, we can do this
cipher.update(message1);
cipher.update(message2);
byte[] ciphertext = cipher.doFinal();

// We simply reverse things to decrypt
cipher.init(Cipher.DECRYPT_MODE, key);
byte[] decryptedMessage = cipher.doFinal(ciphertext);
```

```
// To decrypt multiple chunks of data
byte[] decrypted1 = cipher.update(ciphertext1);
byte[] decrypted2 = cipher.update(ciphertext2);
byte[] decrypted3 = cipher.doFinal(ciphertext3);
```

The `Cipher` class can also be used with `CipherInputStream` or `CipherOutputStream` to encrypt or decrypt while reading or writing streaming data:

```
byte[] data;                              // The data to encrypt
SecretKey key;                            // Initialize as shown earlier
Cipher c = Cipher.getInstance("DESede");  // The object to perform encryption
c.init(Cipher.ENCRYPT_MODE, key);         // Initialize it

// Create a stream to write bytes to a file
FileOutputStream fos = new FileOutputStream("encrypted.data");

// Create a stream that encrypts bytes before sending them to that stream
// See also CipherInputStream to encrypt or decrypt while reading bytes
CipherOutputStream cos = new CipherOutputStream(fos, c);

cos.write(data);                      // Encrypt and write the data to the file
cos.close();                          // Always remember to close streams
java.util.Arrays.fill(data, (byte)0); // Erase the unencrypted data
```

Finally, the `javax.crypto.SealedObject` class provides an especially easy way to perform encryption. This class serializes a specified object and encrypts the resulting stream of bytes. The `SealedObject` can then be serialized itself and transmitted to a recipient. The recipient is only able to retrieve the original object if she knows the required `SecretKey`:

```
Serializable o;          // The object to be encrypted; must be Serializable
SecretKey key;                            // The key to encrypt it with
Cipher c = Cipher.getInstance("Blowfish"); // Object to perform encryption
c.init(Cipher.ENCRYPT_MODE, key);         // Initialize it with the key
SealedObject so = new SealedObject(o, c); // Create the sealed object

// Object so is a wrapper around an encrypted form of the original object o;
// it can now be serialized and transmitted to another party.
// Here's how the recipient decrypts the original object
Object original = so.getObject(key);      // Must use the same SecretKey
```

CHAPTER 5

Java Security

Java programs can dynamically load Java classes from a variety of sources, including untrusted sources, such as web sites reached across an insecure network. The ability to create and work with such mobile code is one of the great strengths and features of Java. To make it work successfully, however, Java puts great emphasis on a security architecture that allows untrusted code to run safely, without fear of damage to the host system.

The need for a security system in Java is most acutely demonstrated by applets—miniature Java applications designed to be embedded in web pages.* When a user visits a web page (with a Java-enabled web browser) that contains an applet, the web browser downloads the Java class files that define that applet and runs them. In the absence of a security system, an applet could wreak havoc on the user's system by deleting files, installing a virus, stealing confidential information, and so on. Somewhat more subtly, an applet could take advantage of the user's system to forge email, generate spam, or launch hacking attempts on other systems.

Java's main line of defense against such malicious code is *access control*: untrusted code is simply not given access to certain sensitive portions of the core Java API. For example, an untrusted applet is not typically allowed to read, write, or delete files on the host system or connect over the network to any computer other than the web server from which it was downloaded. This chapter describes the Java access control architecture and a few other facets of the Java security system.

Security Risks

Java has been designed from the ground up with security in mind; this gives it a great advantage over many other existing systems and platforms. Nevertheless, no system can guarantee 100% security, and Java is no exception.

---

* Applets are documented in *Java Foundation Classes in a Nutshell* (O'Reilly) and are not covered in this book. Still, they serve as good examples here.

The Java security architecture was designed by security experts and has been studied and probed by many other security experts. The consensus is that the architecture itself is strong and robust, theoretically without any security holes (at least none that have been discovered yet). The implementation of the security architecture is another matter, however, and there is a long history of security flaws being found and patched in particular implementations of Java. For example, in April 1999, a flaw was found in Sun's implementation of the class verifier in Java 1.1. Patches for Java 1.1.6 and 1.1.7 were issued and the problem was fixed in Java 1.1.8. Even more recently, in August 1999, a severe flaw was found in Microsoft's Java Virtual Machine (which is used by the Internet Explorer 4.0 and 5.0 web browsers). The flaw was a particularly dangerous one because it allowed a malicious applet to gain unrestricted access to the underlying system. Microsoft has released a new version of their VM, and (as of this writing) there have not been any known attacks that took advantage of the flaw.

In all likelihood, security flaws will continue to be discovered (and patched) in Java VM implementations. Despite this, Java remains perhaps the most secure platform currently available. There have been few, if any, reported instances of malicious Java code exploiting security holes "in the wild." For practical purposes, the Java platform appears to be adequately secure, especially when contrasted with some of the insecure and virus-ridden alternatives.

Java VM Security and Class File Verification

The lowest level of the Java security architecture involves the design of the Java Virtual Machine and the byte codes it executes. The Java VM does not allow any kind of direct access to individual memory addresses of the underlying system, which prevents Java code from interfering with the native hardware and operating system. These intentional restrictions on the VM are reflected in the Java language itself, which does not support pointers or pointer arithmetic. The language does not allow an integer to be cast to an object reference or vice versa, and there is no way whatsoever to obtain an object's address in memory. Without capabilities like these, malicious code simply cannot gain a foothold.

In addition to the secure design of the Virtual Machine instruction set, the VM goes through a process known as *byte-code verification* whenever it loads an untrusted class. This process ensures that the byte codes of a class (and their operands) are all valid; that the code never underflows or overflows the VM stack; that local variables are not used before they are initialized; that field, method, and class access control modifiers are respected; and so on. The verification step is designed to prevent the VM from executing byte codes that might crash it or put it into an undefined and untested state where it might be vulnerable to other attacks by malicious code. Byte-code verification is a defense against malicious hand-crafted Java byte codes and untrusted Java compilers that might output invalid byte codes.

Authentication and Cryptography

In Java 1.1 and later, the `java.security` package (and its subpackages) provides classes and interfaces for *authentication*. As described in Chapter 4, *The Java Platform*, this piece of the security architecture allows Java code to create and verify message digests and digital signatures. These technologies can ensure that any data (such as a Java class file) is authentic; that it originates from the person who claims to have originated it and has not been accidentally or maliciously modified in transit.

The Java Cryptography Extension, or JCE, consists of the `javax.crypto` package and its subpackages. These packages define classes for encryption and decryption of data. This is an important security-related feature for many applications, but is not directly relevant to the basic problem of preventing untrusted code from damaging the host system, so it is not discussed in this chapter.

Access Control

As I noted at the beginning of this chapter, the heart of the Java security architecture is access control: untrusted code simply must not be granted access to the sensitive parts of the Java API that would allow it to do malicious things. As we'll discuss in the following sections, the Java access-control model evolved significantly between Java 1.0 and Java 1.2. The Java 1.2 access-control model is relatively stable; it has not changed significantly in Java 1.3.

Java 1.0: The Sandbox

In this first release of Java, all Java code installed locally on the system is trusted implicitly. All code downloaded over the network, however, is untrusted and run in a restricted environment playfully called "the sandbox." The access-control policies of the sandbox are defined by the currently installed `java.lang.SecurityManager` object. When system code is about to perform a restricted operation, such as reading a file from the local filesystem, it first calls an appropriate method (such as `checkRead()`) of the currently installed `SecurityManager` object. If untrusted code is running, the `SecurityManager` throws a `SecurityException` that prevents the restricted operation from taking place.

The most common user of the `SecurityManager` class is a Java-enabled web browser, which installs a `SecurityManager` object to allow applets to run without damaging the host system. The precise details of the security policy are an implementation detail of the web browser, of course, but applets are typically restricted in the following ways:

- An applet cannot read, write, rename, or delete files. It cannot query the length or modification date of a file or even check whether a given file exists. Similarly, an applet cannot create, list, or delete a directory.
- An applet cannot connect to or accept a connection from any computer other than the one it was downloaded from. It cannot use any privileged ports (i.e., ports below and including port 1024).

- An applet cannot perform system-level functions, such as loading a native library, spawning a new process, or exiting the Java interpreter. An applet cannot manipulate any threads or thread groups, except for those it creates itself. In Java 1.1 and later, applets cannot use the Java Reflection API to obtain information about the non-public members of classes, except for classes that were downloaded with the applet.
- An applet cannot access certain graphics- and GUI-related facilities. It cannot initiate a print job or access the system clipboard or event queue. In addition, all windows created by an applet typically display a prominent visual indicator that they are "insecure," to prevent an applet from spoofing the appearance of some other application.
- An applet cannot read certain system properties, notably the `user.home` and `user.dir` properties, that specify the user's home directory and current working directory.
- An applet cannot circumvent these security restrictions by registering a new `SecurityManager` object.

How the sandbox works

Suppose that an applet (or some other untrusted code running in the sandbox) attempts to read the contents of the file */etc/passwd* by passing this filename to the `FileInputStream()` constructor. The programmers who wrote the `FileInputStream` class were aware that the class provides access to a system resource (a file), so use of the class should therefore be subject to access control. For this reason, they coded the `FileInputStream()` constructor to use the `SecurityManager` class.

Every time `FileInputStream()` is called, it checks to see if a `SecurityManager` object has been installed. If so, the constructor calls the `checkRead()` method of that `SecurityManager` object, passing the filename (*/etc/passwd*, in this case) as the sole argument. The `checkRead()` method has no return value; it either returns normally or throws a `SecurityException`. If the method returns, the `FileInputStream()` constructor simply proceeds with whatever initialization is necessary and returns. Otherwise, it allows the `SecurityException` to propagate to the caller. When this happens, no `FileInputStream` object is created, and the applet does not gain access to the */etc/passwd* file.

Java 1.1: Digitally Signed Classes

Java 1.1 retains the sandbox model of Java 1.0, but adds the `java.security` package and its digital signature capabilities. With these capabilities, Java classes can be digitally signed and verified. Thus, web browsers and other Java installations can be configured to trust downloaded code that bears a valid digital signature of a trusted entity. Such code is treated as if it were installed locally, so it is given full access to the Java APIs. In this release, the *javakey* program manages keys and digitally signs JAR files of Java code. Although Java 1.1 adds the important ability to trust digitally signed code that would otherwise be untrusted, it sticks to the

basic sandbox model: trusted code gets full access and untrusted code gets totally restricted access.

Java 1.2: Permissions and Policies

Java 1.2 introduces major new access-control features into the Java security architecture. These features are implemented by new classes in the `java.security` package. The `Policy` class is one of the most important: it defines a Java security policy. A `Policy` object maps `CodeSource` objects to associated sets of `Permission` objects. A `CodeSource` object represents the source of a piece of Java code, which includes both the URL of the class file (and can be a local file) and a list of entities that have applied their digital signatures to the class file. The `Permission` objects associated with a `CodeSource` in the `Policy` define the permissions that are granted to code from a given source. Various Java APIs includes subclasses of `Permission` that represent different types of permissions. These include `java.lang.RuntimePermission`, `java.io.FilePermission`, and `java.net.SocketPermission`, for example.

Under this new access-control model, the `SecurityManager` class continues to be the central class; access-control requests are still made by invoking methods of a `SecurityManager`. However, the default `SecurityManager` implementation now delegates most of those requests to a new `AccessController` class that makes access decisions based on the `Permission` and `Policy` architecture.

The new Java 1.2 access-control architecture has several important features:

- Code from different sources can be given different sets of permissions. In other words, the new architecture supports fine-grained levels of trust. Even locally installed code can be treated as untrusted or partially untrusted. Under this new architecture, only system classes and standard extensions run as fully trusted.
- It is no longer necessary to define a custom subclass of `SecurityManager` to define a security policy. Policies can be configured by a system administrator by editing a text file or using the new *policytool* program.
- The new architecture is not limited to a fixed set of access control methods in the `SecurityManager` class. New `Permission` subclasses can be defined easily to govern access to new system resources (which might be exposed, for example, by new standard extensions that include native code).

How policies and permissions work

Let's return to the example of an applet that attempts to create a `FileInputStream` to read the file */etc/passwd*. In Java 1.2, the `FileInputStream()` constructor behaves exactly the same as it does in Java 1.0 and Java 1.1: it looks to see if a `SecurityManager` is installed and, if so, calls its `checkRead()` method, passing the name of the file to be read.

What's new in Java 1.2 is the default behavior of the `checkRead()` method. Unless a program has replaced the default security manager with one of its own, the default implementation creates a `FilePermission` object to represent the access

being requested. This FilePermission object has a *target* of "/etc/passwd" and an *action* of "read". The checkRead() method passes this FilePermission object to the static checkPermission() method of the java.security.AccessController class.

It is the AccessController and its checkPermission() method that do the real work of access control in Java 1.2. The method determines the CodeSource of each calling method and uses the current Policy object to determine the Permission objects associated with it. With this information, the AccessController can determine whether read access to the */etc/passwd* file should be allowed.

The Permission class represents both the permissions granted by a Policy and the permissions requested by a method like the FileInputStream() constructor. When requesting a permission, Java typically uses a FilePermission (or other Permission subclass) with a very specific target, like "/etc/passwd". When granting a permission, however, a Policy commonly uses a FilePermission object with a wildcard target, such as "/etc/*", to represent many files. One of the key features of a Permission subclass such as FilePermission is that it defines an implies() method that can determine whether permission to read "/etc/*" implies permission to read "/etc/passwd".

Security for Everyone

Programmers, system administrators, and end users all have different security concerns and, thus, different roles to play in the Java 1.2 security architecture.

Security for System Programmers

System programmers are the people who define new Java APIs that allow access to sensitive system resources. These programmers are typically working with native methods that have unprotected access to the system. They need to use the Java access-control architecture to prevent untrusted code from executing those native methods. To do this, system programmers must carefully insert SecurityManager calls at appropriate places in their code. A system programmer may choose to use an existing Permission subclass to govern access to the system resources exposed by her API, or she may decide to define a specialized subclass of Permission.

The system programmer carries a tremendous security burden: if she does not perform appropriate access control checks in her code, she compromises the security of the entire Java platform. The details are complex and are beyond the scope of this book. Fortunately, however, system programming that involves native methods is rare in Java; almost all of us are application programmers who can simply rely on the existing APIs.

Security for Application Programmers

Programmers who use the core Java APIs and standard extensions, but do not define new extensions or write native methods, can simply rely on the security efforts of the system programmers who created those APIs. In other words, most

of us Java programmers can simply use the Java APIs and need not worry about introducing security holes into the Java platform.

In fact, application programmers rarely have to use the access-control architecture. If you are writing Java code that may be run as untrusted code, you should be aware of the restrictions placed on untrusted code by typical security policies. Keep in mind that some methods (such as methods that read or write files) can throw `SecurityException` objects, but don't feel you must write your code to catch these exceptions. Often, the appropriate response to a `SecurityException` is to allow it to propagate uncaught, so that it terminates the application.

Sometimes, as an application programmer, you want to write an application (such as an applet viewer) that can load untrusted classes and run them subject to access-control checks. To do this in Java 1.2, you must first install a security manager:

```
System.setSecurityManager(new SecurityManager());
```

Then use `java.net.URLClassLoader` to load the untrusted classes. `URLClassLoader` assigns a default set of safe permissions to the classes it loads, but in some cases you may want to modify the permissions granted to the loaded code through the `Policy` and `PermissionCollection` classes.

Security for System Administrators

In Java 1.2 and later, system administrators are responsible for defining the default security policy for the computers at their site. The default policy is stored in the file *lib/security/java.policy* in the Java installation. A system administrator can edit this text file by hand or use the *policytool* program from Sun to edit the file graphically. *policytool* is the preferred way to define policies, so the syntax of the underlying policy file is not documented in this book.

The default *java.policy* file defines a policy that is much like the policy of Java 1.0 and Java 1.1: system classes and installed extensions are fully trusted, while all other code is untrusted and only allowed a few simple permissions. While this default policy is adequate for many purposes, it may not be appropriate for all sites. For example, at some organizations, it may be appropriate to grant extra permissions to code downloaded from a secure intranet.

In order to define secure and effective security policies, a system administrator must understand the various `Permission` subclasses of the Java platform, the target and action names they support, and the security implications of granting any particular permission. These topics are explained well in a document titled "Permissions in the Java 2 SDK," which is part of the Java 1.2 release and also available (at the time of this writing) online at: *http://java.sun.com/products/jdk/1.2/docs/guide/security/permissions.html.*

Security for End Users

Most end users do not have to think about security at all: their Java programs should simply run in a secure way with no intervention by them. Some sophisticated end users may want to define their own security policies, however. An end

user can do this by running *policytool* himself to define personal policy files that augment the system policy. The default personal policy is stored in a file named *.java.policy* in the user's home directory. By default, Java loads this policy file and uses it to augment the system policy file.

In Java 1.2 and later, a user can specify an additional policy file to use when starting up the Java interpreter, by defining the `java.security.policy` property with the `-D` option. For example:

```
C:\> java -Djava.security.policy=policyfile UntrustedApp
```

This line runs the class `UntrustedApp` after augmenting the default system and user policies with the policy specified in the file or URL *`policyfile`*. To replace the system and user policies instead of augmenting them, use a double equals sign in the property specification:

```
C:\> java -Djava.security.policy==policyfile UntrustedApp
```

Note, however, that specifying a policy file is only useful if there is a `SecurityManager` installed. If a user doesn't trust an application, he presumably doesn't trust that application to voluntarily install its own security manager. In this case, he can define the `java.security.manager` system property:

```
C:\> java -Djava.security.manager -Djava.security.policy=policyfile UntrustedApp
```

The value of this property does not matter; simply defining it is enough to tell the Java interpreter to automatically install a default `SecurityManager` object that subjects an application to the access control policies described in the system, user, and `java.security.policy` policy files.

Permission Classes

Table 5-1 lists the various `Permission` subclasses, the target and action names they support, and the methods that require those permissions (in Java 1.2 and later).

Table 5-1: Permission Classes and the Methods They Govern

Permission	*Target, Action*	*Methods*
`AWT-Permission`	"accessClipboard"	`Toolkit.getSystemClipboard()`
	"accessEventQueue"	`Toolkit.getSystemEventQueue()`
	"listenToAllAWTEvents"	`Toolkit.{addAWTEventListener(), removeAWTEventListener()}`
	"readDisplayPixels"	`Graphics2D.setComposite()`
	"showWindowWithoutWarningBanner"	`Window.Window()` (if permission is not granted, window has an "insecure" indication)
`File-Permission`	*command*, "execute"	`Runtime.exec()`
	filename, "delete"	`File.{delete(), deleteOnExit()}`

Table 5–1: Permission Classes and the Methods They Govern (continued)

Permission	*Target, Action*	*Methods*
FilePermission	*filename*, "read"	FileInputStream.FileInputStream(), File.{exists(), canRead(), isFile(), isDirectory(), isHidden(), lastModified(), length(), list(), listFiles()}, RandomAccessFile.RandomAccessFile(), ZipFile.ZipFile()
	filename, "write"	FileOutputStream.FileOutputStream(), File.{canWrite(), createNewFile(), createTempFile(), mkdir(), mkdirs(), renameTo(), setLastModified(), setReadOnly()}, RandomAccessFile.RandomAccessFile()
NetPermission	"requestPasswordAuthentication"	Authenticator.requestPasswordAuthentication()
	"setDefaultAuthenticator"	Authenticator.setDefault()
	"specifyStreamHandler"	URL.URL()
PropertyPermission	"*", "read, write"	Beans.{setDesignTime(), setGuiAvailable()}, Introspector.setBeanInfoSearchPath(), PropertyEditorManager.{registerEditor(), setEditorSearchPath()}, System.{getProperties(), setProperties()}
	"user.language", "write"	Locale.setDefault()
	prop, "read"	System.getProperty()
	prop, "write"	System.setProperty()
ReflectPermission	"suppressAccessChecks"	AccessibleObject.setAccessible()
RuntimePermission	"accessClassInPackage.*pkgname*"	Class.{getClasses(), getDeclaredClasses(), getConstructor(), getConstructors(), getDeclaredFields(), getDeclaredMethods(), getDeclaredConstructors(), getDeclaredField(), getDeclaredMethod(), getDeclaredConstructor(),

Table 5-1: Permission Classes and the Methods They Govern (continued)

Permission	*Target, Action*	*Methods*
RuntimePermission		getFields(), getMethods(), getField(), getMethod()}
	"accessDeclaredMembers"	Class.{getClasses(), getDeclaredClasses(), getDeclaredFields(), getDeclaredMethods(), getDeclaredConstructors(), getDeclaredField(), getDeclaredMethod(), getDeclaredConstructor()}
	"createClassLoader"	ClassLoader.Class-Loader(), URLClassLoader.URL-ClassLoader(), SecureClassLoader.Secure-ClassLoader()
	"exitVM"	Runtime.{exit(), runFinalizersOnExit()}, System.{exit(), runFinalizersOnExit()}
	"getClassLoader"	Class.{forName(), getClassLoader()}, ClassLoader.{getSystemClassLoader(), getParent()}, Thread.getContextClassLoader()
	"getProtectionDomain"	Class.getProtectionDomain()
	"loadLibrary.*libName*"	Runtime.{load(), loadLibrary()}, System.{load(), loadLibrary()}
	"modifyThread"	Thread.{checkAccess(), interrupt(), suspend(), resume(), setPriority(), setName(), setDaemon()}, ThreadGroup.{interrupt(), stop()}
	"modifyThreadGroup"	Thread.{Thread(), enumerate()}, ThreadGroup.{ThreadGroup(), enumerate(), getParent(), interrupt(), setDaemon(), setMaxPriority(), stop(), suspend(), resume(), destroy()}
	"queuePrintJob"	Toolkit.getPrintJob()
	"readFileDescriptor"	FileInputStream.File-InputStream(FileDescriptor)
	"setContextClassLoader"	Thread.setContextClassLoader()

Table 5–1: Permission Classes and the Methods They Govern (continued)

Permission	*Target, Action*	*Methods*
RuntimePermission	"setFactory"	ServerSocket.setSocketFactory(), Socket.setSocketImplFactory(), URL.setURLStream-HandlerFactory(), URLConnection.{setContent-HandlerFactory(), setFileNameMap()}, HttpURLConnection.set-FollowRedirects(), activation.Activation-Group.{createGroup(), setSystem()}, server.RMISocketFactory.set-SocketFactory()
	"setIO"	System.{setIn(), setOut(), setErr()}
	"setSecurityManager"	System.setSecurityManager()
	"stopThread"	Thread.stop(), ThreadGroup.stop()
	"writeFileDescriptor"	FileOutputStream.File-OutputStream(FileDescriptor)
SecurityPermission	"addIdentityCertificate"	Identity.addCertificate()
	"clearProvider-Properties.*provider*"	Provider.clear()
	"getPolicy"	Policy.getPolicy()
	"getProperty.*propname*"	Security.getProperty()
	"getSignerPrivateKey"	Signer.getPrivateKey()
	"insertProvider.*provider*"	Security.{addProvider(), insertProviderAt()}
	"printIdentity"	Identity.toString()
	"putProvider-Property.*provider*"	Provider.put()
	"removeIdentityCertificate"	Identity.removeCertificate()
	"removeProvider.*provider*"	Security.removeProvider()
	"removeProvider-Property.*provider*"	Provider.remove()
	"setIdentityInfo"	Identity.setInfo(String)
	"setIdentityPublicKey"	Identity.setPublicKey()
	"setPolicy"	Policy.setPolicy();
	"setProperty.*propname*"	Security.setProperty()
	"setSignerKeypair"	Signer.setKeyPair()
	"setSystemScope"	IdentityScope.setSystemScope()

Table 5–1: Permission Classes and the Methods They Govern (continued)

Permission	*Target, Action*	*Methods*
`Serializable-Permission`	"enableSubclass-Implementation"	`ObjectInputStream.Object-InputStream()`, `ObjectOutputStream.Object-OutputStream()`
	"enableSubstitution"	`ObjectInputStream.enable-ResolveObject()`, `ObjectOutputStream.enable-ReplaceObject()`
`Socket-Permission`	"localhost:*port*", "listen"	`ServerSocket.ServerSocket()`, `DatagramSocket.DatagramSocket()`, `MulticastSocket.MulticastSocket()`
	host, "accept, connect"	`MulticastSocket.{joinGroup(), leaveGroup(), send()}`
	host, "resolve"	`InetAddress.{getHostName(), getAllByName(), getLocalHost()}`, `DatagramSocket.getLocalAddress()`
	host:port, "accept"	`DatagramSocket.receive()`, `ServerSocket.{accept(), implAccept()}`
	host:port, "connect"	`DatagramSocket.send()`, `Socket.Socket()`

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

About the Author

David Flanagan is a computer programmer who spends most of his time writing about Java. His other books with O'Reilly & Associates include *Java Foundation Classes in a Nutshell, Java Enterprise in a Nutshell, JavaScript: The Definitive Guide, Java Examples in a Nutshell, Java Power Reference,* and *JavaScript Pocket Reference.* David has a degree in computer science and engineering from the Massachusetts Institute of Technology. He lives with his partner Christie in the U.S. Pacific Northwest between the cities of Seattle, Washington and Vancouver, British Columbia.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal appearing on the cover of *Java in a Nutshell, Third Edition,* is a Javan tiger. It is the smallest of the eight subspecies of tiger, and has the longest cheek whiskers, forming a short mane across the neck. The encroachment of the growing human population, along with increases in poaching, led to the near-extinction of the Javan tiger. The Indonesian government has become involved in trying to preserve the tiger. It is to be hoped that the remaining subspecies of tiger will be helped by increasing awareness and stricter protections.

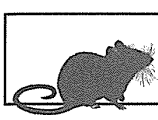
Tigers are the largest of all cats, weighing up to 660 pounds and with a body length of up to 9 feet. They are solitary animals, and, unlike lions, hunt alone. Tigers prefer large prey, such as wild pigs, cattle, or deer. Tigers rarely attack humans, although attacks on humans have increased as the increasing human population more frequently comes into contact with tigers. Tiger attacks usually occur when the tiger feels that it or its young are being threatened. In such cases, the tiger almost never eats its human victim. There are some tigers, however, who have developed a taste for human flesh. This is a particularly bad problem in an area of India and Bangladesh called the Sunderbans.

Mary Anne Weeks Mayo was the production editor and copyeditor for *Java in a Nutshell, Third Edition*; Ellie Cutler, Maureen Dempsey, and Jane Ellin provided quality control, and Ellie Fountain Maden proofread the book. Anna Kim Snow provided production assistance. Lenny Muellner and Chris Maden provided SGML support. Ellen Troutman Zaig and Brenda Miller wrote the index.

Edie Freedman designed the cover of this book, using a 19th-century engraving from the Dover Pictorial Archive. Whenever possible, our books use RepKover™, a durable and flexible lay-flat binding. If the page count exceeds RepKover's limit, perfect binding is used.

Kathleen Wilson produced the cover layout with Quark XPress 3.3 using Adobe's ITC Garamond font. The interior layouts were designed by Edie Freedman and Nancy Priest, with modifications by Alicia Cech, and Lenny Muellner implemented the layout in *gtroff.* Interior fonts are Adobe ITC Garamond and Adobe ITC Franklin

Gothic. The illustrations that appear in the book were produced by Robert Romano and Rhon Porter using Macromedia FreeHand 8 and Adobe Photoshop 5. This colophon was written by Clairemarie Fisher O'Leary.



Java

Java Servlet Programming

By Jason Hunter with William Crawford
1st Edition November 1998
528 pages, ISBN 1-56592-391-X

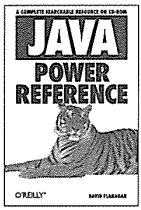
Java servlets offer a fast, powerful, portable replacement for CGI scripts. *Java Servlet Programming* covers everything you need to know to write effective servlets. Topics include: serving dynamic Web content, maintaining state information, session tracking, database connectivity using JDBC, and applet-servlet communication.

Java Swing

By Robert Eckstein, Marc Loy & Dave Wood
1st Edition September 1998
1252 pages, ISBN 1-56592-455-X

The Swing classes eliminate Java's biggest weakness: its relatively primitive user interface toolkit. *Java Swing* helps you to take full advantage of the Swing classes, providing detailed descriptions of every class and interface in the key Swing packages. It shows you how to use all of the new components, allowing you to build state-of-the-art user interfaces and giving you the context you need to understand what you're doing. It's more than documentation; *Java Swing* helps you develop code quickly and effectively.

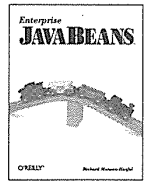
Java Power Reference



By David Flanagan
1st Edition March 1999
64 pages, Features CD-ROM
ISBN 1-56592-589-0

Java Power Reference is a searchable, browser-based resource that documents all the packages and classes of the Java 2(TM) platform on a single CD-ROM. Based on the clear, concise quick-reference style of the bestselling *Java in a Nutshell*, the *Java Power Reference* provides a unique view of the functionality of the Java APIs. In addition to the CD-ROM, the package contains a concise printed overview of the newly released Java 2 platform.

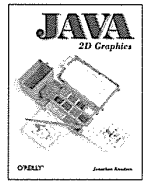
Enterprise JavaBeans



By Richard Monson-Haefel
1st Edition June 1999
336 pages, ISBN 1-56592-605-6

Enterprise JavaBeans is a thorough introduction to EJB for the enterprise software developer. It shows how to get started developing enterprise Beans, how to deploy those Beans in a server, and how to use those Beans to create applications that do useful tasks. The end result is a highly flexible system built from components that can easily be reused and that can be changed to suit your needs without upsetting other parts of the system.

Java 2D Graphics



By Jonathan Knudsen
1st Edition May 1999
366 pages, ISBN 1-56592-484-3

Java 2D Graphics describes the 2D API from top to bottom, demonstrating how to set line styles and pattern fills as well as more advanced techniques of image processing and font handling. You'll see how to create and manipulate the three types of graphics objects: shapes, text, and images. Other topics include image data storage, color management, font glyphs, and printing.

Developing Java Beans

By Robert Englander
1st Edition June 1997
316 pages, ISBN 1-56592-289-1

Developing Java Beans is a complete introduction to Java's component architecture. It describes how to write Beans, which are software components that can be used in visual programming environments. This book discusses event adapters, serialization, introspection, property editors, and customizers, and shows how to use Beans within ActiveX controls.

How to stay in touch with O'Reilly

1. Visit Our Award-Winning Site

http://www.oreilly.com/

★ "Top 100 Sites on the Web" —*PC Magazine*
★ "Top 5% Web sites" —*Point Communications*
★ "3-Star site" —*The McKinley Group*

Our web site contains a library of comprehensive product information (including book excerpts and tables of contents), downloadable software, background articles, interviews with technology leaders, links to relevant sites, book cover art, and more. File us in your Bookmarks or Hotlist!

2. Join Our Email Mailing Lists

New Product Releases

To receive automatic email with brief descriptions of all new O'Reilly products as they are released, send email to:
listproc@online.oreilly.com
Put the following information in the first line of your message (*not* in the Subject field):
subscribe oreilly-news

O'Reilly Events

If you'd also like us to send information about trade show events, special promotions, and other O'Reilly events, send email to:
listproc@online.oreilly.com
Put the following information in the first line of your message (*not* in the Subject field):
subscribe oreilly-events

3. Get Examples from Our Books via FTP

There are two ways to access an archive of example files from our books:

Regular FTP

- ftp to:
  ftp.oreilly.com
  (login: anonymous
  password: your email address)
- Point your web browser to:
  ftp://ftp.oreilly.com/

FTPMAIL

- Send an email message to:
  ftpmail@online.oreilly.com
  (Write "help" in the message body)

4. Contact Us via Email

order@oreilly.com
To place a book or software order online. Good for North American and international customers.

subscriptions@oreilly.com
To place an order for any of our newsletters or periodicals.

books@oreilly.com
General questions about any of our books.

software@oreilly.com
For general questions and product information about our software. Check out O'Reilly Software Online at **http://software.oreilly.com/** for software and technical support information. Registered O'Reilly software users send your questions to: **website-support@oreilly.com**

cs@oreilly.com
For answers to problems regarding your order or our products.

booktech@oreilly.com
For book content technical questions or corrections.

proposals@oreilly.com
To submit new book or software proposals to our editors and product managers.

international@oreilly.com
For information about our international distributors or translation queries. For a list of our distributors outside of North America check out:
http://www.oreilly.com/www/order/country.html

O'Reilly & Associates, Inc.
101 Morris Street, Sebastopol, CA 95472 USA
TEL 707-829-0515 or 800-998-9938
(6am to 5pm PST)
FAX 707-829-0104

Titles from O'Reilly

Web
Advanced Perl Programming
Apache: The Definitive Guide, 2nd Ed.
ASP in a Nutshell
Building Your Own Web Conferences
Building Your Own Website™
CGI Programming with Perl
Designing with JavaScript
Dynamic HTML: The Definitive Reference
Frontier: The Definitive Guide
HTML: The Definitive Guide, 3rd Ed.
Information Architecture for the World Wide Web
JavaScript Pocket Reference
JavaScript: The Definitive Guide, 3rd Ed.
Learning VB Script
Photoshop for the Web
WebMaster in a Nutshell
WebMaster in a Nutshell, Deluxe Ed.
Web Design in a Nutshell
Web Navigation: Designing the User Experience
Web Performance Tuning
Web Security & Commerce
Writing Apache Modules

Perl
Learning Perl, 2nd Ed.
Learning Perl for Win32 Systems
Learning Perl/TK
Mastering Algorithms with Perl
Mastering Regular Expressions
Perl5 Pocket Reference, 2nd Ed.
Perl Cookbook
Perl in a Nutshell
Perl Resource Kit—UNIX Ed.
Perl Resource Kit—Win32 Ed.
Perl/TK Pocket Reference
Programming Perl, 2nd Ed.
Web Client Programming with Perl

Graphics & Multimedia
Director in a Nutshell
Encyclopedia of Graphics File Formats, 2nd Ed.
Lingo in a Nutshell
Photoshop in a Nutshell
QuarkXPress in a Nutshell

Using the Internet
AOL in a Nutshell
Internet in a Nutshell
Smileys
The Whole Internet for Windows95
The Whole Internet: The Next Generation
The Whole Internet User's Guide & Catalog

Java Series
Database Programming with JDBC and Java
Developing Java Beans
Exploring Java, 2nd Ed.
Java AWT Reference
Java Cryptography
Java Distributed Computing
Java Examples in a Nutshell
Java Foundation Classes in a Nutshell
Java Fundamental Classes Reference
Java in a Nutshell, 2nd Ed.
Java in a Nutshell, Deluxe Ed.
Java I/O
Java Language Reference, 2nd Ed.
Java Media Players
Java Native Methods
Java Network Programming
Java Security
Java Servlet Programming
Java Swing
Java Threads
Java Virtual Machine

Unix
Exploring Expect
GNU Emacs Pocket Reference
Learning GNU Emacs, 2nd Ed.
Learning the bash Shell, 2nd Ed.
Learning the Korn Shell
Learning the UNIX Operating System, 4th Ed.
Learning the vi Editor, 6th Ed.
Linux in a Nutshell
Linux Multimedia Guide
Running Linux, 2nd Ed.
SCO UNIX in a Nutshell
sed & awk, 2nd Ed.
Tcl/Tk in a Nutshell
Tcl/Tk Pocket Reference
Tcl/Tk Tools
The UNIX CD Bookshelf
UNIX in a Nutshell, System V Ed.
UNIX Power Tools, 2nd Ed.
Using csh & tsch
Using Samba
vi Editor Pocket Reference
What You Need To Know: When You Can't Find Your UNIX System Administrator
Writing GNU Emacs Extensions

Songline Guides
NetLaw
NetResearch
NetLearning
NetSuccess
NetLessons
NetTravel

Software
Building Your Own WebSite™
Building Your Own Web Conference
WebBoard™ 3.0
WebSite Professional™ 2.0
PolyForm™

System Administration
Building Internet Firewalls
Computer Security Basics
Cracking DES
DNS and BIND, 3rd Ed.
DNS on WindowsNT
Essential System Administration
Essential WindowsNT System Administration
Getting Connected: The Internet at 56K and Up
Linux Network Administrator's Guide
Managing IP Networks with Cisco Routers
Managing Mailing Lists
Managing NFS and NIS
Managing the WindowsNT Registry
Managing Usenet
MCSE: The Core Exams in a Nutshell
MCSE: The Electives in a Nutshell
Networking Personal Computers with TCP/IP
Oracle Performance Tuning, 2nd Ed.
Practical UNIX & Internet Security, 2nd Ed.
PGP: Pretty Good Privacy
Protecting Networks with SATAN
sendmail, 2nd Ed.
sendmail Desktop Reference
System Performance Tuning
TCP/IP Network Administration, 2nd Ed.
termcap & terminfo
The Networking CD Bookshelf
Using & Managing PPP
Virtual Private Networks
WindowsNT Backup & Restore
WindowsNT Desktop Reference
WindowsNT Event Logging
WindowsNT in a Nutshell
WindowsNT Server 4.0 for Netware Administrators
WindowsNT SNMP
WindowsNT TCP/IP Administration
WindowsNT User Administration
Zero Administration for Windows

X Window
Vol. 1: Xlib Programming Manual
Vol. 2: Xlib Reference Manual
Vol. 3M: X Window System User's Guide, Motif Ed.
Vol. 4M: X Toolkit Intrinsics Programming Manual, Motif Ed.
Vol. 5: X Toolkit Intrinsics Reference Manual
Vol. 6A: Motif Programming Manual
Vol. 6B: Motif Reference Manual
Vol. 8 : X Window System Administrator's Guide

Programming
Access Database Design and Programming
Advanced Oracle PL/SQL Programming with Packages
Applying RCS and SCCS
BE Developer's Guide
BE Advanced Topics
C++: The Core Language
Checking C Programs with lint
Developing Windows Error Messages
Developing Visual Basic Add-ins
Guide to Writing DCE Applications
High Performance Computing, 2nd Ed.
Inside the Windows 95 File System
Inside the Windows 95 Registry
lex & yacc, 2nd Ed.
Linux Device Drivers
Managing Projects with make
Oracle8 Design Tips
Oracle Built-in Packages
Oracle Design
Oracle PL/SQL Programming, 2nd Ed.
Oracle Scripts
Oracle Security
Palm Programming: The Developer's Guide
Porting UNIX Software
POSIX Programmer's Guide
POSIX.4: Programming for the Real World
Power Programming with RPC
Practical C Programming, 3rd Ed.
Practical C++ Programming
Programming Python
Programming with curses
Programming with GNU Software
Pthreads Programming
Python Pocket Reference
Software Portability with imake, 2nd Ed.
UML in a Nutshell
Understanding DCE
UNIX Systems Programming for SVR4
VB/VBA in a Nutshell: The Languages
Win32 Multithreaded Programming
Windows NT File System Internals
Year 2000 in a Nutshell

Using Windows
Excel97 Annoyances
Office97 Annoyances
Outlook Annoyances
Windows Annoyances
Windows98 Annoyances
Windows95 in a Nutshell
Windows98 in a Nutshell
Word97 Annoyances

Other Titles
PalmPilot: The Ultimate Guide
Palm Programming: The Developer's Guide

International Distributors

UK, Europe, Middle East and Africa (except France, Germany, Austria, Switzerland, Luxembourg, Liechtenstein, and Eastern Europe)

INQUIRIES
O'Reilly UK Limited
4 Castle Street
Farnham
Surrey, GU9 7HS
United Kingdom
Telephone: 44-1252-711776
Fax: 44-1252-734211
Email: josette@oreilly.com

ORDERS
Wiley Distribution Services Ltd.
1 Oldlands Way
Bognor Regis
West Sussex PO22 9SA
United Kingdom
Telephone: 44-1243-779777
Fax: 44-1243-820250
Email: cs-books@wiley.co.uk

France

ORDERS
GEODIF
61, Bd Saint-Germain
75240 Paris Cedex 05, France
Tel: 33-1-44-41-46-16 (French books)
Tel: 33-1-44-41-11-87 (English books)
Fax: 33-1-44-41-11-44
Email: distribution@eyrolles.com

INQUIRIES
Éditions O'Reilly
18 rue Séguier
75006 Paris, France
Tel: 33-1-40-51-52-30
Fax: 33-1-40-51-52-31
Email: france@editions-oreilly.fr

Germany, Switzerland, Austria, Eastern Europe, Luxembourg, and Liechtenstein

INQUIRIES & ORDERS
O'Reilly Verlag
Balthasarstr. 81
D-50670 Köln
Germany
Telephone: 49-221-973160-91
Fax: 49-221-973160-8
Email: anfragen@oreilly.de (inquiries)
Email: order@oreilly.de (orders)

Canada (French language books)

Les Éditions Flammarion ltée
375, Avenue Laurier Ouest
Montréal (Québec) H2V 2K3
Tel: 00-1-514-277-8807
Fax: 00-1-514-278-2085
Email: info@flammarion.qc.ca

Hong Kong

City Discount Subscription Service, Ltd.
Unit D, 3rd Floor, Yan's Tower
27 Wong Chuk Hang Road
Aberdeen, Hong Kong
Tel: 852-2580-3539
Fax: 852-2580-6463
Email: citydis@ppn.com.hk

Korea

Hanbit Media, Inc.
Sonyoung Bldg. 202
Yeksam-dong 736-36
Kangnam-ku
Seoul, Korea
Tel: 822-554-9610
Fax: 822-556-0363
Email: hant93@chollian.dacom.co.kr

Philippines

Mutual Books, Inc.
429-D Shaw Boulevard
Mandaluyong City, Metro
Manila, Philippines
Tel: 632-725-7538
Fax: 632-721-3056
Email: mbikikog@mnl.sequel.net

Taiwan

O'Reilly Taiwan
No. 3, Lane 131
Hang-Chow South Road
Section 1, Taipei, Taiwan
Tel: 886-2-23968990
Fax: 886-2-23968916
Email: taiwan@oreilly.com

China

O'Reilly Beijing
Room 2410
160, FuXingMenNeiDaJie
XiCheng District
Beijing
China PR 100031
Tel: 86-10-66412305
Fax: 86-10-86631007
Email: beijing@oreilly.com

India

Computer Bookshop (India) Pvt. Ltd.
190 Dr. D.N. Road, Fort
Bombay 400 001 India
Tel: 91-22-207-0989
Fax: 91-22-262-3551
Email: cbsbom@giasbm01.vsnl.net.in

Japan

O'Reilly Japan, Inc.
Kiyoshige Building 2F
12-Bancho, Sanei-cho
Shinjuku-ku
Tokyo 160-0008 Japan
Tel: 81-3-3356-5227
Fax: 81-3-3356-5261
Email: japan@oreilly.com

All Other Asian Countries

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472 USA
Tel: 707-829-0515
Fax: 707-829-0104
Email: order@oreilly.com

Australia

WoodsLane Pty., Ltd.
7/5 Vuko Place
Warriewood NSW 2102
Australia
Tel: 61-2-9970-5111
Fax: 61-2-9970-5002
Email: info@woodslane.com.au

New Zealand

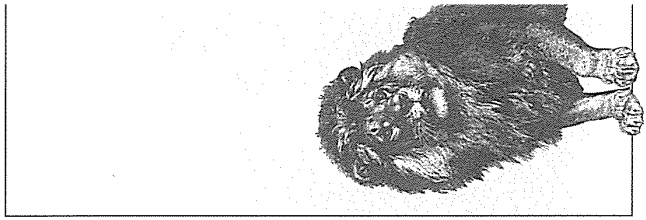
Woodslane New Zealand, Ltd.
21 Cooks Street (P.O. Box 575)
Waganui, New Zealand
Tel: 64-6-347-6543
Fax: 64-6-345-4840
Email: info@woodslane.com.au

Latin America

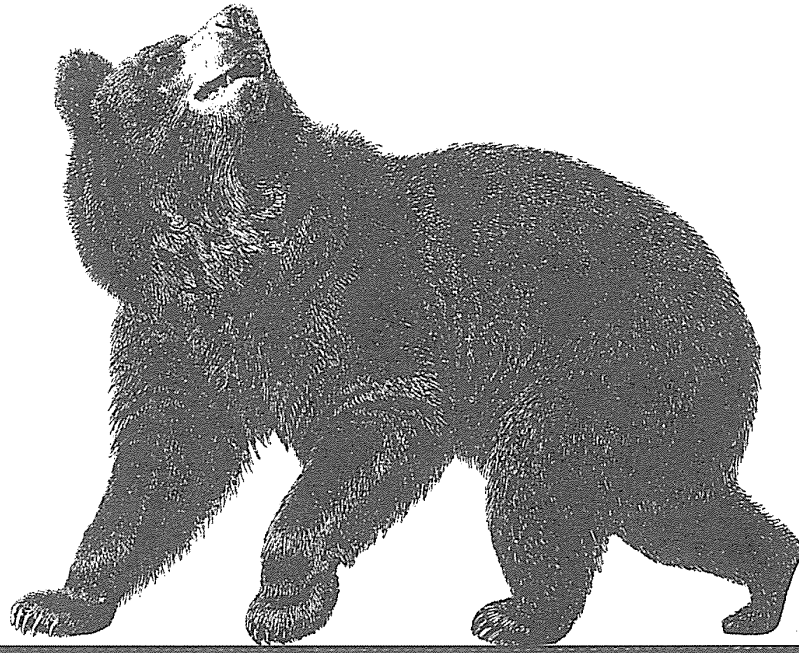
McGraw-Hill Interamericana
Editores, S.A. de C.V.
Cedro No. 512
Col. Atlampa
06450, Mexico, D.F.
Tel: 52-5-547-6777
Fax: 52-5-547-3336
Email: mcgraw-hill@infosel.net.mx

O'REILLY®

TO ORDER: **800-998-9938 • order@oreilly.com • http://www.oreilly.com/**
OUR PRODUCTS ARE AVAILABLE AT A BOOKSTORE OR SOFTWARE STORE NEAR YOU.
FOR INFORMATION: **800-998-9938 • 707-829-0515 • info@oreilly.com**



O'REILLY™
O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472-9902
1-800-998-9938
Visit us online at:
http://www.ora.com/
orders@ora.com



O'REILLY WOULD LIKE TO HEAR FROM YOU

Which book did this card come from?

_____

Where did you buy this book?

❑ Bookstore ❑ Computer Store
❑ Direct from O'Reilly ❑ Class/seminar
❑ Bundled with hardware/software
❑ Other _____

What operating system do you use?

❑ UNIX ❑ Macintosh
❑ Windows NT ❑ PC(Windows/DOS)
❑ Other _____

What is your job description?

❑ System Administrator ❑ Programmer
❑ Network Administrator ❑ Educator/Teacher
❑ Web Developer
❑ Other _____

❑ Please send me O'Reilly's catalog, containing a complete listing of O'Reilly books and software.

Name Company/Organization

Address

City State Zip/Postal Code Country

Telephone Internet or other email address (specify network)

Nineteenth century wood engraving of a bear from the O'Reilly & Associates Nutshell Handbook® *Using & Managing UUCP.*

PLACE STAMP HERE

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 80 SEBASTOPOL, CA

Postage will be paid by addressee

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472-9902

JAVA IN A NUTSHELL

This bestselling book is an essential quick reference for all Java programmers. It contains an accelerated introduction to the Java language and its key APIs, so seasoned programmers can start writing Java code right away. The third edition of *Java in a Nutshell* covers versions 1.2 and 1.3 beta of the Java 2 platform and includes:

- A description of the syntax of the Java language, written in a tight, concise style, that can serve as both a fast-paced tutorial and a language reference
- An explanation of the object-oriented features of Java that doesn't assume any prior object-oriented programming experience
- An overview of the essential Java APIs that shows how to perform common tasks, such as string manipulation, input/output, and thread handling, with the classes and interfaces that comprise the Java 2 platform
- Documentation for the Java development tools shipped with Sun's Java SDK

This book also includes O'Reilly's classic-style, quick-reference material for all the classes in the essential Java packages, including *java.lang*, *java.io*, *java.beans*, *java.math*, *java.net*, *java.security*, *java.text*, *java.util*, and *javax.crypto*. This reference material covers all the new classes in Java 1.2 and 1.3. Once you've learned Java, you'll keep this book next to your keyboard for handy reference while you program.

This book is part of the two-volume set of quick references that every Java programmer needs. It is an essential companion to *Java Foundation Classes in a Nutshell*, which covers the graphics and graphical user interface APIs in the Java 2 platform, including Swing, AWT, and Java 2D. A third volume, *Java Enterprise in a Nutshell*, focuses on the Java Enterprise APIs and is of interest to programmers working on server-side or enterprise Java applications.

ISBN 1-56592-487-8
US $29.95
CAN $43.95
X000YXAHAR
Java in a Nutshell
New
O'REILLY
A NUTSHELL® HANDBOOK
RepKover.