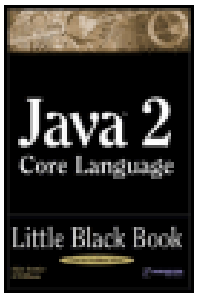

Java 2 Core Language Little Black Book
by Alain Trottier ISBN:1932111662

Paraglyph Press © 2002 (438pages)
The essential guide to Java programming.

Table of Contents

Java 2 Core Language Little Black Book
Introduction
Chapter 1 - Getting Started with Java
Chapter 2 - Essential Java Syntax
Chapter 3 - Blocks and Statements
Chapter 4 - Methods, Classes, and Packages
Chapter 5 - Data Structures
Chapter 6 - Files and Streams
Chapter 7 - Java Database Connectivity
Chapter 8 - The Internet and Networking
Chapter 9 - XML and Java
Chapter 10 - Advanced Techniques
Chapter 11 - Security
Chapter 12 - Internationalization
Appendix A - Development Tools
Appendix B - References
Appendix C - The Java Virtual Machine
Appendix D - Active RFCs
Index
List of Figures
List of Tables
List of Listings

TEAMFLY

Team-Fly®

Java 2 Core Language Little Black Book

Alain Trottier
Al Williams
CORIOLIS
President and CEO
Roland Elgey
Publisher
Al Valvano
Associate Publisher
Katherine R. Hartlove
Acquisitions Editor
Jawahara Saidullah
Development Editor
Jessica Choi
Product Marketing Manager
Jeff Johnson
Project Editor
Sally M. Scott
Technical Reviewer
Sumit Pal
Production Coordinator
Peggy Cantrell
Cover Designer
Laura Wellander

Copyright © 2002 The Coriolis Group, LLC.

All rights reserved.

This book may not be duplicated in any way withoutthe express written consent of the publisher, except in the form of brief excerpts or quotations for the purposes of review. The information contained herein is for the personal use of the reader and may not be incorporated in any commercial programs, other books, databases, or any kind of software without written consent of the publisher. Making copies of this book or any portion for any purpose other than your own is a violation of United States copyright laws.

Limits of Liability and Disclaimer of Warranty

The author and publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no

warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book.

The author and publisher shall not be liable in the event of incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the programs, associated instructions, and/or claims of productivity gains.

Trademarks

Trademarked names appear throughout this book. Rather than list the names and entities that own the trademarks or insert a trademark symbol with each mention of the trademarked name, the publisher states that it is using the names for editorial purposes only and to the benefit of the trademark owner, with no intention of infringing upon that trademark.

The Coriolis Group, LLC
14455 North Hayden Road
Suite 220
Scottsdale, Arizona 85260
(480) 483-0192
FAX (480) 483-0193
http://www.coriolis.com

Library of Congress Cataloging-in-Publication Data

Trottier, Alain

Java 2 core language little black book/by Alain Trottier and Al Williams.

p. cm.
Includes index.
1-932111-66-2

1. Java (Computer program language). I. Williams, Al, 1963- II. Title.
QA76.73.J38 T76 2002

005.13—3–dc21 2001058409

10 9 8 7 6 5 4 3 2 1

This book is dedicated to my wife, Patricia, the love of my life, and to my son, Devyn, who has brought us tremendous joy.

—Alain Trottier

As always, for my wife, Pat.

—Al Williams

About the Authors

Alain Trottier observes the dot-com warfare of Southern California from his vantage point as an independent consultant (Trottier Technologies) and Adjunct Professor at Vanguard University of Southern California. He has been in the tech sector for two decades, wearing many hats such as technologist, customer-support provider, programmer, architect, manager, and director. He has worked in a wide range of environments such as the U.S. Navy, Chevron's research center, and Adforce. His experience includes methodical, exacting research as well as code-til-ya-drop Internet, pre-IPO, ventures.

He is as fascinated by people as he is by the IT trenches they inhabit. His Microsoft and Sun certifications are balanced by his Bachelor's and Master's degrees in religion. Alain would be delighted to hear from readers who have requests or comments; he can be reached at http://www.inforobo.com/javacorelanguage.

Al Williams is a long-time consultant and author. His articles have appeared regularly in magazines such as *Web Techniques*, *Dr. Dobb's*, *Visual Developer*, and many others. He's the author of more than a dozen books on programming and computers, including *MFC Black Book* and *Java 2 Network Protocol Black Book* (both from The Coriolis Group). Al's programming career has seen him programming in Fortran, C, C++, and—more recently—Java. Al's consulting projects have included aerospace projects, chemical production software, and many embedded systems. He also teaches programming courses across the United States.

When Al's not working (which isn't often), he enjoys tinkering with amateur radio projects, watching or reading science fiction, and maintaining a few personal Web sites. He lives near Houston, Texas, with his wife, Pat, and a varying number of kids, dogs, and cats.

Acknowledgments

I would like to thank Jawahara Saidullah, Acquisitions Editor, and Chris Van Buren, agent at Waterside, who made this book possible. Thank you also to Jessica Choi, Development Editor, for your valuable guidance. Thank you, Sally Scott, Project Editor, and your team, for all the detailed work you do. Every book purchase is a nod to your effectiveness.
—*Alain Trottier*

Producing a book is a lot of work. Sure, writing is hard, but that's just the tip of the iceberg. Behind the scenes, a lot of people do most of the real work—the proofreading, the typesetting, the layout, the indexing, and all the other details that it takes to transform some random typing in Microsoft Word into the book you're holding in your hands.

To that end, I'd like to thank Sally Scott, Project Editor; Peggy Cantrell, Production Coordinator; Catherine Oliver, Copyeditor; Sumit Pal, Technical Reviewer; Christine Sherk, Proofreader; William Hartman, Compositor; Christina Palaia, Indexer; and Laura Wellander, Cover Designer. I'd also like to thank Jawahara Saidullah, Acquisitions Editor, for bringing me in to join the team that produced this book.

My thanks also go to Alain Trottier for all of his example code and other material. There's an old saying that two workers make twice the work, but fortunately, in this case, it cut the work in half, as it should.

Some of the material in this book appeared in a different format in my Java@Work column in *Web Techniques* magazine and appears with permission. Thanks to Amit Asaravala at Web Techniques for his continuing support and friendship.

I couldn't even start to tackle projects of this size and scope without the support of my family. My wife, Pat, and kids Jerid, Amy, and Patrick all have to put up with me seeming to live in my office, but they never complain (well—almost, never). Thanks guys! Oh, and if you've read this far—thanks to you, for taking the time to read about all of these people who are important to me.
—Al Williams

Introduction

Suppose you wanted to take up painting. You can learn about painting, basically, in one of two ways. First, you can make a study of painting and spend a lot of time reading and thinking about colors, lighting, and perspective. Eventually, you could make some simple sketches— maybe just some geometrical shapes—and tentatively try to paint them. After a great deal of study and effort, you might even be able to produce something you might be able to sell.

The second way is to buy a paint-by-number set. You've seen these, of course. You get a canvas with all the areas of the painting already marked and numbered. All you have to do is fill in the areas with paint from the little numbered tins, and you have a masterpiece.
Programming is a lot like painting. You can spend years training and planning and writing small tentative programs. Or you can use wizards and example code; essentially, this is a program-by-number. In this book, you'll find a hybrid approach that bridges these two philosophies. Modern tools and the Internet have made it very easy for people to create programs without having a real understanding of the details. This is especially true with graphical user interface builders that automatically write all but the small details of your application. On the other hand, most of us need professional results *now*. We don't want to spend years honing skills; instead, we need to produce results today.

We wrote this book to provide you with plenty of cut-and-paste examples, and also to explain the reasoning behind them. This lets you find a balance between your need to get things done and your need to understand why and how things work.

Why This Book Is for You

Java 2 Core Language Little Black Book is for the intermediate-to-advanced Java programmer. If you are just starting out, however, you might find it useful to study this book in conjunction with a language reference (such as the Sun documentation). This book concentrates on helping you apply the language in real-world development, presenting building blocks and the details about their construction.

You'll find that this book focuses on the Java language and its core libraries. While many books jump into graphical programming, this book focuses on non-graphical, text-based programs. Why? Because graphics programming techniques can obscure the fundamental Java tasks you need to understand. And also, setting up user interface screens is one place where the "paint-by-number" tools are something you probably will use. It is better that you have a firm grasp of the underlying Java language and calls. Even the most sophisticated graphics program uses the same core language techniques that this book illustrates.

Some of the book's sample code also comes from a working online service, the engine behind *Words Count, a Writing Analysis Tool* at http://wordscount.ezpublishing.com. This is because there is no tougher test for code than placing it in production—if you miss something, customers will howl. Using the unforgiving production test, as a result, keeps the material practical, and we wanted to give you something more than academic code to study.

In short, if you want to learn about the core Java language and libraries—in settings as diverse as Internet access, security, and XML— then this book is for you. If you've been using a "paint-by-numbers" tool and you want to increase your understanding of what's going on behind the scenes, you'll find that this book will help you fill in the gaps. If you are an experienced developer looking for quick code examples for database code, network servers, and other topics covered in this book, you'll find the *Little Black Book* format fast and useful.

You'll find quite a few example programs in this book, including:

- Examples of major language features, including arrays, exceptions, casting, and more.
- Object and package examples that show you how to get the most from the class system.
- Examples of using vectors, arrays, and other data structures to store and manipulate data efficiently.
- File and stream handling examples. These programs show you a variety of ways to deal with files and also illustrate how to apply the techniques to any data source, not just files.
- Examples that use JDBC (Java DataBase Connectivity) for database access. You can use JDBC to access a variety of SQL databases.
- Programs that act as Internet clients and servers, including programs that extract data from Web pages. These examples also show techniques used with network sockets and streams.
- Examples of using SAX and DOM parsers to read XML documents.
- Programs that use multiple threads to execute several tasks at once, which can provide better performance for programs that can benefit from multitasking.
- Examples that serialize objects to persistent storage and later restore them. You can use these techniques to save and restore data or transmit data across a network.
- Encryption, key generation, and digital signature examples. Encryption is necessary for many e-commerce and communications programs.
- Examples of internationalized programs that show how to accommodate multiple languages using techniques like resource bundling.

How to Use This Book

Since this is a *Little Black Book*, you'll find that each chapter has two sections. First, there is a brief overview of the chapter's topics. The second section, a set of practical immediate solutions, contains specific examples that illustrate the chapter's points. Often you can find a specific code example in this second section. If you want to customize it, you can drill down into the details by reading the first section.

If you haven't already installed Sun's free Software Development Kit (SDK) for Java, start with Chapter 1. Although you can use other products (like VisualAge or JBuilder), you'll get more of a low-level view with the SDK. The experience you gain will directly apply to writing programs using these more sophisticated tools. If you insist on using some other tool, be sure to avoid or disable as much of the automatic code generation as you can because, like painting by number, this hides much of what is really happening at the code level. Also, be sure your tool is using the 1.4 version of the Java 2 SDK.

This book uses version 1.4 of the Java 2 SDK, Standard Edition—the latest version of the language. Although many of the topics will still be useful if you use a different version, some of the examples might require modification if you aren't using the same version that we use in this book.

After Chapter 1, you can decide which approach you want to take. If you are already familiar with Java's syntax and semantics, you might just randomly thumb through the chapters starting with Chapter 5 until you find topics that interest you. Although some of the examples in these chapters assume familiarity with earlier chapters, if you are comfortable with Java you shouldn't have much trouble picking out what you need. If you are just starting with Java, or if you've been using tools that hide a lot of detail from you, or even if you just want a refresher course, pay special attention to Chapters 2, 3, and 4. These core concepts are not glamorous, but they are the foundation that all programs require. You can't build a castle on a shaky foundation, and you can't write a great Java program without understanding why and how things work in Java (including the class system). Once you are comfortable with these chapters, you can pursue the more advanced chapters that appear later in the book.

The *Little Black Book* Philosophy

Written by experienced professionals, Coriolis *Little Black Books* are terse, easily "thumb-able" question-answerers and problem solvers. The *Little Black Book*'s unique two-part chapter format—brief technical overviews followed by practical immediate solutions—is structured to help you use your knowledge, solve problems, and quickly master complex technical issues to become an expert. By breaking down complex topics into easily manageable components, this format helps you quickly find what you need with the code you need to make it happen.

We welcome your feedback on this book. You can email us at The Coriolis Group at ctp@coriolis.com.

Chapter 1: Getting Started with Java

In Brief

This book focuses on the core Java language, an easy-to-learn, yet powerful, programming language. The book will teach you what you need to know in order to produce full-featured software products based on Java. It concentrates on the core language, the use of which involves much more than understanding a list of simple keywords and syntax. Knowing how to use a language well requires more than the ability to spell correctly; the nuances come, instead, from word combinations. Java and English are similar in that they both have syntax, grammars, rules, and conventions—the core of any language.

You can divide Java into two main parts. The first part (the *Java Virtual Machine*) executes Java programs. The second part (the *Software Development Kit*) is the set of tools used to develop Java programs.

For execution, you need a compiled program and a Java Virtual Machine (JVM), which serves as the intermediary between your program and the host computer. The JVM provides cross-platform portability because your program will work on any JVM regardless of the operating system or hardware involved. Of course, there are different versions (or editions) of the JVM, so in practice some programs will require specific JVM versions. However, the idea is that one Java program can run on many JVMs regardless of the operating system or computer hardware involved.

Note Sun (and Sun-approved vendors) give away a JVM with every major operating system. Of course, your users' machines need a JVM, but many new PCs, servers, and even handheld devices already have one installed. If not, Sun allows you to redistribute the user portion of Java with your product.

For development, you need only a text editor and the basic Java tools that Sun provides for free. There are fancier tools available—some for free.

This chapter will introduce the Java architecture, providing you with the background information you'll need in order to better understand what you're doing in later chapters. This chapter will also teach you how to enter code, compile it, and then run it on a JVM.

Java Editions

Sun uses a peculiar naming scheme to differentiate different versions of Java. First, Sun generates an abstract specification that defines what Java is. This is known as the *platform*. Major specification changes would require a change in platform. Then, a particular version of Java might target a different type of application (for example, a desktop computer or a handheld computer). These different types are known as *editions*. Finally, each specific implementation of an edition on the platform has a version number and is known as a *Java SDK* (Software Development Kit; formerly known as the Java Development Kit or JDK).

Tip For a good glossary, see "Unraveling Java Terminology," by Dana Nourie, at http://www.developer.java.sun.com/developer/onlineTraining/new2java/programming/learn/unravelingjava.html.

Sun has three editions of Java for a given platform or version (the current platform is Java 2). The editions for the current platform are:

- *J2ME (Micro Edition)*—Used to create programs that run on small handheld devices, such as phones, PDAs (personal digital assistants), and appliances.
- *J2SE (Standard Edition)*—Used primarily to create programs for desktop computers or for any computer too large for J2ME and too small for J2EE.
- *J2EE (Enterprise Edition)*—Used to create very large programs that run on servers managing heavy traffic and complicated transactions. These programs are the backbone of many online services, such as banking, e-commerce, and B2B (business-to-business) trading systems.

Each edition has different capabilities. It's important for you to know the differences among the editions because they affect your projects. Many programmers start with J2SE and intend to jump to J2EE later. The advantage of this approach is that the development team can get up to speed on Java technology with J2SE, which is not as complex as J2EE. The disadvantage is that the team will face a major conversion to move to J2EE. In addition, equipping a team for J2EE can be significantly more expensive than equipping a team for J2SE. The SDK is free, but you'll probably want a server machine in addition to workstations—not to mention database software and multiple clients for testing. There is a world of difference between writing and testing code snippets on a standalone machine and stress-testing the whole product on a true enterprise system that involves multiple servers, clients, and middleware.Therefore, if you have J2SE v1.3, you know that it is the Java SDK version 1.3, targeting desktop computers (Standard Edition) and that it is based on the Java 2 platform (specification). Presumably in the future there will be a Java 3 and even additional editions (unless Sun changes the naming scheme again).

The Java SDK

For each Java edition, Sun gives away a Software Development Kit (SDK), which has everything you need to compile and test programs. Most of the code in this book was developed with the J2SE SDK. Be warned that the programs you compile for one edition will often not work for the other two editions. J2SE compiled programs will work on J2EE because J2EE is a true superset of J2SE. J2EE programs that use special J2EE features won't work with J2SE, however. Nor will J2SE or J2EE programs run on J2ME, since the JVMs are quite different between the two editions.

The SDK is a development environment for building programs using the Java programming language; the SDK includes everything you need to develop and test programs. The tools include command-line programs (which were used, incidentally, to develop the samples for this book). Although these tools do not provide a graphical user interface (GUI), using them is a good way to learn the Java language.

Besides, if you understand how the core tools work, you'll have no problems using one of the many integrated development environments (IDEs) available.

Tip You can download the current release of the Java 2 SDK, Standard Edition from http://www.java.sun.com.

The SDK provides many tools, the three most important of which are:

- The compiler—The compiler converts the human-readable source file into platform-independent code that a JVM interprets. This code is called *bytecode*.
- The runtime system—The SDK includes a JVM that allows you to run Java programs and test your programs. The runtime system also includes a command-line debugger that you can use to monitor your program's execution.
- The source code—Sun provides quite a bit of source code for the Java libraries that form part of the JVM. You shouldn't change this code directly. Thanks to object orientation, however, you can modify these classes by making new classes that extend the existing ones. Examining the source code is often helpful in understanding how a class works.

If you are familiar with other programming languages, you might wonder about linking. A C program, for example, is not only compiled but also linked with other library modules to form an executable program. This linking is not necessary (or even possible) in Java. The JVM dynamically searches for and loads library modules as the program needs them. This dynamic loading is a crucial capability. For example, a Java program embedded in a Web browser can load modules over the Internet. The browser does not need to know anything about the modules at compile time. The linkage is handled completely at run time.

The Java Virtual Machine

Java is the first truly useful portable language. The JVM architecture offers you several advantages: cross-platform portability, size, and security.

Cross-Platform Portability

The JVM provides cross-platform portability. You write code for the JVM, not for the operating system (OS). Because all JVMs look the same to Java programs, you have to write only one version of your program, and it will work on all JVMs. The JVM interprets the byte-code and carries out the program's operations in a way that is compatible with the current hardware architecture and operating system.

Team-Fly®

Size

The second interesting side effect of using JVM architecture is the small size of its compiled code. Most of the functionality is buried in the JVM, so the compiled code that runs on top of it doesn't need to be loaded with large libraries. Of course, the JVM is, among other things, a large library, but it is shared among all Java programs. That means that a Java program can be quite small—at least, the part of the program that is uniquely yours. All Java programs share the large JVM, but presumably it is already on the target machine. This is especially important when users are downloading programs over the Internet, for example. Of course, if users' computers don't have a JVM, they'll have a large download for installing the JVM on their machines first. After the JVM installs, the users won't have to worry about installing again.

Security

Java has been designed to protect users from malicious programs. Programs from an untrusted source (for example, the Internet) execute in a restricted environment (known as a *sandbox*). The JVM can then prevent those programs from causing mischief. For example, a Java applet (a small program that runs on a Web page) usually can't access local files or open network connections to arbitrary computers. These restrictions prevent a Web page from erasing your critical files or sending threatening email from your computer.

Data Types and Unicode

Software has to manage many types of data, including numbers, dates, currencies, and letters. There are several numeric data types and two character data types, among others. Whereas the decimal number system is universal, alphabets vary considerably. What do you do if you want to name a variable using Cyrillic letters, for example? To accommodate these variations, Java uses the Unicode character set.

At the basic level, computers only crunch numbers. When it comes to character data, the computer assigns a number for each letter. Because computers need to communicate with each other, there should be a standard way to map characters to numbers. For years, the two predominant standards were ASCII (the American Standard Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code). Both of these used 8-bit numbers to represent common characters used in the United States.

With only 8 bits, these character sets can handle only 256 unique characters (including spaces and control characters). However, this capability is not adequate for truly international programs that might encounter languages using a variety of alphabets, such as Farsi or Cyrillic.

To deal with these problems, a new standard emerged: Unicode (see http://www.unicode.org). Unicode uses 16-bit (or even 32-bit) characters that allow it to represent a large number of symbols. The 32-bit Unicode system (with 4.3 billion characters) can handle all known languages, including the huge Asian character sets, and still leave room for growth. There are even Unicode characters for

dead languages such as Sanskrit. Unicode is used by Java, as well as by XML, ECMAScript (JavaScript), and many others. It is also used by most modern operating systems and Web browsers. That means you can write programs that deal with international character sets. This Unicode support is provided throughout Java. Not only can your programs handle user input and output in Unicode, but the programs themselves are written using Unicode. You can name variables and functions using any character set you can represent.

If you are an American used to dealing with ASCII, don't worry. It just so happens that the first 256 characters of the Unicode set correspond to those in the ASCII character set. Programs like the Java compiler can recognize ASCII files, so you can write programs with any text editor. A special encoding (known as *Unicode Transformational Format*, or *UTF*) allows you to specify extended characters in an ASCII file so you can take advantage of Unicode even if you don't have a Unicode-aware text editor.

Java Tools

Many tool vendors provide additional tools for Java (some free and some for a price). Because the SDK is command-line oriented, it's no surprise that tool vendors sell IDEs (integrated development environments) that enable you to edit and compile code from a GUI, improving productivity. Even Sun has a GUI IDE: Forte. You can download a free version of Forte or buy a version with additional features.

> **Note** You can read more about tools in Appendix A. For now, just understand that, although these tools can improve productivity, they are not generally discussed in this book. Once you are comfortable with the basic Java system, you'll be able to use any IDE with ease.

Several good editors are on the market. IBM provides a world-class IDE called VisualAge for Java. The entry-level version is free and targets the IBM WebSphere software platform (a combination Web server and application server)—not bad, considering that IBM is giving away WebSphere with a one-user license. You can download these two tools and use them to develop powerful applications. Inprise (formerly Borland) offers JBuilder, and Sun provides Forte.

> **Note** Be careful if you install these IDEs because they might take a huge amount of disk space. Also, many are written in Java. This is a good idea, but, because the JVM interprets the programs, the IDEs may require a fast computer and lots of memory to be truly useful.

Other tools are also available, but all you need for now is a text editor. We will be starting with simple programs, so you don't need anything fancy. I use Notepad, the simple editor included with the Windows OS. You can use plenty of other editors, ranging from simple Notepad replacements and vi (a common Unix text editor), to large text-editor systems such as Emacs. The key is to use something you are already comfortable with so you can focus on the Java code instead of on the text editor.

SDK Contents

The SDK provides you with several tools that you'd expect to receive from a language vendor, along with a few additional tools that help with the overall development effort. The basic components include the compiler (javac.exe under Windows), the runtime engine (java.exe), and the debugger (jdb.exe). The SDK provides a few other tools that you probably won't use as often:

- *javadoc*—Generates HTML documentation from special comments in your files.
- *appletviewer*—Runs and debugs applets (small programs that run in other programs).
- *jar*—Manages Java archives (collections of files similar to a Zip file or a compressed tar archive).
- *native2ascii*—Used to convert files that contain native-encoded characters into UTF format.
- *keytool, jarsigner, policytool*—Provide security tools.

The SDK also has tools that handle network programming, but you won't need these for a while yet.

Multiple Versions of the Java 2 SDK

The examples in this book were written using the Java 2 SDK version 1.4. Usually, newer versions of the SDK will work with older versions, but to be safe you should work through the examples using version 1.4, if possible. Installing multiple SDK versions is possible; if you are not careful, however, installing two or more versions can cause problems. For example, if you compile with one version and inadvertently execute with a different runtime version, your program will probably not work properly.

You can have two versions of the SDK installed, however, if you are very careful to keep them separate. If you have 1.3 of the Java 2 SDK, and you install Java 2 SDK 1.4, you will have to decide which version's binaries (i.e., java.exe, javac.exe) you want the **PATH** variable to point to. Also, Windows systems have DLLs that are difficult to keep separate between versions.

New Features in Java 2 SDK 1.4

Java in general has many powerful features, and Java 2 SDK 1.4 provides a few new twists. Many of the improvements are performance related, so they don't change your programming.

One major change that is apparent is Java's handling of XML. Prior to SDK 1.4, you had to add someone else's XML parser to your program to handle XML. Now, Java has its own XML parser. The next release will include a major expansion of this area, but at least now there is a native XML API.

Before SDK 1.4, real database work required you to buy or download and install third-party packages. It is such a relief that SDK 1.4 now includes APIs for JDBC (Java Database Connectivity) 3 as part of the core Java platform. Relying on additional, third-party packages for this important functionality was

irksome. SDK 1.4 also adds full support for Unicode 3. Java now supports all major locales on the planet, making internationalization easier. For example, this support affects the handling of currency—Java now makes the euro the default currency used by the **java.text.NumberFormat**, **DecimalFormat**, and **DecimalFormatSymbol** classes for the member countries of the European Monetary Union.

Finally, there is a major improvement to security. Java now includes JSSE (Java Secure Socket Extensions), JAAS (Java Authentication and Authorization Service), JCE (Java Cryptography Extensions), and Kerberos security capabilities. In fact, Java Kerberos supports single sign-on using OS credentials.

You may redistribute the Java 2 runtime environment with your applications, subject to Sun's license. The Java 2 runtime environment can be downloaded separately, offering you a way to distribute your program and the Java virtual machine it needs to run. The JVM comes in the Java 2 SDK, or you can download the JVM alone and then give it out so your end users will have a JVM with which to run your software.

Source Files and Compiled Files

To produce a Java program, complete the following three steps:

1. Create your source code.
2. Compile your files into bytecode.
3. Execute the main bytecode file using a JVM.

A *source file* contains the Java program as text. You can place this file anywhere, but keep in mind that Java development is easier if you organize files properly.

Tip We recommend creating a directory, like C:\myPrograms, somewhere other than under the installed SDK directory. Under this new directory, create a few throw-away directories, like test1 and test2. Keep the source and compiled files together. Some people advocate keeping all the source files together and all the compiled files in another directory. Don't do that at first. Keep things simple. We also don't recommend that you put your files under the SDK directory because you might accidentally delete your own files when you delete the SDK (for example, after upgrading to a newer SDK).

The source file *must* have a .java extension, like this: myFirstProgram.java. If you don't use this extension, the compiler will ignore the file.

The compiler creates a file by the class name, not by the file name. So when you compile a file, the compiler will create a new file with the class name and will add the extension .class, like this:

myFirstProgram.class. This file is the one containing the bytecode. Once you have all this, you can run the program, and it is at that point that the JVM will interpret your bytecode file and execute code.

Because the compiler needs the class name, the compiler will insist that the file name matches any public class name (names are case-sensitive). So if you try to put a public class named **MyFirstClass** in a file named My1stclass.java (or myfirstclass.java), the compiler will generate an error.

Immediate Solutions

Downloading the SDK

A current release of the Java 2 Platform SDK (Standard Edition version 1.4 for Windows, Linux, and Solaris) is available at Sun's Web site. To download the SDK, follow these steps:

1. Go to http://www.java.sun.com.
2. Scroll down until you see a drop-down list box labeled "The Java Platform" on the left side of the screen.
3. From the drop-down list, select Java 2 Platform–Standard Edition. Click the Go button to go to the home page for the Java 2 Platform, Standard Edition (shown in Figure 1.1).

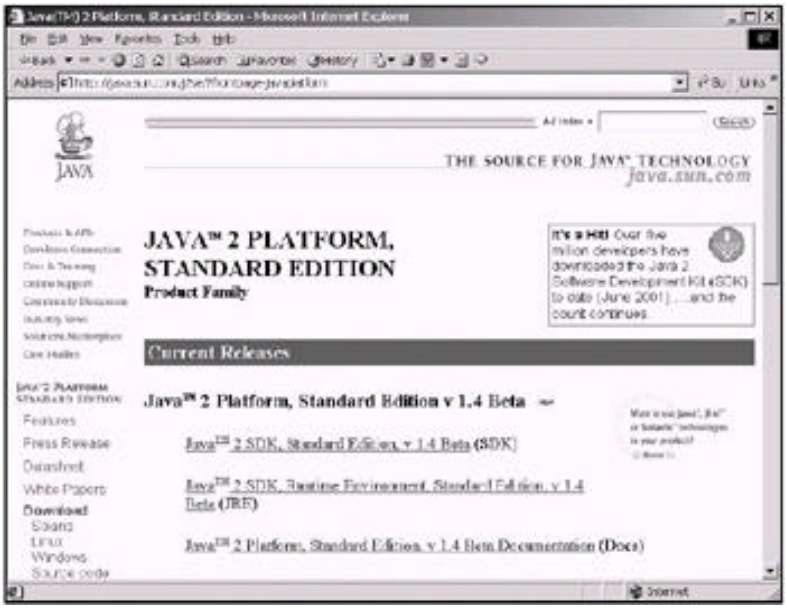

Figure 1.1: The Java 2 Platform, Standard Edition home page.

4. Click the hyperlink for the Java 2 SDK, Standard Edition v 1.4.
5. Choose your operating system, as shown in Figure 1.2.

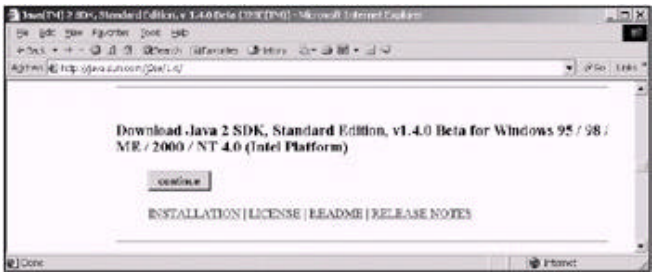

Figure 1.2: Selecting the operating system.

6. You will be presented with Sun's "Terms and conditions of the license" agreement. Click the button that signifies you agree.

7. You'll next get the page where you can finally download the SDK; see Figure 1.3. Click one of the Download buttons to download the software.

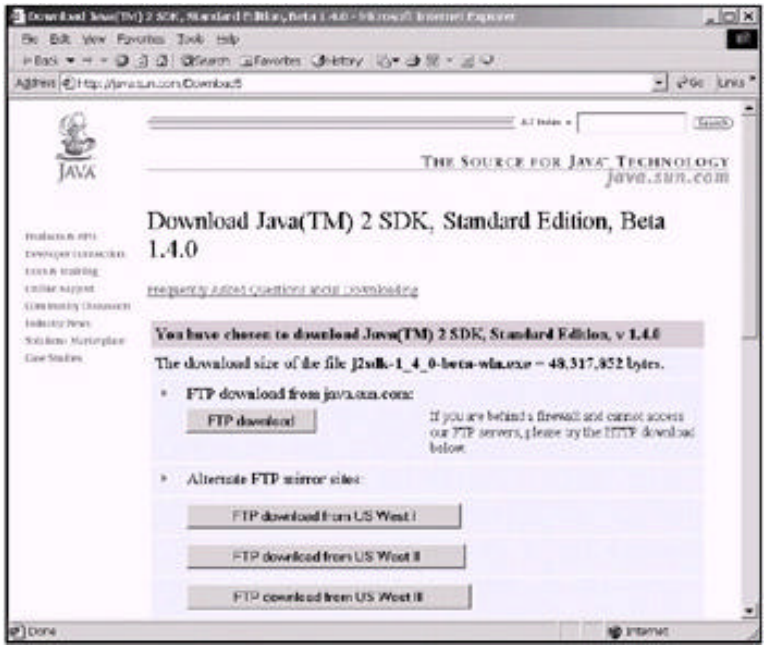

Figure 1.3: The download page.

> **Tip** We always select one of the alternate download sites; the main site is sometimes slow because most people click the top Download button.

8. You will get a file like this: j2sdk-1_4_0-win.exe (for Windows). It is the installer that decompresses the files and then copies them to your destination folder. The installer file is large, around 50MB.

> **Tip** If you don't have a fast or reliable Internet connection, you might want to use a download-management program (for example, FlashGet at http://www.amazesoft.com) that allows you to download faster and, what's more important, download this large file in multiple attempts.

The SDK for the Standard Edition doesn't include advanced features that are found in the Enterprise Edition, such as RMI (Remote Method Invocation) and email functionality. This SDK does, however, include everything you need in order to build non-Enterprise applications, including typical network programs.

Installing the SDK

The file j2sdk-1_4_0-win.exe is the Java 2 SDK installer. Double-click it when you have it on your hard drive. Then follow the instructions the installer provides. When you're done with the installation, you can delete the download file to recover disk space.

Running the SDK Installer

When you're ready to run the SDK installer, you will be prompted for the installation directory. The default for the Java 2 SDK 1.4 is c:\jdk1.4. You can install the SDK wherever you want to, but keep in mind that you will be typing this path in many places.

Tip We don't like installing software in the root directory—this, to us,is sacred real estate. Instead, we always create a dev directory for all development files, such as sample Java code. Under the dev directory, we have a java directory for all things related to Java development. So the directory looks like this: c:\dev\java\jdk1.4.

If you accept the default installation, then create this directory: C\jdk1.4\myPrograms\test1. The Windows Java 2 SDK has the directory structure shown in Figure 1.4.

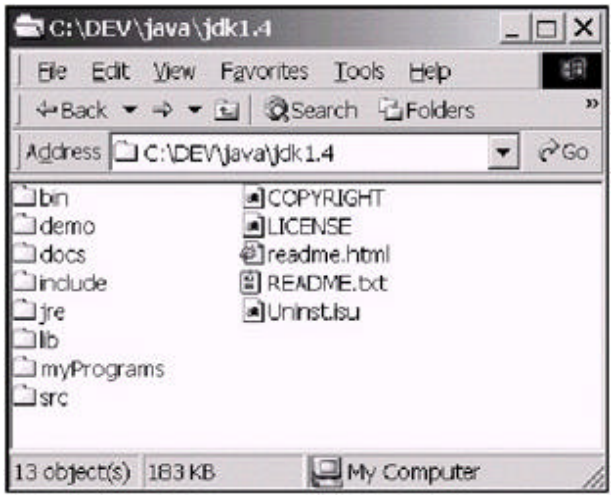

Figure 1.4: The recommended directory structure for the local hard drive.

Setting the Installation Path

You can run the Java 2 SDK without setting the Windows **PATH** variable, but it is best to set it. The installation instructions describe how to do this. For example, to set the **PATH** variable permanently in Windows 2000, you open the Control Panel, choose the System icon, select the Advanced tab of the System Properties dialog box, and choose the Environment Variables button. Take a look at Figure 1.5, which shows the Edit System Variable dialog box.

Figure 1.5: Setting the **PATH** system variable.

Setting the **PATH** variable permanently allows you to run your Java programs from any directory. Otherwise, it can be cumbersome to specify the path to the executable files every time you compile or run your program. A few little things can go wrong when you try to compile and run a Java program. You can run out of disk space, or you can get the following error message: "This program cannot be run in DOS mode." You can fix this problem easily by changing the properties of the command window, as explained in the Java Installation Notes that were installed with the SDK.

In addition to **PATH**, Java uses another environment variable **CLASSPATH** to locate nonstandard class files and packages. Recent versions of Java automatically include standard classes, and classes in the current directory. Therefore, you won't have to deal with **CLASSPATH** until the examples become a bit more complicated.

Related solution:	Found on page:
Using Packages	106

Obtaining Documentation

The SDK doesn't include much documentation. We recommend that you download the documentation by going back to the http://www.java.sun.com page and selecting Documentation under the heading "The Java Platform." The documentation installer is just as big and expands to about 9,000 files, eating 200MB of your hard drive. This documentation includes general information, API and language details, a guide to features, documentation for the tools (compiler, documentation creator, runtime engine, and others), and tutorial and demonstration information. If you have a fast,

always-on Internet connection, you might prefer to just browse the documentation online at the http://www.java.sun.com documentation site (http://www.java.sun.com/j2se/1.4/docs/index.html).

Writing Code

After installing Java, start your text editor. Enter the following code into your text editor (for this example, the file name must be countShoppersApplication.java):

```
/*****************************************
* This application counts new,           *
* returning and total shoppers.          *
 *****************************************/
class countShoppersApplication
{

    public static void main(String[] args)
    {
      // tracks the number of first time shoppers
     int numberOfNewShoppers = 72 ;

    // tracks the number of shoppers with previous history
     int numberOfReturningShoppers = 81 ;

    // keeps a running total of all shoppers,
      // representing a count of all customers for the day
     int numberOfShoppers = 0 ;
       numberOfShoppers = numberOfNewShoppers +
             numberOfReturningShoppers ;

     // display the number of shoppers
     System.out.println( numberOfShoppers ) ;

    }

}
```

Now, save the source file in your development directory as countShoppersApplication.java.

Note Depending on your Windows setup, Notepad might try to append a .txt extension to your file. Notepad doesn't check to see if you have typed your own extension, so be sure to change Save As Type to All Files, or you will end up with the incorrect name countShoppersApplication.java.txt. One way to ensure that this doesn't happen is to put double quotes around the file name in the Save As dialog box.

This test program will be explained line-by-line in Chapter 2.

Compiling a Java Program

Now, you are ready to compile your first Java program. To do so, follow these steps:

1. Execute the javac.exe program (the compiler) to compile the source file you just created. In older versions of Windows, use the DOS prompt; in newer versions (Windows 2000), choose Start|Programs|Accessories|Command Prompt.

   Do not use the Start|Run command; if you do, the window will close upon completion of your program without giving you a chance to see what happened.
   At the prompt, change the directory by typing the following:

   CD \myPrograms\test1

2. You should have set the path as mentioned earlier. If you didn't, you can issue the following command (using your SDK directory's name, of course):
3. PATH %PATH%;c:\dev\java\jdk1.4\bin
4. Compile your source by typing the following:
5. javac countShoppersApplication.java
6. Look in the directory by entering "dir" or using a Windows Explorer window.

   If there were no errors, you should find a file named countShoppersApplication.class.

The compiler does not display a message in the Command Prompt window unless you have encountered an error. If you do get an error message, then you probably have a path problem or have mistyped something. Check your path carefully. Examine the lines the compiler doesn't like and the nearby lines for mistakes.

Notice that the file name is the same as the class name. You can use a different name for the file, and Java will compile and run it correctly. That's because the class is not a public class. We strongly recommend naming source files the same as the primary class names, however. Not only does this make things easier, but it is a requirement when you write public classes, as you will in later chapters.

Team-Fly®

The file named countShoppersApplication.class should appear in the directory. You will be working with twin files: the source file, which ends with .java; and the compiled file, which ends with .class.

Executing a Java Program

The last thing we will do in this chapter is to run the program. To do this, issue the following command in the same command window that you used for compiling:

```
java countShoppersApplication
```

You'll need the same path setup that you used for compiling.

The program you wrote defined a class (**countShoppersApplication**). In that class, you defined a method (**main**). Within that method, you declared variables (**numberOfNewShoppers**, **numberOfReturning Shoppers**, **number Of Shoppers**), assigned values to them, computed a sum, and then printed the sum to the screen (**System.out.println**). Congratulations! It gets easier from here.

Chapter 2: Essential Java Syntax

In Brief

In this chapter, you'll learn what the compiler recognizes as legal characters and statements. Although syntax details aren't the glamorous part of programming, a solid understanding of Java's syntax is necessary before you plumb Java's depths.

Syntax

Computers can't understand humans very well. Even a young child has a better grasp of language than do today's most sophisticated computers. To help the computer, we use special languages to describe the tasks we want it to do. The compiler—a special computer program itself—translates our special language into commands the computer can execute. Because the compiler itself is a computer program, it isn't very at good understanding humans, either.

To simplify the computer's task, languages such as Java define strict rules you must follow. If you deviate from these rules, the compiler will generate an error. You want to learn the correct rules—the language syntax—so that they become second nature. That way, you can focus on the actual programming problem at hand and not on the language details.
Compilation starts by breaking your code into lines that are defined by line terminators. Then the compiler breaks lines into chunks of text called *tokens* . Tokens are defined or delimited by white space. The compiler then identifies the meaning of each token and converts the program into bytecodes.

Objects Everywhere

Java programs are built around classes. A *class* defines data and programming that go together to represent something. For example, if I worked for a grocery store, I might write one class to represent the store, another dass to represent aisles in the store, and yet another class to represent cash registers.

The **CashRegister** class is a prototype for all cash registers. In my program, I'll use a class to create one or more *instances* of these classes; these instances are often called *objects (* these two terms are more or less interchangeable). Each class contains a number of *fields* (places to store data) and *methods* (program steps that operate on the object's data).

For example, the **CashRegister** class might have a field (some people call these fields *variables*) that contains the amount of money in the drawer. The methods of this hypothetical class might be **sale**, **refund**, **printReceipt**, and similar cash-register-related operations.
Don't forget, the **CashRegister** class doesn't represent the cash register. Instead, you use the **CashRegister** class to create objects that represent cash registers. Think of classes as cookie cutters. Cookie cutters aren't cookies; you use the cutters to make cookies. You can make one cookie or dozens of cookies. They will all be the same, but they will all be separate entities.

You'll read more about classes, fields, and methods in this chapter and throughout the rest of this book. Understanding the basic structure of a Java program will help you digest the material in the rest of this chapter.

Basic Structure

Every Java program has at least one class. Some classes are *public*; that is, they are visible from any other part of the program. Other classes may not be visible from everywhere; either they are *private*, or they have package scope (discussed in Chapter 4). Java is very strict about public classes. Each file you create can contain only one public class. If your program needs more than one public class, you'll need to create more than one file. Furthermore, the file's base name must exactly match the name of the public class. If you create a public class named **CashRegister**, therefore, it must reside in the file CashRegister.java. Java even recognizes the case of the file name, so it must match exactly.

Before stating the class definition, your source file may contain **import** statements. Superficially, **import** statements resemble **include** statements that you might find in other languages, such as C or C++. This analogy isn't perfect, however. The **import** statement is actually much smarter than a C **include** statement (for example, you don't have to worry about importing the same file twice). Also, C **include** statements can contain any sort of code or macros you want. The Java **import** merely provides simplified access to another object's public methods and fields.

To understand **import** completely, you have to realize that Java organizes classes into *packages*. By default, if you use a class name in your program, the compiler looks for this class in two places: in the package you are currently creating (which may be the default package that has no name), and in the special package **java.lang**. For example, suppose you write:

```
String s = "Welcome to Coriolis";
```

This defines a string variable named **s**. In Java, strings are objects, and the **String** object is part of the special system package **java.lang**. Therefore, you could write:

```
java.lang.String s = "Welcome to Coriolis";
```

These two statements are equivalent. Suppose you want to use classes from a different package, however. For example, you might want to use the **Socket** class from the **java.net** package. You could write:

```
java.net.Socket sock;
```

It gets tiresome to keep specifying this lengthy prefix in front of the class name, however. That's where **import** helps you. By specifying an **import**, you can tell the compiler that you will be using classes from another package. You can **import** a specific class like this:

```
import java.net.Socket;
```

You can also **import** an entire package, and gain access to all the classes in that package. Simply place a star at the end of the package name, like this:

```
import java.net.*;
```

You read earlier that a class is a cookie cutter that creates objects. You've also read that all Java programs are objects. This leads to a chicken-and-egg problem: What creates the first object in your program? The answer is that you do. In a conventional Java program, one class will have a static function named **main**. When a function is static, it is really part of the class and not part of the individual objects in the class. That means that static functions exist even before you create any objects.

Java calls the **main** function in your object. This is your chance to create a new object or to do any other processing that is required. Very simple programs can do all their work in the **main** function. Consider the simple program in Listing 2.1. This program prints a friendly Texas-style greeting.

Listing 2.1: This simple program does all of its work in the main function.

```
public class Howdy

{

  public static void main(String [] args)

  {

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

  }

}
```

This program has only three significant lines. The first line begins the definition of the class (**Howdy**). The second line begins the definition of the **main** function. The third line prints a message on the system console.

> **Tip** A common mistake is to try to simplify the above program by using **import** with **System.out.println** or **System.out.*** so that you can use **println** instead of **System.out.println**. That sounds like a good idea, but it won't work. In future chapters, you'll learn that although **System.out** looks like a package, it isn't. **System** is really an object itself and not a package.

Although some of the details in Listing 2.1 might not be apparent, you can probably make several observations about the program. First, the **Howdy** class is public, so the file name must be named Howdy.java. You can see that curly braces group statements together. You'll read more about this

throughout this chapter. For now, notice that the final curly brace ends the definition of the class. The penultimate brace ends the definition of the **main** function. (Functions are also known as *methods*.)

> **Tip** Parameters are the variables that are declared in a method's opening parentheses and that contain values sent to the method. In this case, the **main** method accepts a **String** array named **args**. Java supplies any command-line arguments in this array. You'll learn more about arrays later in this chapter.

This simple program doesn't create any objects. You could rewrite it to be more object oriented, if you like (see Listing 2.2). This example uses several features you'll read about later.

Listing 2.2: This simple program creates an object.

```
public class Howdy {

// This function does the work

  void execute() {

      System.out.println("Howdy");

      }

  public static void main(String [] args) {

    new Howdy().execute();   // create a new object and

                             // call execute

      }

 }
```

Comments

Notice that Listing 2.2 contains two pairs of slash characters (//). This notation is one form of *comment*. These notes are meant for humans, and the compiler ignores them. Comments are very helpful to other people who are trying to read your code. Comments can also help you when you have to revisit code you wrote in the past.

You can use two different types of markers for comments. One type of marker, as you've seen, is the pair of forward slashes (//). The compiler ignores anything after the double forward slash on the same line (between the slashes and the end of the line). This type of comment is often used to make a short note about a particular line.

For multiple-line comments, you can use /* and */ to delimit as much text as you want. The compiler ignores anything between /* and */, even if the comment spans many lines. The following examples would all be ignored by the compiler:

```
/* This single line is ignored by the compiler. */

/*******************************
* These seven lines            *
* are ignored by               *
* the compiler,                *
* including the next statement: *
* int a = 29;                  *
*******************************/
```

You can place the /* and */ comments anywhere you can put a space. The examples at the top of the next page are legal (although not necessarily good form because they make the code harder to read):

```
int numberOfNewShoppers/* first time shoppers*/ = 72;
/*integer*/int /*new shoppers*/ numberOfNewShoppers = 72 ;
```

Comments do not nest. That means you shouldn't mix single-line and multiple-line comments. Once you begin a single-line comment, the multiple-line comment tag has no meaning, so /* and */ are ignored on the same line if they occur after //. Also, // is ignored if it occurs anywhere between the multiple-line comment tags.

There is one more rule you must follow. You can place multiple-line comments between any tokens (sometimes very good practice), but don't place comments within quotes (character literals or string literals) or in the middle of a number or name (such as a variable, keyword, or reference). After the comments are stripped away, the compiler starts looking for line terminators.

Sometimes you'll see comments that start with a slash and two asterisks. To the compiler, this is just an ordinary comment that happens to start with an asterisk. However, special tools can read your program and automatically generate documentation (in the form of Web pages) by reading and interpreting these special comments. Therefore, you shouldn't use the /** syntax unless you mean to create these special comments. In Appendix A, you'll learn about the javadoc tool, which interprets these comments.

Line Terminators

The compiler collects all characters it encounters, left to right, until it finds a terminator. Everything the compiler finds between terminators is considered a single line (which is different from a statement). The compiler recognizes three line terminators. They are:

- LF (line feed or newline)
- CR (carriage return or just return)
- CR + LF (*CR* immediately followed by *LF* is counted as one line terminator)

When the compiler finds one of these terminators, it parses the text of that line into tokens by looking for white space.

White Space and Tokens

Java recognizes tokens—words that potentially have meaning— by using white-space characters to mark the end of tokens. White-space characters include the blank, tab, form-feed, and end-of-line characters. When the compiler runs across a white-space character, the compiler ignores subsequent white space until it reaches another token (indicated by a non-white-space character). The exception, of course, is within quoted string constants (like the output string in Listings 2.1 and 2.2), where every space counts. The compiler finds that the following two statements are equivalent:

```
int numberOfNewShoppers = 72 ;
  int             numberOfNewShoppers      =          72      ;
```

The characters the compiler finds between white-space characters are tokens (**int**, **numberOfNewShoppers**, **=**, **72**, and **;** in this example). The compiler further breaks the tokens it finds on this initial pass into more tokens based on specific rules. For example, consider this statement:

```
int x=10;
```

Initially, this appears to be two tokens, **int** and **x=10;**, but there are actually five tokens: **int**, **x**, **=**, **10**, and the semicolon.

Although the compiler doesn't care much about white space, you can use spaces to your advantage to make your code more readable.

Now that the compiler has broken a given line into tokens, it goes through each one, character by character, to refine the token list. One of the tasks is to figure out which tokens are names of things (*identifiers*).

Separators

Some separators are used to group code fragments, and others are used to distinguish between fragments. The following nine characters are the separators:

```
{ } ( ) [ ] ; , .
```

The following code snippet employs all of these separators:

```
// Create an integer with the digits between the two indexes
// Assumes start < end. The result may be negative, but it
// is to be treated as an unsigned value.
private int parseInt(char[] source, int start, int end) {
    int result = Character.digit(source[start++], 10);
    if (result == -1)
        throw new NumberFormatException(new String(source));
     for (int index = start; index<end; index++) {
        int nextVal = Character.digit(source[index], 10);
        if (nextVal == -1)
           throw new NumberFormatException(new String(source));
        result = 10*result + nextVal;
         }
     return result;
     }
```

The braces, { and }, are the separators that group the largest sections of code. Braces are used to define blocks (see Chapter 3). The braces define the start and end of code for classes and methods. Braces can be nested. Parentheses, (and), are used for method parameters such as the method **parseInt** in the previous example.

Brackets, [and], are used for arrays, which are discussed later in this chapter. As seen earlier, the semicolon (;) is a statement terminator. The comma (,) is used to separate arguments in a method call, parameters in a method declaration, and declarations.

Identifiers

An *identifier* is the name of an item (such as a package, class, object, interface, method, variable, or constant) used in a program written in the Java language. Identifiers can use letters, numbers, dollar signs, and the underscore character (_). The letters can be anything defined by Unicode. That means you can declare your variables as half English and half Greek or all in Cyrillic.

Identifiers must follow these rules:

- Identifiers must start with a letter, underscore, or dollar sign.
- The remaining characters can be letters, numbers, dollar signs, and the underscore character.
- Identifiers can be any length (in practice, you'll rarely use more than a few dozen characters).

- You can use letters from multiple languages in one identifier.
- You can mix letters and numbers as long as the first character is a letter.
- Identifiers cannot be the same as reserved keywords (such as **int**, **long**, **class**, **true**, **false**, **null**, etc.).

The following are all legal identifiers:

numberOfNewShoppers

start12345zcounting

alphaBetaGamma___

We recommend two principles: be consistent, and use descriptive names. Make sure your names are self-documenting, and spell them out. If you wrote Fortran in the 1960s, the length of variable names was limited by the compiler or memory limitations. These days, there is no excuse for using cryptic identifiers. It is hard to read code in which the variable names don't follow an easily recognized pattern. Table 2.1 shows examples of common naming conventions used for various entities.

Table 2.1: Examples of entity names.

Type	Examples	General Rules
Packages	**java.lang.ref**, **com.coriolis.jutil**	Package names are lowercase; use reverse domain names for uniqueness.
Classes	**BigDecimal**	Each word begins with an uppercase letter; the first letter of the class name is capitalized.
Interfaces	**Checksum**	Same rules as classes.
Methods	**compareTo**	Method names are verbs with the first letter lowercase; then capitalize each subsequent word.
Variables	**keyBytes**	Variables are nouns with the first letter lowercase; then capitalize each subsequent word.
Constants	**BATCHSIZE**	Constants are all uppercase; words are separated by

Table 2.1: Examples of entity names.

Type	Examples	General Rules
		underscores.

Keywords Reserved by Java

You can't name your objects by the same name Java uses already. The following are keywords reserved by Java:

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

TEAMFLY

While you can't use the exact keywords in a name, you can create a name that incorporates a keyword, as long as it doesn't duplicate the reserved word. For example, you might write **returnTrue** or **return_ break**, both of which are legal identifiers.

Reusing Names

Problems can arise when you reuse the name of an existing object such as a class or variable declared previously. You need to be very careful about this. It is legal, but it can cause errors. For example, if you declare the variable **shopperCount** at the class level and then declare it again in a method within the same class, Java considers these to be two different variables and assigns them

Team-Fly®

two different memory locations even though they have the same spelling. We will cover *scope* (where objects can be accessed) in Chapter 4. For now, just be careful to avoid repeating an object name unless you have a good reason to do so (such as for method overloading).

Strings and Characters

A character literal is a single character enclosed in single quotes (for example, **'a'**). In addition to one character, you can also use special escape sequences to represent characters that are difficult to enter directly. In fact, you can use escape characters to enter any Unicode character by specifying its hexadecimal value (for example, \u0041 is a capital A). The following are examples of character literals:

```
'x' '$' '\n' '\\' '\'' '@' '_'
```

A string literal is composed of characters enclosed in double quotes. You can make an empty string (one with zero characters) by writing two double quotes together. The following are examples of string literals:

```
"This is a string literal."
"r"
"This is on the first line.\nThis is on the second line."
```

Two strings that have the same set of characters are not necessarily the same string object. However, the compiler will form one object for multiple string literals that contain the same characters. This makes it tricky to compare strings for equality. If two variables point to the same object in memory, then they are equal. If two string variables refer to different objects—even if both contain the same sequence of characters—they are not equal. Look at the following:

```
String firstString = "Patricia" ;
String secondString = new String("Patricia");
System.out.print(firstString == secondString ) ;
```

The two string variables—**firstString** and **secondString**—appear identical to a human, but to the computer they are quite different. The program will print **false** when run. That's because the compiler generates two separate string objects and the **==** operator tests to see if two object references refer to the same object—not if the objects appear to be the same to a human observer. Think of **firstString** and **secondString** as identical twins: No matter how much alike they look, they are still two separate entities. The compiler will try to identify the identical literal. Try changing the second line to this:

```
String secondString = "Patricia";
```

Now the compiler will recognize that the two strings are the same literal and generate a single object (causing the program to print **true**).

A few characters require special treatment if they are intended to be literal characters or part of a string. For example, you might want to include a quotation mark. Java allows you to use escape sequences (these begin with a backslash; see Table 2.2) to write special characters. The **\u** escape sequence allows you to enter any arbitrary Unicode character.

Table 2.2: The escape sequences in Java.

Escape	Sequence	Description
\b	Backspace	Positions the cursor one position back.
\n	Newline	Positions the cursor at the beginning of the next line.
\f	Form feed	Positions the cursor at the start of the next page.
\t	Horizontal tab	Moves the cursor to the next tab stop.
\r	Carriage return	Positions the cursor at the beginning of the current line without advancing to the next line. (On some platforms, this is the same as a newline character.)
****	Backslash	Prints a backslash character.
\”	Double quote	Prints a double-quote character.
\’	Single quote	Prints a single-quote character.

Variables

In Java, a *variable* is an item of data named by an identifier. Your program stores data in variables so it can manipulate the variables. Each variable has a name, a type, and a scope. The three rules for variable names are:

- A variable name must be an identifier that begins with a letter.
- A variable name cannot be a keyword, a Boolean literal (true or false), or the reserved word **null**.

- A variable name must be unique within its scope (covered in detail in Chapter 4).

Scope and the related idea of visibility are covered in Chapter 4.

Consider these Java statements:

```
int x;
float salary;
```

These statements declare two variables: **x**, which contains an integer, and **salary**, a floating-point number. These variables can hold values for use in computations. For example, you might write:

```
x = x * 128;
```

You'll see shortly that this statement will multiply **x** by 128 and put the result back in the **x** variable.

Declarations

When memory is first allocated for an object, the data it contains is unpredictable. This can cause bizarre behavior. Java prevents this type of trouble by initializing variables to safe default values. The main steps to using a variable in Java are:

- *Declaration*—Give the variable a name and a data type.
- *Instantiation*—Allocate memory for the object.
- *Initialization*—Assign the first value to the variable.

Declarations tell the compiler that you will be using a certain name to refer to a variable whose type is explicitly given. For basic data types (like **int**, for example), the compiler will reserve enough memory to hold the data. However, a declaration of an object reference type does not create an object; the declaration just adds the name to an internal list of names that Java knows will be holding objects, and it reserves enough space to hold a reference to an object. Instantiating an object allocates memory for it. For basic types, this happens automatically; for object types, your program must explicitly instantiate the object. Initializing places the first value in that object reference.

Java has a shortcut for initializing variables that use primitive data types. You can do all three steps simultaneously, like so:

```
int count = 243;
```

The **int count** portion declared the variable and instantiated it as well. The **= 243** portion initializes the new variable. We would say that this variable is initialized with an assignment statement upon declaration. Table 2.3 shows the instance and static variables' default values upon declaration.

Table 2.3: Default values for given data types.

Data Type	Instance's Default Initial Value
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000' (note: not a space (' \u0020') !)
boolean	**false**
object reference	**null**

When you create a variable, Java always initializes the variable to a default value as a safety measure. After you declare a variable, you can depend on this initial value without having to set it yourself manually.

Declaring a Final Variable

You can declare a variable to be **final**; a **final** variable is similar to a constant in other programming languages. You can't change the value of a **final** variable after initialization or assignment. You declare a **final** variable like this:

```
final int maxAge = 30;
```

You might think that you would have to initialize **maxAge** because it can't be changed. That was true in old versions of Java, but since the release of the 1.1 version of Java the rules have changed. You can leave a **final** variable uninitialized. If you do, you must assign a value to it before you use the variable, and you can only assign to it once. Multiple assignments will result in an compile-time error.

Using the new Operator

The **new** operator is used to declare objects (not primitive data types, but instantiated classes and arrays). Look at the following examples:

```
StringBuffer buf = new StringBuffer("This is like a string.");
appleObject apple = new appleObject(); // create a new apple
                                       // object
```

The word after the **new** operator specifies the type of object you want to create. The arguments in parentheses allow you to modify the object creation process (through something known as a *constructor*, which you'll read about in Chapter 4).

Strings are so common that Java allows you to use a shortcut to declare and initialize them like a basic type. For example:

```
String firstName = "Devyn";
```

This is exactly equivalent to writing:

```
String firstName = new String("Devyn");
```

Reference Data Types

The discussion about the **new** operator in the previous section implies that variables can be objects. While this appears to be true, it isn't. In Java, a variable can be a simple type (like an **int**) or it can be an object reference. In fact, objects (including strings and arrays) are never actually directly and uniquely named by an identifier. Instead, reference variables refer to the objects. This is a subtle distinction but an important one.

A good analogy is a phone number. You probably have a phone number (perhaps several). That phone number isn't really you, though; it just refers to you. If you move, someone else will get your phone number (and you might get another one). You might even have a regular phone number and a cell phone number. They both still refer to you. Consider the following code:

```
String myCodeName, codeName;
myCodeNam e=new String("Destroy");
codeName=myCodeName;
```

This code creates one **String** object (it contains the string **"Destroy"**). However, there are two reference variables (**myCodeName** and **codeName**). These variables now refer to the same object.

Operators

When you think of a computer program, you often think of computations. Operators are the mechanism that allows programs to perform computations on various values. There are three types of operators. A *unary* operator acts on one operand; a *binary* operator acts on two operands; and a *ternary* operator acts on three operands.

Operators tell Java to perform a task using one, two, or three values. For example, consider this bit of code:

```
a = b + c;
```

This statement uses two operators. The **+** operator tells Java to add variables **b** and **c**. The **=** operator puts the result into the **a** variable. Table 2.4 shows the 37 symbols that are Java operators.

Table 2.4: Java operators.

Operators	Type	Description
=	Binary	Assigns value on the right-hand side to the variable on the left-hand side
>, **<**, **<=**, **>=**	Binary	Less than, greater than, less than or equal to, and greater than or equal to
+, **-** , *****, **/**	Binary	Basic math operations (addition, subtraction, multiplication, division)
%	Binary	Remainder from integer division
&, **|**, **?**	Binary	Bitwise AND, OR, and exclusive OR
~	Unary	Bitwise NOT
&& , **||**	Binary	Logical AND, logical OR
!	Unary	Logical NOT
?:	Ternary	If-then-else operator

Table 2.4: Java operators.

Operators	Type	Description
<<, >>	Binary	Left shift, right shift
>>>	Binary	Unsigned right shift
==, !=	Binary	Test for equality or inequality
+=, - =, *=, /=, >>>=, >>=, <<=, %=, ?=	Binary	Performs indicated operation on the left-hand and right-hand expression, and then places the result back in the left-hand expression; for example, x+=10 adds 10 to x and places the result in x
++, - -	Unary	Increment or decrement

Math Operators

Most of the math operators in Java are familiar to everyone (**+**, **-**, *****, and **/** represent addition, subtraction, multiplication, and division). Keep in mind that dividing integers results in an integer. So 10/3—if both numbers are integers—results in 3. The **%** operator provides the remainder from a division, so 10%3 results in 1 because 10/3 leaves a remainder of 1.

The **–** operator is usually the binary subtraction operator, but it can also be a unary operator. For example, 10+-3 uses the unary **–** operator to negate the number 3.

It is very common to perform an operation on a variable and put the result back in the same variable. This is so common that Java provides a shortcut to simplify this operation. Consider this expression:

```
x=x+10;
```

You could also write this as:

```
x+=10;
```

Another common operation is to add (or subtract) 1 from a variable. Java provides a shortcut for this as well in the **++** and **–** operators. These operators have a special property because they are unary.

Prefix and Postfix

Consider these statements:

```
x = 100;
y = 5 + x++;
```

The **++** operator adds 1 to **x**. Because the **++** operator appears *after* the **x** (postfix), however, Java uses the original value of **x** in the expression. Therefore, after these statements are executed, **y** will contain 105, and **x** will contain 101. You can rewrite the code so that the **++** operator appears before the **x**(prefix) like this:

```
x = 100;
y = 5 + ++x;
```

After this code executes, **x** will still contain 101, but now **y** is 106.

Relational Operators

Relational operators are those that compare two values (for example, **==** and **<** are relational operators). These operators produce a **true** or **false** result. You could store these Boolean values in a variable of type **boolean**. However, usually you'll use these in **if** and similar statements (see the next chapter). For example:

```
if (x==10) System.out.println("X is 10");
```

You can change the sense of any Boolean value (including a relational operator) by using the unary **!** operator. This operator turns **true** into **false** and vice versa. So writing **a<b** is the same as writing **!(a>=b)**.

You can also join relational operators (or any Boolean expressions) by using the **&&** or **||** operators (**&&** is a logical AND; **||** is a logical OR). One point about these operators bears mentioning: These operators evaluate values from left to right and stop processing as soon as the result is clear.

Tip A common error is mixing up the equality operator (**==**) with an assignment operator (**=**). The equality expression is a test returning **true** or **false**. The assignment expression copies what is on the right to the left.

Testing for Equality

How do you test for equality? For the primitive data types, you use the **==** operator, like so:

```
int a = 1;
int b = 2;
int c = 1;
```

```
System.out.println(a==b); // returns false
System.out.println(a==c); // returns true
```

The **char** data type is treated as an integer internally, so it also uses the == operator. Strings and objects are more complicated. For example, if two different string variables contain the same sequence of characters, we say they are lexicographically equal. They hold equivalent strings, but these two string objects are held in two separate memory locations. Look at the string comparisons shown in Listing 2.3

Listing 2.3: This program demonstrates some string comparisons.

```
public class stringEquality
{
  public static void main(String [] args)
   {
     String a="Java Book";
      String b;
     b=new String("Java Book");
      System.out.println("(a==b)=" + (a==b));
      System.out.println("(a==(b+\"\"))=" + (a==(b+"")) );
     System.out.println("(a.equals(b))=" + a.equals(b));
      System.out.println("(a.equals(b+\"\"))=" + a.equals(b+""));
      System.out.println("((a+\"\")==(b))=" + ((a+"")==b));
   }
}
```

The code listing returns the following:

```
(a==b)=false
(a==(b+""))=false
```

(a.equals(b))=true

(a.equals(b+""))=true

((a+"")==(b))=false

You need to be careful with objects, especially strings. The equality operator is not intuitive here. The **==** operator simply tests to see if the object references on either side refer to the exact same object. If there are two objects involved (as there are in this case) the **==** operator returns **false** even if the strings contain the same characters. The **equals** method retrieves the contents and compares the letters; this is probably what you usually want to do when comparing strings or objects.

Tip Use the **equals** or **compareTo** methods when comparing strings unless you really want to test for object equality.

Operator Precedence

Java follows strict rules regarding which operators it processes first when there are multiple operators. These rules will be familiar to you if you remember your high-school math classes. You learned in math class that, by convention, you perform multiplication and division before addition and subtraction. Java uses this rule as well as others. Consider this series of statements:

```
x = 10;

y = 3;

z = x + y * 5;
```

TEAMFLY

Because multiplication occurs first, **z** will contain 25 (not 65). Table 2.5 shows Java's operator precedence listed in order; each row has priority over all rows beneath it. Operators on the same row have equal precedence and are performed in the order in which they appear in code (reading left to right).

Table 2.5: Operator precedence.

Name	Symbol
postfix operators	[] . (params) expr++ expr--
unary operators	++expr --expr +expr -expr ~ !
creation or cast	new (type)expr

Team-Fly®

Table 2.5: Operator precedence.

Name	Symbol
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	?
bitwise inclusive OR	|
logical AND	&&
logical OR	||
conditional	? :
assignment	= += -= *= /= %= &= ?= |= <<= >>= >>>=

Integer Bitwise and Boolean Logical Operators

The operators **&**, **?**, and **|** have two applications. When you use these operators with Boolean values, Java treats a **false** value as a 0 and a **true** value as a 1. Then it applies the logic in Table 2.6. If the result of the operation is a 1, the resulting value is **true**; otherwise, the result is **false**. If the arguments to the operators are integers, Java treats each value as a binary number and forms a new binary number using the algorithm in Table 2.6. You can probably deduce that the AND algorithm requires both values to be 1 to produce a 1. The OR algorithm returns a 1 if any input is a 1. The exclusive OR (XOR) algorithm is not so common. The XOR only returns a 1 when either input is a 1, but not both.

You can find an example of using **&** (and) on the numbers 2 and 3 in Figure 2.1. For clarity, I've shown only the least significant bits of the numbers.

Table 2.6: Boolean operations.

A	B	A & B (AND)	A ? B (XOR)	A | B (OR)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

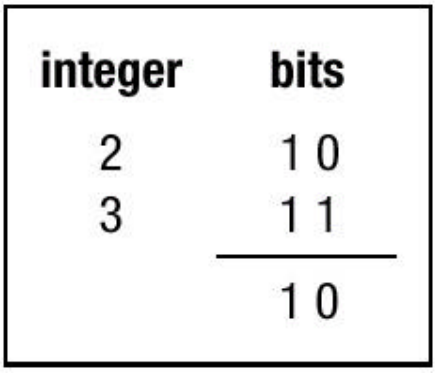

Figure 2.1: Result of using **&** on the numbers 2 and 3.

The first bit in the integer 2 is 1, as is the first bit in the integer 3. The expression **1 & 1** returns **true** or a **1**. The second column, **0 & 1**, returns **0**. The result is the integer 2. The following code demonstrates this:

```
System.out.println("2 & 3 = " + (2 & 3));
//Returns:
//2 & 3 = 2
```

Shift Operators

A *shift operator* moves bits in a binary number: All bits slide over a certain number of positions, determined by the second operand. The first operand bits are shifted right or left by the number of positions specified in the second operand. Shifting a number left has the effect of multiplying the

number by a power of two. So shifting a number left by one position will multiply the number by 2 (2^1). Shifting three places is the same as multiplying by 8 (2^3). Shifting right divides by a power of 2. Table 2.7 summarizes the shift operators.

Table 2.7: Shift operators.

Operator	Use	Action
<<	A << B	Shifts bits of A left by B positions.
>>	A >> B	Shifts bits of A right by B positions.
>>>	A >>> B	Shifts bits of A right by B positions (unsigned).

The right shift operator (**>>**) knows that the leftmost bit of an integer corresponds to its sign. Negative numbers always have a one in the leftmost bit, and positive numbers always have a zero in that position. The normal right shift operator (**>>**) preserves this bit so that dividing a negative number yields a negative result. If you want a true shift, you can use the unsigned shift operator (**>>>**).

The ShiftOperators.java program (see Listing 2.4) demonstrates how to use the shift operators.

Listing 2.4: Using the shift operators.

```
public class shiftOperators

{

  public static void main(String args[])

  {

    int v1=5, v2=125, r;

     int shift;

     for (shift=0;shift<8;shift++)

     {
```

```
        r=v1<<shift;
      System.out.println("5<<" + shift + " = " + r);
        r=v2>>shift;
      System.out.println("230>>" + shift + " = " + r);
      }
   }
}
```

We say, "a shift right 1" or "a is shifted right by 1." Notice that shifting by zero doesn't affect the number at all.

Assignment Operators

An *assignment operator* (**=**) places the result of what is on its right- hand side into the left-hand expression. It is very common to perform an operation on a variable and then reassign the result to the same result. For example, you might write:

```
x=x+10;
```

Because this practice is so common, Java allows you to combine this sort of operation into a shorthand statement (see Table 2.8). The previous statement can be written as:

```
x+=10;
```

Table 2.8: Assignment operators.

Operator	Use	Same As
+=	A += B	A = A + B
- =	A - = B	A = A - B
***=**	A *= B	A = A * B
/=	A /= B	A = A / B
%=	A %= B	A = A % B

Table 2.8: Assignment operators.

Operator	Use	Same As
&=	A &= B	A = A & B
|=	A |= B	A = A | B
?=	A ?= B	A = A ? B
<<=	A <<= B	A = A << B
>>=	A >>= B	A = A >> B
>>>=	A >>>= B	A = A >>> B

The Conditional Operator ?:

The **?:** operator is similar to the **if-else** statement. The **?:** operator returns B if A is true or returns C if A is false. Chapter 3 describes the **if- else** statement in some detail, but for a moment look at this example:

```
if (a > b) {
  maxValue = 100;
}
else {
  maxValue = 200;
}
```

Java provides a nice shorthand way of doing the same thing, like this:

```
maxValue = (a > b) ? 100 : 200;
```

This is the only operator that accepts three arguments, and it is sometimes called the *ternary operator* for this reason.

Arrays

An *array* is a structure that holds multiple values of the same type. In Java, arrays are really a form of object. You use the square brackets to indicate an array variable. The square brackets also appear

after the **new** keyword to specify the size of the array. In the following example, the array has 10 elements ranging from **anArrayofIntegers[0]** to **anArrayofIntegers[9]**. For example:

```
int[] anArrayOfIntegers = new int[10]; // create an array of
                                       // integers
```

Just as strings have a shortcut initializer, arrays also have a special syntax you can use. Here's an example:

```
char[] idArray = { 'a', '4', '-', '3', '3', 'b', 'x' };
```

This array has seven elements, as you'd expect.

Arrays are actually objects, and therefore they can have fields. The main field you will use for arrays is **length**. This field returns the number of elements in the array.

Arrays can even contain other arrays; this capability allows you to create arrays of many dimensions. For example, here's a three-dimensional string array:

```
String[][] javaKeyWords =
{
  { "abstract", "default", "if", "private", "this" },
  { "boolean", "do", "implements", "protected", "throw" },
  { "break", "double", "import", "public", "throws" }
};
```

You can copy elements from one array to another, like so:

```
char[] copyFrom = { 'b', 'l', 'a', 'c', 'k', 'b', 'o', 'o', 'k', };
char[] copyTo = new char[5];
System.arraycopy(copyFrom, 0, copyTo, 0, 5);
String blackWord = new String(copyTo);
```

In this example, the **blackWord** string contains the string "black." An element within an array can be accessed by its index. The snippet **copyTo[3]** will return the character "c" (an array index is zero based, so this is the fourth element). The primary rules for arrays are:

- The first array element is index 0. The last element is the array length–1.

- Once you create an array, you can't directly change its size.
- For arrays of primitive data types, each element gets the default value when the array is created. Object arrays are initialized with **null** (a special value that marks a reference variable that doesn't refer to anything).
- You can precede the array name with [] in the declaration. Alternatively, you can use the [] as a suffix.
- [][] declares a two-dimensional array. The number of brackets determines how many dimensions are used. Technically, this forms an array of arrays.
- Arrays are objects (this is true even if the array's contents are simple data types, such as integers).

Casting

When you have a variable of one type, it is often useful to convert it temporarily into a different (yet compatible) type. For example, suppose you are computing the average of several integers. You add the integers together in the **total** variable and you count the integers in the **count** variable. Then you might write:

```
float avg;
avg=total/count;
System.out.println(avg);
```

The problem is that the right-hand side of the assignment contains only integers. Therefore, Java performs integer division. If **total** is 125 and **count** is 99, **avg** will contain 1.0; this is technically correct, but probably not what you expected.

Of course, you could assign **total** or **count** to a temporary floating- point variable, but it is more efficient to use a cast. To cast a value to a different type, put the type's name in parenthesis preceding the expression. You could therefore rewrite the previous example to read:

```
float avg;
avg=total/(float)count;
System.out.println(avg);
```

Converting either variable to **float** will cause the division to use floating point and will yield a more correct result. You can't cast to any type; it has to be a compatible type. For example, trying to cast **count** to a **String** will cause a compile-time error.

Immediate Solutions

Declaring Variables and Arrays

In Java, every variable and every expression have a data type known as *compile time*. Java is a strongly typed language. Data typing defines what kind of values a variable can contain. The data type restricts both the variable's values and the operations that can be performed on it.

Data types fall into two broad categories: primitive (such as **int**) or reference (these refer to classes such as **String**). The primitive data types appear in Table 2.9.

Table 2.9: Primitive data types.

Data Type	Range	Description
boolean	**true/false**	True or false value
char	Unicode characters	Single character
byte	–128 to +127	8-bit integer
short	–32,768 to +32,767	16-bit integer
integer	±2?31 – 1	32-bit integer
long	(±2?63 – 1)	64-bit integer
float	± 3.4028235E38	Real number
double	± 1.7976931348623157E308	Double-precision floating-point number

You can easily declare a variable by naming a type and then listing one or more variable names:

```
int myCount;
int ramSize, diskDrives;
```

Declaring arrays requires that you use square brackets. You can place them after the type name or after the variable name; it doesn't matter which you choose. Here's an example of an integer array:

```
int [] numList;
```

You could write this as:

```
int numList[];
```

In Java, arrays are technically objects, so to create the array you must use the **new** operator and provide the size of the array. Here's how you'd make an array of three integers and set the first and last elements:

```
int primes[] = new int[3];
primes[0]=1;
primes[2]=5;
```

Using Literals

A *literal* is what you "literally" type to represent a value of a primitive data type or a string. For example, if you type the number 49.2 in your code somewhere, then this number is a literal. The form of the literal determines its data type (see Table 2.10).

Table 2.10: List of literal data types.

Data Type	Literal	Bits	Range
integer			±2?31–1 (except byte, short)
byte	3	8	–128 to +127
short	461	16	–32,768 to +32,767
decimal	243	32	–2147483648 to +2147483647
hexadecimal	0x59fff000	32	0x7fffffff
octal	03228	32	017777777777

Table 2.10: List of literal data types.

Data Type	Literal	Bits	Range
long integer			(±2?63–1)
decimal	544297L	64	–9223372036854775808L to +9223372036854775807L
hexadecimal	0x3AAA89L	64	0x7fffffffffffffffL
octal	087897L	64	0777777777777777777777L
float	3.14f or 5.827e+23f	32	± 3.4028235E38
double	3.14 or 5.827e+23	64	± 1.7976931348623157E308
boolean	true	1	true or false
char	'b'	16	One character
String	"Thank you!"	0–**8**	empty–unlimited length

TEAMFLY

Listing 2.5 shows a program that demonstrates how to get the maximum and minimum values for Java's numerical data types.

Listing 2.5: Getting the maximum and minimum values for Java's numerical data types.

```
public class DataTypeMinMax{

    public static void main(String[] args) {

        byte maxByte = Byte.MAX_VALUE;

        byte minByte = Byte.MIN_VALUE;

       short maxShort = Short.MAX_VALUE;

        short minShort = Short.MIN_VALUE;
```


Team-Fly®

```
        int maxInteger = Integer.MAX_VALUE;
        int minInteger = Integer.MIN_VALUE;
        long maxLong = Long.MAX_VALUE;
        long minLong = Long.MIN_VALUE;
        float maxFloat = Float.MAX_VALUE;
        float minFloat = Float.MIN_VALUE;
        double maxDouble = Double.MAX_VALUE;
        double minDouble = Double.MIN_VALUE;
        System.out.println("maxByte = " + maxByte);
        System.out.println("minByte = " + minByte);
        System.out.println("maxShort = " + maxShort);
        System.out.println("minShort = " + minShort);
        System.out.println("maxInteger = " + maxInteger);
        System.out.println("minInteger = " + minInteger);
        System.out.println("maxLong = " + maxLong);
        System.out.println("minLong = " + minLong);
        System.out.println("maxFloat = " + maxFloat);
        System.out.println("minFloat = " + minFloat);
        System.out.println("maxDouble = " + maxDouble);
        System.out.println("minDouble = " + minDouble);
    }
}
```

Using Conditional Operators

The conditional AND (**&&**) and conditional OR operators have "conditional" in their names because the compiler evaluates the first operand but may or may not evaluate the second operand. The second operation is conditional because it depends on the result of the first evaluation. With the **&&** operator, the second operand is not evaluated if the first result is **false**. This makes sense because

the result could not possibly be **true**. The second value is immaterial. Similarly, the second operand is not evaluated if the OR (**||**) operator finds a **true** result in the first operand. In this case, a **false** result is not possible.

Listing 2.6 demonstrates the advantage of using conditional operators.

Listing 2.6: A conditional operator demonstration program.

```
import java.util.Random;


 public class SkipStepsWithConditionalOperators {

   public static void main(String args[])

    {

      Random randomGenerator = new Random();

     boolean a = true, b = true;

      int count = 0, actualEvals = 0,

           totalEvals = 20, skippedEvals = 0;


      for (int i = 1; i <= 10; i++){

           a = randomGenerator.nextBoolean();

           b = randomGenerator.nextBoolean();

           count = 1;

            ++actualEvals;


          if (a) { ++count; ++actualEvals;}


        System.out.println("a=" + a + "; b=" + b +

                                "; a && b = " + (a && b) +

                                 "; " + count + " evaluations");
```

```
    }

        skippedEvals = totalEvals-actualEvals;

         System.out.println("\nSteps saved using Conditional " +

                            Operators: " + skippedEvals + " (" +

                               skippedEvals * 100 /totalEvals + "%)");

  }

}


//returns:

//a=false; b=false; a && b = false; 1 evaluations

//a=false; b=false; a && b = false; 1 evaluations

//a=false; b=true; a && b = false; 1 evaluations

//a=false; b=true; a && b = false; 1 evaluations

//a=false; b=false; a && b = false; 1 evaluations

//a=true; b=true; a && b = true; 2 evaluations

//a=false; b=true; a && b = false; 1 evaluations

//a=true; b=true; a && b = true; 2 evaluations

//a=true; b=true; a && b = true; 2 evaluations

//a=false; b=false; a && b = false; 1 evaluations

Steps saved using Conditional Operators: 7 (35%)
```

You'll read about the random generator later in the book. For now, focus on the highlighted lines. These lines make up a single statement that conditionally tests **a** and **b** for true values. They must both be **true** for the expression to return **true**. Therefore, if **a** is **false**, there is no need to evaluate **b**. With the ordinary OR operator (**|**), the situation is reversed. If **a** is **true**, there is no reason to continue. The exclusive OR operator (?) always evaluates both operands (there is no ?? operator).

If you try to apply a Boolean operator directly to a data type other than **boolean**, you will get a compile-time error.

Related solution:	Found on page:
Replacing **if** with the Conditional Operator **?**:	69

Using the Boolean NOT Operator

The Boolean NOT operator reverses a Boolean value. If the operand is **true**, then **false** is returned. If the operand is **false**, then **true** is returned. The NOT operator is often combined with other operators (such as **!=,** which is "not equal to") and used to reverse the outcome of expressions, like so:

```
Boolean isCustomerHappy = !(highPrice | lateDelivery );
```

This statement defines customer satisfaction by two conditions. If the price is too high or he was overcharged, then the customer will not be happy. If either of these conditions is true, the expression between the parentheses will return **true**. If one of the two conditions is **true**, we want to show that the customer is unhappy, so we use the NOT operator to set the **isCustomerHappy** flag to **false**.

Tip The NOT (**!**) operator has a high precedence, so parentheses are often needed to make a statement correct.

Using the Complement Operator

The NOT operator (**!**) reverses a single Boolean value. It is easy to confuse NOT with the *complement operator* (**~**). The difference between these two operators is that the NOT operator works with Boolean values. The complement operator works on binary numbers, inverting each bit in the value. The following program demonstrates both the NOT and complement operators:

```
import java.util.Random;


public class NotAndComplimentOperators {
  public static void main(String args[])
  {
     Random randomGenerator = new Random();
      boolean randomBoolean = true;
     int randomInteger = 0;


    for (int i = 1; i <= 3; i++){
          randomBoolean = randomGenerator.nextBoolean();
          randomInteger =randomGenerator.nextInt();


          System.out.println("randomBoolean=" + randomBoolean +
```

```
                                "; !randomBoolean=" + !randomBoolean );
            System.out.println("randomInteger=" + randomInteger +
                                " ~randomInteger = " + ~randomInteger);
        }
    }
}
//returns:
//randomBoolean=false; !randomBoolean=true
//randomInteger=1595993021 ~randomInteger = -1595993022
//randomBoolean=true; !randomBoolean=false
//randomInteger=-697145322 ~randomInteger = 697145321
//randomBoolean=true; !randomBoolean=false
//randomInteger=1911563148 ~randomInteger = -1911563149
```

In this program, the Boolean value simply gets inverted. Look at the integers, however. The complement operator inverts every bit in the integer; this has the effect of changing the sign and incrementing the result by one. If the operand was positive, then the result is equivalent to "add one and negate." If the operand was negative, the result is "take away one and remove the negative sign." The complement operator works on all integer types.

Mixing Data Types

The need to mix data types is common in programming. Java performs data type conversions when you mix the data types in the same statement. Java tries to avoid losing information, so it converts all variables and expressions to the type that is the largest (most bits) and most precise type in the statement. For example, if you have five variables, of which four are integers and one is **long**, the result will be a **long** type:

```
int firstInteger = 1;
int secondInteger = 1;
int thirdInteger = 1;
int fourthInteger = 1;
long firstLong = 10L;

//mixing data types to show biggest wins
System.out.print(firstInteger+secondInteger+thirdInteger+
                 fourthInteger+firstLong);
```

The previous example prints "14" to the console. However, if you try to do the following:

fourthInteger = firstInteger+secondInteger+thirdInteger+firstLong;
the compiler will complain because you tried to place a **long** into an **int**. The following does work, however:

firstLong = firstInteger+secondInteger+thirdInteger+firstLong;

You can read more about *casting* (explicitly changing a data type) in the next section. For now, observe that the data type precedence for arithmetic operations (and most others, if possible) should be read left to right where the resulting data type is the leftmost type in the following list: **double**, **float**, **long**, **int**, **short**, **byte**.

Tip The expression **+count** promotes **count** to **int** if it's a **byte**, **short**, or **char**; however, **-count** arithmetically negates **count** regardless of its data type.

Performing Casts and Promotions

When you convert one data type to another, there is always a risk that information will be lost. You can't hope to change a **long** to an **int** correctly because you would be trying to stuff 64 bits of data into a 32-bit container. Going from a bigger data type to a smaller one is known as a *narrowing* conversion. Going from a smaller data type to a larger one is known as a *widening* conversion. Generally, widening works fine, but narrowing loses information.

If widening is required (multiplying an **int** by a **float**, for example), Java handles the conversion automatically. Java does what is called an *implicit conversion*. When you go the other way, Java does not automatically *narrow* the data for you to avoid losing information. You have to use a *cast* to perform a narrowing conversion. There are two types of casts:

- *Implicit*—A cast that Java performs automatically.
- *Explicit*—A programmer-directed conversion.

Casting is used to force an explicit conversion of data from one type to another. It requires you to use the type name in parentheses, like this:

(type)variableName

Java then takes the value of the variable (or expression) and converts it to the data type you specified in the cast. So, the expression 100*5 returns an integer, but the following returns a **long**: (long)(100 * 5). This is an example of a *widening conversion*. It is convenient to let Java do this automatically, but it is usually better to explicitly cast an expression if you know the resulting data type requirement. Explicit casting prevents future problems when you change the expression to include data types that, if left to Java's implicit conversion, would result in an undesirable data type.

The following is a widening conversion scale for numbers. Converting left to right will happen automatically (implicitly), but converting right to left requires explicit casting:

byte **?** short or char **?** int **?** long **?** float **?** double

This code shows that the widening primitive conversions are:

- **byte** to **short**,**int**,**long**,**float**, or **double**
- **short** to **int**, **long**, **float**, or **double**
- **char** to **int**, **long**, **float**, or **double**
- **int** to **long**, **float**, or **double**
- **long** to **float** or **double**
- **float** to **double**

All the previous conversions can be done by Java implicitly. What about narrowing conversions? The following is an example of a narrowing cast:

```
float a =3.583453f;
int b=0;
b = (int)a;
```

The following is an example of an implicit widening conversion:

```
float a =0f;
int b=2390;
a = b; //no cast necessary
```

Whenever you perform math on dissimilar data types, Java will always perform a widening conversion. You have to explicitly cast to override this like so:

```
byte maxByte = Byte.MAX_VALUE;
System.out.println( (byte)(100 * maxByte) );
```

Without the explicit byte cast, Java would return an integer because the 100 is an integer and Java promotes the byte to an integer. You will get a compile-time error if you try to perform a narrowing conversion, say of a **long** to an **int**, without the cast operator. For example, if you try to put the literal 12L into an integer variable (for example, **i**), Java will complain:

```
castingTest.java:3: possible loss of precision
found : long
required: int
int i = 12L;
        ^
1 error
```

Of course, removing the L from the literal will solve this problem. You could also cast to an **int**. Be careful when mixing numbers and strings. You should not rely on Java's implicit casting unless it is a trivial situation. We recommend avoiding any math directly in a string construction. Do the math first; then cast, as in the following:

```
int morningCustomers = 428, lunchCustomers = 342;
int customerCount = morningCustomers + lunchCustomers;
String customerCountDisplay = "We did business with " +
                              customerCount + " customers today!";
```

Sun's documentation states, "Despite the fact that overflow, underflow, or other loss of information may occur, narrowing conversions among primitive types never result in a run-time exception." The phrase "loss of information," though, means that you can't predict what is going to happen. For example, if you assign a **float** value that is too large to fit into an **integer** variable, Java returns the maximum **int** value. However, a **double** value that is too big for a **float** variable returns infinity.

Tip Generally, you can't cast primitive types to or from **reference** or **boolean** data types.

Chapter 3: Blocks and Statements

In Brief

In the previous chapter, we looked at the foundation of the Java language. You learned how to create variables and write expressions. Real programs typically don't perform a fixed set of calculations, however; rather, programs repeat steps or make decisions that control what steps to perform. Real programs join multiple statements together into blocks. *Blocks* allow you to define pieces of your program that you can then treat as a unit. For example, you might execute a certain piece of your program only if some condition is true. You might want to execute the same set of steps for each record in a database. Blocks make this sort of manipulation possible.

Expressions, Statements, and Blocks

By using literals, variables, and operators, you create expressions. An *expression* is the smallest piece of code that is executed and returns a value. The *statement* is the equivalent of a sentence in Java. Not all statements return a value. A variable declaration is an example of a statement that doesn't return a value.

Some statements aren't expressions. Instead, they control how Java executes other statements. You've already seen the **if** statement in other examples. It evaluates a Boolean expression. If the expression is **true**, the **if** statement executes the code it controls; otherwise, that code does not execute.

By default, the **if** statement (and most other control statements) affects only one statement. With that in mind, consider the following three lines of code:

```
x=x+1;
if (x==100) x=0;
System.out.println(x);
```

The lines before and after the **if** statement will execute, no matter what. However, the **x=0** statement executes only when **x** is equal to 100.

Of course, you often want to do a series of statements when some condition is true. You can group statements together with curly braces, { and }. All blocks begin with an open brace and end with a matching closing brace. You can nest blocks—that is, put one block inside another. Here's an example of *nesting*:

```
if (x==100)
{
    int someValue;
    someValue=y/100;
```

```
    x=someValue+35;
    y=y+1;
   if (y==100)
       {
          int tmpval;
           tmpval=9*z/5+32-someValue;
           y=tmpval/2+y;
       }
    System.out.println(x);
}
```

The braces do two things. First, they define *scope* (the accessibility of variables). Variables declared within braces are accessible only within that block (and any nested blocks). In the previous example, all the code within the outer **if** statement can refer to the **someValue** variable. However, only the code within the inner **if** statement can refer to the **tmpval** variable. Outside the inner curly braces, **tmpval** is not defined. The second function of the braces, of course, is to group multiple statements together so that each **if** statement controls multiple statements.

In many contexts, you must use braces. For example, the definition of a class or method requires braces. However, some statements use braces to define a block of code that the compiler treats as a single statement. For example, consider this Java code:

TEAMFLY

```
if (x==10)
{
     f(x);
}
```

The braces, in this case, are optional because there is just one line in the body of the **if** statement. You could write this as:

```
if (x==10) f(x);
```

Because white space is irrelevant, you can also indent the one-line block to make it clearer that it is a block:

```
if (x==10)
    f(x);
```


Team-Fly®

Note This notation is succinct. However, many programmers shy away from this style, and you might, too, because you might later add more statements before or after the ***f(x)*** statement. If you indent these new statements to line up with the original one, you might think you've created a multi-statement block. However, the compiler doesn't care how you lined it up; without the braces, the ***if*** statement applies only to the next statement.

Sequencing

In addition to **if**, there are many other statements that control execution. What's more, you can create a block using curly braces anywhere (although it is typically bad form to do so). Normally, a block begins execution at the beginning and—as you'd expect—terminates execution at the end of the braces.

Two events, however, can cause execution of a block to stop before reaching the end of the block:

- With certain types of loops, you can use the **break** or **continue** statements to force an early end to the loop (see the Immediate Solutions for more information about these statements).
- An exception can cause a block to stop processing; when this happens, Java searches for an appropriate exception handler. Exceptions occur when something happens that requires exceptional processing. For example, if your program attempts to divide by zero (an illegal operation), Java will raise an exception.

Java might signal an exception for many reasons. For example, when you divide by zero, Java causes an exception to occur. Consider the simple program in Listing 3.1.

Listing 3.1: A program that generates an exception.

```
public class ZException

{


  void go(int denom)

  {

     System.out.println(100/denom);

  }

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

```
    {
      new ZException().go(0);
    }
}
```

This causes a divide-by-zero exception:

```
java.lang.ArithmeticException: / by zero
        at ZException.go(ZException.java:6)
        at ZException.main(ZException.java:11)
Exception in thread "main"
```

This exception terminates your program. What if you want to print 0 if the input to the **go** method is zero? You can, of course, simply test for this particular condition with an **if** statement. However, exception handling provides a different technique, which has advantages you'll see soon.

Consider the program in Listing 3.2. It also generates an exception, but now the program uses the **try** keyword to detect exceptions in the block of code in the **go** method. When an exception occurs, execution resumes at the **catch** keyword. Actually, each **try** statement can have multiple **catch** keywords. Execution will continue at the **catch** keyword that matches the type of thrown exception (more on that later).

Listing 3.2: This program catches exceptions that occur.

```
public class ZException
{

  void go(int denom)
    {
      try
      {
        System.out.println(100/denom);
      }
```

```
      catch (Exception e)
      {
         System.out.println(0);
      }
   }

   public static void main(String args[])
   {
      new ZException().go(0);
   }
}
```

As it stands, using the code in Listing 3.2 isn't much different from using an **if** statement to detect a zero argument. However, the exception handling protects any amount of code and—if written correctly—detects any error condition, not just the ones you've planned for. If you wanted to handle only math errors (like the divide-by-zero error), you could replace the **Exception** class name in the **catch** statement with **ArithmeticException**.

In fact, you can use multiple **catch** statements to handle different types of exceptions. The **Exception** type is the most general and should appear last because Java processes **catch** blocks in order. If you put a more general exception (like **Exception**) before a specific exception (like **ArithmeticException**), the compiler will complain. Consider the example in Listing 3.3.

Listing 3.3: This code handles multiple exceptions differently.

```
public class ZE xception
{

   void go(int denom)
   {
      try
```

```
    {
       System.out.println(100/denom);
    }
    catch (ArithmeticException e)
    {
       System.out.println(0);
    }
    catch (Exception e1)
    {
      System.out.println("Unexpected exception occurred");
    }
  }
  public static void main(String args[])
  {
    new ZException().go(0);
  }
}
```

What's really interesting about exceptions is that they can be handled in any calling routine. Java searches the current block to find an exception handler. If one isn't found, Java begins searching other active blocks. In other words, because **main** calls **go**, **main** could provide some or all exception handlers for the code. This is useful when **go** knows that an error occurred but doesn't know what to do about it. With exceptions, each caller can specify its own error-handling code. Listing 3.4 shows an example. The **go** method could continue to maintain some exception handlers if you wanted to handle some errors without sending them to the caller. Exceptions handled in the **go** routine would not propagate to the **main** method.

Listing 3.4: Calling routines can handle exceptions.

```
public class ZException
{
  void go(int denom)
```

```
 {
   System.out.println(100/denom);
}
public static void main(String args[])
{
 try
   {
    new ZException().go(0);

   }
   catch (ArithmeticException e)
   {

    System.out.println(0);
   }
 }

}
```

It is possible to create your own exceptions for custom conditions. To do this, however, you need to create your own objects (derived from **Exception**)—something we haven't covered yet. Then you can use **throw** to generate the exception.

Types of Exceptions

The divide-by-zero exception is an example of an unchecked exception. You are not required to handle these exceptions (although, as you've seen, you can). If an unchecked exception occurs and you don't handle it, your program terminates.

Many exceptions are checked exceptions, however. A method that performs operations that might generate a checked exception must handle it in one of two ways. First, the method can provide the appropriate **try** and **catch** blocks that will handle the exception. Second, the method can specify the exception in the **throws** statement for the method. This tells the compiler that the method's callers must handle the exception (or delegate it to their callers).

For example, suppose you are writing a method named **go**. It will call another method called **work**. The **work** method might throw a checked exception called **CustomException**. If the **go** method doesn't specify a matching exception handler, it must specify the exception name in its **throws** clause:

```
void go() throws CustomException
{
    . . .
 work();
  . . .
}
```

Block Cleanup

Consider this example:

```
if (using_a_file == true)
{
   open_a_file();
   process_data();
   close_the_file();
}
```

This block does fine when everything works normally. If **process_data** throws an exception, however, **close_the_file** never executes. Presumably, this leaves the file open, and that isn't a good idea.

The answer to this problem is to use a **finally** block (which you must use with a **try** statement). The code in a **finally** block executes when the main block ends for any reason (including when no exceptions occur). Listing 3.5 shows an example.

Listing 3.5: The finally block executes at the end of its main block in all cases.

```
public class ZException
```

```
{
 void go(int denom)
 {
   try
   {
    System.out.println(100/denom);
   }
    catch (ArithmeticException e)
   {

      System.out.println(0);
   }
   finally
   {
     System.out.println("Go complete");
   }

}

public static void main(String args[])
{
  ZException obj = new ZException();
  obj.go(0);
  obj.go(50);
```

```
 }


}
```

Style Guidelines

Because Java is white-space-independent, you have a lot of leeway in formatting your source code. Four major elements affect the readability of your code, although they are not pertinent to the program's execution. First, spacing between lines can help distinguish code blocks. Second, the position and alignment of braces can help readers follow the structure of your program. Third, since white space isn't important to the compiler, you can indent your code to show what parts of the program go together. Finally, you can use spaces (including new lines) to separate portions of statements and make them more readable.

Lines and Line Spacing

Each line of code should fit on a printed page; this guideline gives you about 80 characters per line. Most people prefer to read code with lots of spaces instead of code that is jammed together with minimal white space. Consider the difference that proper line spacing and self- descriptive names make in the example that follows at the top of the next page. The first two lines are valid Java but are hard to read. The same code is rewritten below those first two lines to illustrate the preferred way to write code.

```
int i=5,j=97,n=40; // avoid!
for(i =0;i<j;i++){j--;i++;n+=i+j;++n;}


int count = 5; // preferred
int colorFlag = 97;
int customerNumber = 40;


for(count = 0; count < colorFlag; count++)
{
 colorFlag–;
 count++;
 customerNumber += count + colorFlag;
 ++customerNumber;
}
```

Braces and Indentation

Some programmers place the starting brace at the end of the conditional statement or on the next line directly under the first letter of the keyword. It is easier to find the start and end of a block by finding braces that line up vertically. The eyes just work better that way. On the other hand, most braces in most of the world's source code (for example, the Sun documentation) start at the end of the first line rather than underneath on the second line. This book generally uses our personal preference—brace underneath.

Whatever brace style you use, please be consistent. Use one of these:

```
if (expression) { // popular – hard to match braces.
  statement;
}
```

or:

```
if (expression) // better – easier to match braces!
{
   statement;
}
```

Some people like to line up the braces with the code they surround. For example:

```
if (expression) // better – easier to match braces!
{
   statement;
}
```

Again, this is a matter of personal preference.

Indenting code makes it easier to read. The idea is to indent the same amount for all statements that are in the same block. How many spaces you indent isn't important, but all the code in a block should line up. Many text editors will automatically indent with spaces or tabs for you.

Spaces

The spaces affect the readability quite a bit. Which of the two following code snippets do you prefer?

```
int count=5,colorFlag=97,customerNumber=40;
customerNumber= customerNumber*colorFlag+(count+count*32);
```

```
int count = 5, colorFlag = 97, customerNumber = 40;
customerNumber = customerNumber * colorFlag +
```

```
(count + count * 32);
```

Even though the only difference between the two is spacing, the second is obviously better. Use lots of spaces.

Immediate Solutions

Using Statements

A statement is the combination of variables, operators, and meth od calls in one Java sentence that represents a complete unit of execution. Table 3.1 shows the common Java statements you'll encounter.

Table 3.1: Common Java statements.

Type	Example
Empty statement	;
Assignment statement	**carPrice = 38933.72;**
Expression statement	**++counter;**
Method call statement	**System.out.print("Welcome" + employeeName);**
Object creation statement	**Integer customerCalls = new Integer(78);**
Declaration statement	**String helpMessage = "Pardon for the delay, but you goofed.";**
Block statement	**{statements}**
if statement	**if (testCase == maxLimit) statement;**
for statement	**for (assignment/declaration; expression; expression) statement;**
switch statement	**switch (*expression*) { *statements*}**

TEAMFLY

Team-Fly®

Table 3.1: Common Java statements.

Type	Example
while statement	**while(*expression*) *statement;***
do...while statement	**do *statement* while (*expression*)**

Defining Blocks

Statements can contain expressions and even other statements. A *block* is defined as a bunch of statements and local-variable declaration statements within curly braces, { and }. Blocks have the following form:

```
{

 statement_1;
 statement_2;
 statement_3;
}
```

Using the Empty Statement

The empty statement does nothing. It is useful when you want a placeholder for a construct that requires a statement, but you aren't ready to write the code yet. For example:

```
if (count > maxUsers) ; // what to do?
```

Defining Scope

The scope defines where, in a program, you can refer to a variable — in other words, its *accessibility*. Also, the scope defines when the variable is created and destroyed. The location of a variable's declaration defines the variable's scope.

Java defines four categories of scope:

- *Member variable*—Declared within a class but not within the body of a method.
- *Local variable*—Declared within a block.
- *Method parameter*—Used to pass values to a method.
- *Exception-handler parameter*—Specifies the type of exception for an exception handler and also provides a parameter for the exception-handling code.

The scope of a variable is anything to the right of the declaration (including other declarations on the same line) and down within the rest of the block (including all nested blocks) in which it was declared.

In Chapter 4 you'll see that another consideration is a variable's (and a class's) visibility. This affects how other classes can interact with the variables you declare.

Related solution:	Found on page:
Defining and Using Classes	102

Using the if Statement

The **if** statement executes a set of statements if a Boolean **expression** is true. The **if** statement has the following forms:

```
//Simple if:
if (condition)
{
    statements;
}
//if-else: one statement or the other.
if(condition)
{
    statement(s);
}else
{

    statement(s);
}
//if-else if-else: Serially test executing only one condition.
if(condition_1)
{
    statement(s);
} else if(condition_2)
{
    statement(s)
}else if(condition_3)
{
    statement(s);
}else
```

```
{
      statement(s);
}
```

The following program is an adaptation of Sun's random number generator. There is one **if-else** statement that tests for a user-provided seed number for the generator. If there isn't a seed number, then the program generates its own seed as an integer (a **long**) based on the system time in milliseconds (the number of them since January 1, 1970, 00:00:00):

```
import java.util.*;

public class simpleRandomIntegerGenerator
{
  public static void main(String[] args)
   {
    long multiplier = 0x5DEECE66DL;
    long addend = 0xBL;
    long mask = (1L << 48) - 1;
     long seedInitializer = 0L;

    if (args.length > 0)
     {
       String seedString = args[0]; //command line-input
      seedInitializer = Long.parseLong(seedString.trim());
     } else
     {
      Date date = new Date();
       seedInitializer = date.getTime();
     }
      long seed=(seedInitializer*multiplier+addend) & mask;
    int randomInteger = Math.abs( (int)(seed >>> (16)) );

    String output = "A random integer: " + randomInteger;
     System.out.println( output );
     System.out.println( "seed " + seedInitializer );
      }
}
// java simpleRandomIntegerGenerator 87658
```

```
//returns:
// A random ineteger: 633419283
// seed 87658
```

Replacing if with the Conditional Operator?:

It is very common to use an **if** statement to set a variable to one of two values based on some logical expression. For example:

```
if (firstValue < secondValue)
{
  minimumValue = firstValue;
} else
{
  minimumValue = secondValue;
}
```

This construction is used so often that Java has a shortcut notation for it. The shortcut is called the *conditional operator*, having the form:

```
answer = (expression) ? returnThisIfTrue : returnThisIfFalse;
```

The expression must return a **boolean** value. The **returnThisIfTrue** and **returnThisIfFalse** expressions must yield a value of the same data type as the **answer** variable. We can replace the previous **if** statement with a conditional operator:

```
minimumValue =
 (firstValue < secondValue) ? firstValue : secondValue;
```

The following example is adapted from Java's **Short** class. A string is converted to a **short** integer by using one procedure for a positive number and another procedure for a negative number:

```
shortValue = negative ?
                new Short((short)-inputString.shortValue()) :
                new Short((short)inputString.shortValue());
```

Using the switch Statement

The **switch** statement evaluates an expression and attempts to match it with any number of **case** statements. When Java finds a matching **case** statement, execution begins at that point. Execution continues until the end of the **switch** statement or until a **break** occurs. (You'll find more about the **break** statement later in the "Using the **break** Statement" section). Without the **break** statement, Java continues executing even if it encounters more **case** statements. If no **case** statements match, you can add a **default** label to specify code to execute. The **default** label is really a special form of **case** statement that matches anything.

A **switch** statement has the following form:

```
switch (expression)
```

```
{
case ABC:
        statements;
    /* falls through to "DEF" statements */
case DEF:
       statements;
      break; // prevents falling through
default: // executes when no other case matches
       statements;
     break; // not strictly necessary
}
```

You can use a **switch** statement to test for a valid character within a string. Do this by using a switch-based test to parse a token of only alphanumeric characters. The following shows just the **switch** portion of this test:

```
switch ( testCharacter )
{
   case 'A': case 'B': case 'C': case 'D': case 'E':
   case 'F': case 'G': case 'H': case 'I': case 'J':
   case 'K': case 'L': case 'M': case 'N': case 'O':
   case 'P': case 'Q': case 'R': case 'S': case 'T':
   case 'U': case 'V': case 'W': case 'X': case 'Y':
    case 'Z':
     if ( convertToLower )
      {
      testCharacter =
     Character.toLowerCase( testCharacter );
      }
  case 'a': case 'b': case 'c': case 'd': case 'e':
 case 'f': case 'g': case 'h': case 'i': case 'j':
 case 'k': case 'l': case 'm': case 'n': case 'o':
 case 'p': case 'q': case 'r': case 's': case 't':
 case 'u': case 'v': case 'w': case 'x': case 'y':
  case 'z':

 case '0': case '1': case '2': case '3': case '4':
```

```
  case '5': case '6': case '7': case '8': case '9':
     token.append( testCharacter );
     break;
   default:
    isCharacter = false;
    break;
}
```

Tip One of the most limiting things about **switch** is that the **case** statement takes only a constant primitive value. You can't write a **switch** statement, for example, that matches strings (note that the previous example uses characters, not strings).

Using the while Statement

A **while** statement has the following form:

```
while (condition)
{
      statements;
}
```

Java evaluates the **condition** expression (which must be Boolean). If the expression is true, Java executes the block. When the block execution is finished, Java reevaluates the **condition** expression. If it is still true, the block repeats. This process continues until the condition is false. For example, this code would print numbers from 5 down to 1:

```
int i=5;
while (--i!=0)
  System.out.println(i);
```

Consider a program that uses a **while** loop to compare two strings based on the Unicode value of each character in the strings. The program prints a negative number if the first **String** precedes the second string in dictionary order. The program prints a positive integer if the opposite is true. The program prints zero if the strings are equal. Here is a **while** loop that does the work:

```
        while (position < testLength)
        {
            char c1 = char1[position];
            char c2 = char2[position];

            if (c1 != c2)
```

```
            {
                result = c1 - c2;
                break;
            }

            ++position;
        }

//--code removed for space--

        while (position < testLength)
        {
            char c3 = char3[position];
            char c2 = char2[position];

            if (c3 != c2)
            {
                result = c3 - c2;
                break;
            }

            ++position;
        }
//returns:
//The results of the comparisons are:
//Devyn compared to Devyn = 0
//Devon compared to Devyn = -10
```

The **while** loop always tests the condition before it executes any code. That means that if the condition is initially **false**, the **while** loop's body will never execute.

Tip If you are certain that the loop must execute once, then use the **do...while** statement instead of the **while** loop. Also, be careful about specifying an expression that is never **false** so that the loop never terminates.

Using the do...while Statement

A **do…while** statement is the same as the **while** loop except for this one important difference: The **do…while** loop always executes its statements at least once because the termination test doesn't happen until the end of the loop. Here's an example:

```
int index;
do
{
    System.out.print(i);
   i++; //remember: equivalent to i=i+1;
} while(index < 10)
//prints: 12345678910
```

Tip Use the **do** loop when the number of loops to execute is uncertain. If you want to loop a fixed number of times, use the **for** statement.

Using the for Statement

You should use the **for** statement when you have a definite number of times you want to execute a loop. This statement includes three sections: **initialization**, **condition**, and **update**. A **for** statement should have the following form:

```
for (initialization; condition; update)
{
    statements;
}
```

When Java first encounters a **for** statement, it executes the **initialization** clause. This can set an initial value for a loop variable (**i=0**, for example), or you can declare a unique variable for the loop and initialize it (**int i=0**). If you declare a variable here, its scope is just the body of the loop.

The next step is to evaluate the **condition** expression. If the condition is false, the loop does not execute. Java repeats this test when the loop repeats.

If the body of the loop executes, the **update** clause evaluates at the end of the block, and then Java tries the **condition** expression again. If the condition is true, the block repeats. If the expression is false, the loop is over; Java skips the block and continues execution. This loop will print the numbers 1 to 10 on the console:

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

The following program demonstrates using a **for** loop to march through the characters of a string to build a hash value for the string:

```
public class stringHashCode
{
    public static void main(String[] args)
    {
        String stringToHash = args[0];
        char stringChar[] = stringToHash.toCharArray();

        int hashcode = 0, count = 0;
        int len = stringToHash.length();

        for (int index = 0; index < len; index++)
        {
            hashcode = 31*hashcode + stringChar[count++];
        }

        String output = "Hashcode for " + stringToHash +
                        " = " + stringToHash.hashCode() + "\n";
        output += "Java native hashcode = " + hashcode + "\n";
        output += "          our hashcode = " + hashcode ;
        System.out.println( output );
    }
}
//returns:
//Hashcode for john = 3267851
//Java native hashcode = 3267851
// our hashcode = 3267851
```

You don't have to specify all three parts of the **for** loop if you don't want to. For example, if the loop is already initialized, you can omit the **initialization** clause:

```
for (;i<10;i++) . . .
```

Of course, that assumes that **i** and any other variables you need are already set before you enter the loop. You can even omit all three clauses if you want to loop forever (or until a **break** statement occurs, which is the topic of the next section):

```
for (;;;) go(); // execute forever
```

Using the break Statement

The **break** statement is used to exit a block contained within a **switch**, **while**, **do**, or **for** statement. By default, execution continues after the controlling statement. For example, this loop will exit if the loop variable becomes negative:

```
for (;i==5;i--)
{
    if (i<0) break;
// other statements here
}
```

Presumably, this **break** statement will prevent the problem in which **i** was less than 5 already. When **i** becomes negative, the **break** statement will cause Java to stop executing the loop.

You can also break out of multiple loops by using a label (see the "Using Labeled Statements" section).

Using the continue Statement

The **continue** statement is used to skip the current iteration of a **for**, **while**, or **do** loop. When Java encounters a **continue** statement, it short-circuits the current loop and moves to the next iteration. For example, consider this loop:

```
for (int i=1;i<=3;i++)
{
  if (i==2) continue;
   System.out.println(i);
}
```

This loop will print only 1 and 3. When **i** is 2, the **continue** statement causes the loop to skip that iteration.

Sometimes when multiple loops are nested, you'll want to continue an outer loop from within an inner loop. You can do that with labels.

Using Labeled Statements

Statements can have labels. You can refer to these labels with the **continue** and **break** statements. Unlike variables, labels can have the same name as a package, class, interface, method, field, parameter, or local variable.

Consider this example:

```
outerloop:
```


Team-Fly®

```
for (int i=0;i<3;i++)
{
  for (int j=0;j<10;j++)
  {
  if (i==2) continue outerloop;
   System.out.println(i*10+j);
  }
}
```

Here, the **continue** statement causes the **i** loop to start again instead of the **j** loop (which would start again if you didn't use the label). Of course, you can also use **break** with a label if you want to exit the outer loop instead of continuing it.

Using the return Statement

Use **return** to terminate the current method. The **return** statement terminates a method and (optionally) sends back a value to the calling statement. You'll see more about the **returnType** method when you start to write your own methods in Chapter 4.

The following program computes the length of the hypotenuse of a right triangle, where the sides of the triangle are **side_1** and **side_2**. The **return** statement returns the length as a **double** data type:

```
static double computePythagorus(double side_1, double side_2)
{
  double hypotenuse = 0D;
 hypotenuse = Math.sqrt(side_1 * side_2 + side_2 * side_2);
  return hypotenuse;
}
```

Tip You can embed a **return** inside any statement (such as **for**, **do**, **switch**, or **if**), and it will terminate the whole method rather than just that loop. However, a **finally** block will still execute.

Related solution:	Found on page:
Defining Methods	97

Using the try-catch Statement

The **try** statement executes a block of code and handles exceptions that might occur during execution. For example, the following method calls another method, **compute**, that might throw an **Arithmetic-**

Exception (perhaps from a divide-by-zero error). If an exception occurs, the method returns 0, as shown at the top of the next page:

```
int process(int z)
{
    try
    {
        return compute(z);
    }

   catch (ArithmeticException e)
    {
        return 0;
    }
}
```

You can add as many **catch** clauses as you need, putting more specific types before more general types. Java searches from the top down until it finds a match. If there is no match, the exception propagates to the method that called **process** (in this example). This continues until a matching handler is found. If there is no handler, Java reports an error and terminates the program.

Using the try-catch-finally Statement

You can add a **finally** block to a **try** block. When the block under the **try** statement exits for any reason (including a normal exit), the **finally** block executes.
The following is an adaptation from a **finally** statement used in Java's **Timer** class:

```
public void run()
{
      try
      {
            mainLoop();
      } finally
      {// Dead Thread, behave as if Timer cancelled
            synchronized(queue)
            {
                newTasksMayBeScheduled = false;
               queue.clear(); // Eliminate obsolete references
            }
```

```
    }
}
```

Tip Use **finally** to ensure that certain cleanup tasks are executed regardless of any exceptions.

Using the throw Statement

A **throw** statement forces an exception. This transfers control to the next **try-catch** statement that matches the object named in the **throw** statement. The following code shows you how to use it:

```
throw e; //e is a class derived from Throwable
```

Executing a **throw** statement exits the enclosing block. Here's an example:

```
try
{
   return cons.newInstance(new Object[] { string });
} catch (Throwable ex)
{
   throw new ParseException("Error creating instance", 0);
}
```

Tip You'll get a compile-time error if you attempt to throw an object that is not throwable. For example, you can't throw an **int** or a **String**.

Chapter 4: Methods, Classes, and Packages

In Brief

Although primitive data types and control structures comprise the details of Java programming, classes form the backbone of all Java programs. Although previous chapters have mentioned classes and objects, they have not yet been discussed in detail. This chapter describes how Java uses classes to create objects and why these ideas are important in all modern programming languages.

About Objects

Object-oriented programming provides many benefits. In particular, modern programming languages strive to meet the following goals:

- *Reuse*—You'd like to be able to reuse code in multiple places both inside and outside your program.
- *Encapsulation*—Information about details should be accessible on a need-to-know basis. For example, if you read a phone number from a database, only one part of your program should know if the phone number resides in a text file, in a SQL database, or on a remote Internet server. Objects that provide uniform access to a resource are said to *encapsulate* that resource. Suppose your program has one object that knows how to retrieve phone numbers. If the source of the phone numbers changes, only the one phone-number object requires modification.
- *Polymorphism*—Polymorphism allows you to handle items as though they were of a more generic type. For example, chocolate chip cookies and oatmeal cookies are both types of cookies. For many purposes, the fact that they are cookies is enough information. I might create a generic class that represents all cookies. Subclasses of this object would represent specific kinds of cookies, but my program could treat the specific types as the generic cookie class when it is handy to do so.

If you aren't familiar with object-oriented concepts, think of objects as smart data. Consider a program that looks up phone numbers. In an object-oriented world, the phone number is likely to be an object. The main program doesn't store phone numbers per se. Instead, it stores one or more **phoneNumber** objects. When you construct a **phoneNumber** object, you provide a name. The details about how the name is transformed into a number is up to the object. That's *encapsulation*.

An object like the **phoneNumber** object might be useful in several parts of your program. For example, you might want to use a separate set of **phoneNumber** objects to handle fax numbers. If you plan the class correctly, you'll also be able to pull it out of one program and put it in another program with a minimum of hassle. That's *reuse*.
Suppose your program starts with a simple **phoneNumber** class that reads numbers from a text file. As your user base grows, you decide you need to handle different databases and Internet servers.

You can write specialized versions of the **phoneNumber** class to handle each case. One type of phone number knows how to read data from the Windows address book. Another type of phone number reads from Oracle databases. A third class looks up the number via a special Internet server.

There are two important points to know about this scenario. First, the main program doesn't need to know about the details because of encapsulation. As long as the main program's interface to the phone number remains unchanged, you don't care how it works.

The second point is that if you design the host of **PhoneNumber** classes correctly, your program can treat all the specializations as a plain phone number. The fact that some use one database and others use a different one is not material. That's *polymorphism*.

In Java, objects will expose *variables* (known as *fields*) and *functions* (often called *methods*). Some of these fields and methods will be visible to the outside world. These fields and methods constitute an object's public interface. In the phone-number example, the object might expose one method called **lookup** and another method called **getNumber** (or it might simply have a **number** field). Any object that provides this functionality would work with your main program. Other methods and fields would not be public, so the main program would not have access to them.

Inside Classes

In Java, everything you write (except **import** and **package** statements) will reside inside a class. It is easy to think of classes as objects, but that isn't accurate. Classes aren't objects any more than cookie cutters are cookies. A class provides a pattern that you can use to create one or more objects.

The items that make up the class are known as *members*. Members are accessible by only certain parts of your program. For example, a class can declare a member **private**. Then only code in the same class definition can access that member. A **public** member is visible to all programs. Members can also be **protected**—something you'll read about shortly.
If you don't specify **public**, **private**, or some other access modifier, the member is visible to any class that shares your class's package. Packages allow you to group related classes together.

Don't confuse local variables with fields. The placement of a variable determines its scope. If the variable's declaration appears inside a method, the variable is local. If the declaration appears within the class but outside any method, however, then you have a field (sometimes called a *member variable*). You don't specify the access to local variables because they have a well-defined scope and are not accessible by any other part of your program anyway.

You can use a variety of modifiers to control the scope of a member. Table 4.1 shows the possible modifiers (some of these modifiers apply only to methods). You'll see many of these used in the "Immediate Solutions" section.

Table 4.1: Member modifiers.

Modifier	Definition

public	Accessible by any class in any package.
protected	Accessible by the member's class, subclasses, or classes in the same package.
<none>	Is the default if no modifier is specified. Classes in the same package can access this member.
private	Accessible only to the class in which it is declared.
static	Indicates a class variable (all objects that belong to the class share it), rather than a variable maintained by each object.
final	Indicates a constant.
transient	Indicates that the member won't be serialized nor the state saved.
volatile	Omits this member from certain compiler optimizations.
abstract	Marks a method as a placeholder; derived classes must supply an implementation.

You can declare any variable to be **final**. Doing this enforces the rule that the value of this member cannot change. The following variable declaration defines a variable that remains constant. This variable is named **MONA_LISA_PAINTER**, whose value is **Leonardo da Vinci**:

```
final String MONA_LISA_PAINTER = "Leonardo da Vinci";
```

You'll get a compile-time error if your code tries to change the value of **MONA_LISA_PAINTER**. By convention (this is not a rule), the name of a constant is written in uppercase letters.

The **static** modifier indicates that the member belongs to the class, not to a particular object. Suppose you wanted to track how many phone numbers the **phoneNumber** object retrieved. Where would you store that value? It doesn't belong with any one phone number because the value applies to all phone numbers. To solve this problem, you could store the number in a **static** variable. To extend the cookie-cutter analogy, if a class is a cookie cutter and an object is a cookie, **static** variables might include a counter built into the cookie cutter to count cookies and a timer to measure elapsed time. These things apply to all the cookies, not just to one.

Constructors

When you create an object, you use the **new** operator to instantiate an object from a class. A **String** is an object, so you might write:

```
String s = new String();
```

This statement creates a new, empty string. The second occurrence of the word **String** is a call to a special method known as a *constructor*. The constructor's name is always the same as the class name. Java allows you to have multiple methods with the same name (as long as they have different parameters), so one class can have many different constructors.

As with any method, you can also provide arguments for this special constructor. You could write:

```
String s = new String("Black Book");
```

When you write your own classes, you'll provide a constructor for every case you want to handle. If you don't provide any constructors, Java creates a default constructor (a constructor method with no arguments) for you. If you provide any constructor, however, Java won't add a default for you; you'll have to code it if you want a default constructor. Of course, some objects always require construction arguments and therefore don't have default constructors.

A Simple Object

Consider this simple class definition:

```
public class T
{
   private String msg="Wow!";
   public void printMsg()
    {
       System.out.println(msg);
    }
  public static void main(String [] args)
    {
      T test = new T();
       test.printMsg();
    }
}
```

This is a complete program (it includes a test **main** to exercise the method within the object). Notice that there is no constructor; the program uses the default constructor. This is possible because you

can initialize member variables without using a constructor (in this case, setting **msg** to "Wow!" during the variable's declaration).

You could specify a constructor to allow the **main** program to set the message:

```
public class T
{
    protected String msg;
    public T(String s) { msg = s; }
    public void printMsg()
    {
      System.out.println(msg);
    }
  public static void main(String [] args)
  {
    T test = new T("Black Book");
     test.printMsg();
  }
}
```

The **msg** field is **protected**. This status is similar to **private**. In the next section, however, you'll see that the **protected** modifier allows subordinate objects to access this field.

In this case, if you wanted to continue to allow a default constructor, you'd have to specifically write it, adding this line, for example:

```
public T() { msg="Wow!"; }
```

Extending Objects

Many times you need to create an object that is almost like an existing object but with slight differences. In this situation, you can extend the existing object. When you extend an object, you inherit all of its members marked as **public** (unless you choose to override them). You can also add new public members. In addition, your new class can access items in the original class marked as **protected** (but not **private**).

For example, suppose the simple class from the last example (**T**) is almost what you need, but you want another method to print the string in uppercase. You can write:

```
class Tupper extends T
{
    public Tupper(String s)
```

```
    {
      super(s); // chain constructor
    }
    public void printMsgUpper()
    {
       System.out.println(msg.toUpperCase());
    }
}
```

In this case, you say that class **Tupper** extends **T**. That makes **T** the base class for **Tupper**, and **Tupper** derives from **T**.

This new class is just like **T** but includes an extra method. The **Tupper** class has the same members as the **T** class (except for the constructor). Notice that each class must provide its own constructors (unless the base class has a default constructor).

You could just as easily replace **printMsg** with a new version instead of creating a new method. For example:

```
class Tupper extends T
{
    public Tupper(String s)
    {
      super(s); // chain constructor
    }
    public void printMsg()
    {
       System.out.println(msg.toUpperCase());
    }
}
```

The **super** keyword refers to the base class. As a special case, you can use this keyword as the first line in a constructor to call a particular constructor in the base class. You can also use this keyword anywhere in any method to access methods in the base class. For example:

```
public void printMsg()
{
```

```
    super.printMsg();
    System.out.println(msg.toUpperCase());
}
```

This is a good example of polymorphism. Consider this **main** method:

```
public static void main(String [] args)
{
  T test = new Tupper("Black Book");
   test.printMsg();
}
```

Notice that the variable **test** is of type **T**. When you run the program, however, it will execute **Tupper**'s **printMsg** method. Here, **Tupper** is a kind of **T**. In fact, any class that extends **T** is a kind of **T**. Moreover, classes that extend **Tupper** are not only kinds of **Tupper**, but they are also kinds of **T**.

Interfaces

Using the **extends** keyword not only supports polymorphism but also handles code reuse—two major goals of object-oriented programming. However, sometimes you want an object that has attributes of two different base classes. Java doesn't directly support this capability; each class can have only one base class (although that base class might, of course, extend another class).

Suppose you build a class named **Vehicle**. Classes such as **Car**, **Truck**, and **Motorcycle** would be obvious subclasses. You might construct another class named **Appliance**. Subclasses of **Appliance** would include **Dishwasher**, **Stove**, and **Freezer**. These class hierarchies don't seem to have much in common at first glance, but what about an ice cream truck? It is a truck, but it is also a freezer. Which class would you extend—**Truck** or **Freezer**? You must pick one.

To work around this problem, Java introduces the idea of an *interface*. You define an interface in a similar way to defining a class, but you don't provide any code for the interface. Classes that should be polymorphic with other objects must implement the interface—that is, supply the code. Programs can treat any object that implements an interface as though it were an object of the nonexistent class that corresponds to the interface.

Consider this example:

```
interface Printable
{
  public void print();
}
class T2 implements Printable
{
```


Team-Fly®

```
    int n;
   public T2(int n) { this.n=n; }
    public void print()
    {
       System.out.println("Number = " + n);
    }
}
class T3 implements Printable
{
    String s;
   public T3(String s) { this.s=s; }
    public void print()
    {
    System.out.println("String=" + s);
    }
}
public class T1
{
   public static void main(String [] args)
    {
      Printable p = new T2(100);
       p.print();
       p=new T3("Coriolis");
       p.print();
    }
}
```

Because both classes (**T2** and **T3**) implement **Printable** , they must include the methods defined in that interface (one method; in this case, **print**). What these classes do in that method is their own business, but they must include it. (Not including the interface methods will earn you a compile-time error.) Because each object contains a **print** method, the **main** method can treat them both as a **Printable** object (which isn't really an object at all). So **T2** and **T3** are polymorphic, but they don't share any code or base classes (except, of course, for **Object**, which is the eventual base class of all objects).

Another interesting point about the previous code is that the constructors use the same parameter name as the field they set. For example:

```
    int n;
  public T2(int n) { this.n=n; }
```

The **this** keyword represents the current object and allows the constructor to set the field without conflicting with the parameter of the same name.

Packages

When you don't specify any package, your class belongs to the default package. However, most classes you use will belong to a package. For example, every time you use a **String** object, you are using an object from the **java.lang** package.

You can create your own package by simply placing a **package** directive at the start of your source file. When you create a class in a package, the generated class file must reside in a subdirectory of a directory listed in **CLASSPATH** or in the same subdirectory of a JAR (or ZIP) file that resides in a directory named in the **CLASSPATH**.

For example, if you create a class in the package **com.coriolis**, then a directory or JAR or ZIP file in your **CLASSPATH** must contain a subdirectory called /com/coriolis (or \com\coriolis, if you prefer). By convention, you should select a package name that includes your domain name (if you have one) reversed (in other words, **com.coriolis**, not **coriolis.com**). This convention allows most packages to reside in one of a few top-level directories (com, net, org, and so on). You can have arbitrary levels in a package, so you can put anything you like after the reversed domain name (**com.coriolis.jlbb.samples**, for example).

When your class is in one package and you want to use a class in a different package, you must specify the entire name (for example, **com.coriolis.jlbb.samples.testClass**). The alternative, of course, is to **import** the class or the entire package. You don't need to fully specify classes in your own package or in **java.lang**.

Immediate Solutions

Using Access Modifiers on a Field

Fields may be **public**, **protected**, **package**, or **private**. These modifiers control how fields are used outside of the class that defines them—their *visibility*. Before we can discuss visibility directly, we must first revisit the idea of scope. Say you have a variable named **phoneRinger**:

```
int phoneRinger = 5;
```

This declaration defines **phoneRinger** as an integer and initializes it to 5. The parts of the program that can use this variable are determined by its placement in the source code. You can determine what can access **phoneRinger** by checking the nearest preceding curly brace. Everything between that brace and its matching closing brace has access to the variable.

If you place your variable declaration inside a method, then the variable is reachable only inside that method. Listing 4.1 demonstrates the scope of variables.

Listing 4.1: A scope demonstration program.

```
public class ScopeProgram

{

   public static void main(String args[])

    {

      int minValue = 1;

     System.out.println("main.minValue = " + minValue );

      int newMinValue = testVariableScope();

      System.out.println("main.minValue = " + minValue );

       System.out.println("testVariableScope.minValue = " +

                                  newMinValue );

    }

   public static int testVariableScope()

    {

       int minValue = 2;

       System.out.println("testVariableScope.minValue = " +

                                    minValue );

        return minValue;

    }

}

//returns:

// main.minValue = 1

//testVariableScope.minValue = 2

//main.minValue = 1
```

```
//testVariableScope.minValue = 2
```

You can see that the original declaration of **minValue** in **main** is 1. A variable with the same name also appears in a second method named **testVariableScope** . Although these two variables have the same name, they are completely unrelated.

The scope of the **minValue** variable is the **main** method. That means that every statement in **main** can use this variable. It also means that if a statement in **main** changes the value, then all subsequent statements that access that variable will see the new value. This also applies to the separate variable by the same name in the **testVariableScope** method. All statements in the **testVariableScope** method can see its own **minValue** . The two variables are oblivious to one another.

One occurrence of **minValue** has no effect on the other. You can pass the value between methods, however. This happens when the **testVariableScope** method is called, and it returns the value of its own **minValue** variable to the calling statement in the **main** method. Still, the scope or accessibility of each variable is confined to its own method.
These two variables are local variables. This means that they can be used only in the method in which they are declared. The program in Listing 4.2 demonstrates the scope of variables, whether local or member, and shows how the access modifiers affect them.

Listing 4.2: An access modifier demonstration program.

```
import java.lang.reflect.*;


class Trunk

{

   public String publicModifier = "public"; // all packages

    String packageModifier = "package"; // this package only

   protected String protectedModifier = "protected";//subclass

   private String privateModifier = "private"; // this instance

   String defaultAccessLevel = "trunk default access level";

    public void TrunkVariables ()

   { //print publicModifier, packageModifier, protectedModifier

       //privateModifier, defaultAccessLevel

}  {
```

```
class Branch extends Trunk
{
   String defaultAccessLevel = "Branch default access level";
    public void BranchVariables()
{  //print publicModifier, packageModifier, protectedModifier
    //privateModifier,  defaultAccessLevel
{
}
public class AccessLevel
{
   static String defaultAccessLevel = "default access level";
   public static void main(String[] args)
{
     Trunk pineTrunk = new Trunk();
     Branch pineTrunkBranch = new Branch();

      pineTrunk.TrunkVariables();
      pineTrunkBranch.BranchVariables();
      //print publicModifier gives error
      //error-packageModifier,protectedModifier,privateModifier

      //print pineTrunk.publicModifier,pineTrunk.packageModifier
     //  pineTrunk.protectedModifier,pineTrunk.protectedModifier
     // pineTrunk.privateModifier = error
```

```
// pineTrunkBranch.publicModifier
// pineTrunkBranch.packageModifier
// pineTrunkBranch.protectedModifier
// pineTrunkBranch.privateModifier = error
// defaultAccessLevel );


try
{
Class c = Class.forName("Trunk");
int m = c.getModifiers();
if (Modifier.isPublic(m))
    System.out.println("pineTrunk=public");
if (Modifier.isPrivate(m))
    System.out.println("pineTrunk=private");

Field[] declaredFields = c.getDeclaredFields();
int mod = 0;
for (int index = 0;
      index < declaredFields.length;
      index++)
    {
      mod = declaredFields[index].getModifiers();
      //Modifier.toString(mod) //access and type
      //also getType() getName()
       System.out.println(declaredFields[index].toString());
```

```
            }
        } catch (Throwable e){ System.err.println(e);   }
    }
}
//returns:
//publicModifier = public modifier
//packageModifier = package modifier
//protectedModifier = protected modifier
//privateModifier = private modifier
//defaultAccessLevel = trunk default access level
//publicModifier = public modifier
//packageModifier = package modifier
//protectedModifier = protected modifier
//defaultAccessLevel = Branch default access level
//pineTrunk.publicModifier = public modifier
//pineTrunk.packageModifier = package modifier
//pineTrunk.protectedModifier = protected modifier
//pineTrunkBranch.publicModifier = public modifier
//pineTrunkBranch.packageModifier = package modifier
//pineTrunkBranch.protectedModifier = protected modifier
//defaultAccessLevel = AccessLevel default access level
//public java.lang.String Trunk.publicModifier
//java.lang.String Trunk.packageModifier
//protected java.lang.String Trunk.protectedModifier
//private java.lang.String Trunk.privateModifier
```

```
//java.lang.String Trunk.defaultAccessLevel
```

This program shows you how to control access to variables with access modifiers. Other modifiers are covered later in this chapter.

Using the final Modifier to Create a Constant

Often, it is useful to define a constant value. For example, you might want to create a variable named **pi** that contains the value 3.1415927. Of course, you could just use a simple variable. When the value is a constant, however, using a simple variable has two drawbacks. First, an errant part of the program might accidentally change the value. And second, the compiler might be able to generate more efficient code if it knows the value of a constant can't change.

Here's how to create a constant:

```
final float pi = 3.1415927.
```

Tip A **final** variable cannot change from its initialized value.

Because you can't change a **final** value, you must supply an initial value when declaring this type of variable. You can also apply **final** to classes. In this context, **final** means that you can't extend the class to form a new class. A **final** method is one that subclasses can't override.

Keep in mind that for reference types (objects and arrays), a **final** modifier makes only the reference **final**. You can still alter the contents of the object or array. For example, consider this declaration:

```
final int[] iary = new int[30];
```

Because this is **final**, you cannot later write:

```
iary = new int[20];
```

However, you can write:

```
iary[4]=10;
```

Creating a Class Variable

Sometimes you want a field to apply to all instances of a class. You can do this with the **static** modifier. A **static** (or class) field exists even when no object instances exist. When you do instantiate the object, all the instances share the same **static** variable. Suppose you have a simple object, and you want to track how many times you've created it. Listing 4.3 shows a solution using a **static** field, **newCount**.

Listing 4.3: A static keyword demonstration program.

```
class StaticDemonstration

{
```

```
    static int newCount = 0;
     StaticDemonstration() { newCount++; }
    static void work()
     {
    // create some objects
   StaticDemonstration demo1 = new StaticDemonstration();
   StaticDemonstration demo2 = new StaticDemonstration();
    }

   public static void main(String[] args)
    {
    work();
// notice there are no StaticDemonstration objects in scope
    System.out.println(StaticDemonstration.newCount);
    }
}
```

Related solution:	Found on page:
Declaring Variables and Arrays	45

Defining Methods

In Chapter 3 , you encountered blocks and statements. You've also seen methods throughout the book. A *method* is a group of variables and statements that functions together as a logical unit. Like fields, methods can have modifiers (like **private** , **public** , or **static**). Methods also have a return type (which may be **void** if the method returns nothing) . This return type can be a reference type (such as an object or an array).

A method can have any number of parameters or arguments. When another part of the program calls the method, Java copies the arguments and passes the copies to the method. For simple types, the method gets a private copy. For reference types, any changes the method makes to the parameter will appear in the calling program as well.

Consider this example:

```
class methods
{
   static void x(int y,int[] ary)
    {
    System.out.println("Y= " + y + " ary[0]=" + ary[0]);\
    y=33;
    ary[0]=50;
    }

   public static void main(String[] args)
    {
      int v=100;
     int a[] = new int[10];
       a[0]=22;
       x(v,a);
     System.out.println("V=" + v + " a[0]=" + a[0]);
    }
}
```

TEAMFLY

You'll notice that method **x** changes the array (as seen by **main**) but doesn't affect the simple integer because the method is changing only a local copy of the data. The following rules apply to parameters:

- You can declare a method without parameters by using an empty pair of parentheses.
- No duplicate parameters are allowed in a method.
- A final parameter cannot be assigned within the body of the method.
- You can refer to a parameter with the simple name in the body of the method.
- The scope of a parameter is the entire body of the method.
- If a parameter has the same name as a class member, it is the parameter (not the class member) that is referred to within the body of that method. The class member is *hidden* by the parameter.
- Parameter names cannot be redeclared as local variables or as parameters of **catch** or **try** clauses.

Within a class, each method must have a unique signature. The signature is the method's name and its argument types. So, for example, you can write a method named **print** that takes an **integer** argument, and then you can write another method named **print** that requires a **String** argument. Still another version might accept two integers and a floating-point number. This is known as *overloading a method*. Of course,

Team-Fly®

methods in different classes can have the same name with no restrictions.

A method has two primary parts—the declaration and the body. The *declaration* defines the attributes, including access level, return type, name, and parameters. The *body* contains all the statements. Let's look more closely at the elements of a method declaration, as shown in Table 4.2.

Table 4.2: Method declaration elements.

Modifier	Definition
public	Accessible by any class in any package.
protected	Accessible by the method's class, subclasses, or classes in the same package.
package	Is the default if no modifier is specified. Classes in the same package can access this method.
private	Accessible only to the class in which the method is declared.
static	Indicates a class method, which is shared by all instances of the class.
final	Indicates a method that cannot be overridden by subclasses.
native	Refers to methods implemented in another language such as C; these are declared native.
synchronized	Prevents multiple threads using this method from stepping on each other.
returnType	Indicates the data type of the return value. Use **void** if nothing is returned.
methodName	Can be any legal Java identifier.
parameters	Specify the data you pass into a method.
exceptions	Indicate the exceptions your method throws.

Earlier, you saw that you can control access to a field by using **public**, **protected**, **package**, and **private** modifiers. These access modifiers also apply to methods. The program in Listing 4.4 demonstrates the effect they have on a method.

Listing 4.4: A method access demonstration program.

```
import java.lang.reflect.*;


class Apple

{

   protected void appleMethod (String accessor)

   { System.out.println( accessor ); }

}


class ApplePie extends Apple

{

    public void applePieMethod(String accessor)

   { System.out.println( accessor );

          appleMethod ("applePieMethod"); }

}

    class Orange

    {

      private void orangeMethod (String accessor)

     { System.out.println( accessor ); } }


    class Grape

    {
```

```
    void grapeMethod (String accessor)
    { System.out.println( accessor ); }
  }

  public class MethodAccessLevel
  {
    public static void main(String[] args)
    {
      //applePieMethod("MethodAccessLevel.main"); //error
      Apple apple = new Apple();
      ApplePie applePie = new ApplePie();
      Grape grape = new Grape();
      Orange orange = new Orange();
      apple.appleMethod("MethodAccessLevel.main");
      applePie.applePieMethod("MethodAccessLevel.main");
      grape.grapeMethod("MethodAccessLevel.main");
      //orange.orangeMethod("MethodAccessLevel.main"); //error

      MethodReporter report = new MethodReporter();
      report.methodReport("Apple");
      report.methodReport("ApplePie");
      report.methodReport("Grape");
      report.methodReport("Orange");
  }
}
```

```
class MethodReporter
{
    void methodReport (String className)
    {
        try
        {
           Class c = Class.forName(className);
           Method[] declaredMethods = c.getDeclaredMethods();
          for (int index = 0;
                index < declaredMethods.length;
                 index++)
            {
              //print declaredMethods[index].getName(), c.getName()

              Class[] exceptions =
                        declaredMethods[index].getExceptionTypes();
              for (int count = 0;
                     count < exceptions.length;
                      count++)
             { System.out.println( exceptions[count] ); }

              Class[] parameters =
                        declaredMethods[index].getParameterTypes();
             for (int count = 0;
```

```
                    count < parameters.length;

                     count++)

                {System.out.println( parameters[count].getName());}


                //declaredMethods[index].getReturnType().toString()

              System.out.println(declaredMethods[index].toString());


            //Object invoke(Object obj, Object[] args)//powerful

            }

     } catch (Throwable e) { System.err.println(e); }

    }

}

//Full source returns:

//protected appleMethod() accessed from = MethodAccessLevel.main

//public applePieMethod() accessed from = MethodAccessLevel.main

//protected appleMethod() accessed from = applePieMethod

//default grapeMethod() accessed from = MethodAccessLevel.main

//_____

//Apple Access Report

//The declaration of appleMethod

//class = Apple

//java.lang.String

//void

//protected void Apple.appleMethod(java.lang.String)

//_____
```

```
//...repeated for ApplePie, Grape and Orange
```

As you can see, using the access level is a nice way to control which classes can use each other. The access levels for methods parallel very closely the access levels for fields.
Another option you can specify in a method is the **throws** clause. This clause informs Java which checked exceptions the method might throw (and therefore, which exceptions the caller must catch). For example:

```
void myFunction() throws MyException
{
  // your code here
}
```

You can read more about exception handling in Chapter 3.

Related solutions:	Found on page:
Using the **return** Statement	77
Using the **try-catch** Statement	77

Creating final methods

You can declare a method as **final**. Whereas **final** makes a field into a constant, it has a different meaning for a method. A **final** method can't be overridden in a derived class. An example looks like this:

```
final void countCustomers(int initialCount, int newCount)
{
    // do something here
}
```

Defining and Using Classes

We have covered variables, statements, blocks, and methods. Many of the previous sample programs have included classes. Now it is tim e to formally define a class and then use one. When we talk about objects, we are technically talking about the use or *instance* of a class, not the class itself. Before we get to objects and object-oriented programming topics (*inheritance*), we must begin with the class. The following is a simple class:

```
public class ApplePie extends Apple implements Fruit
{
  public String color = "lime";
  public String size = 25;
  public ApplePie (int initialSize)
```

```
  { this.size = initialSize; }

  public void printColor()
 { System.out.println("the color is " + this.color ); }

  public void printSize()
 { System.out.println("the size is " + this.size ); }
}
```

The first line starts with the access modifier **public**. This is followed by the class name **ApplePie**. The keyword **class** tells the compiler that this is, of course, a class. The keyword **extends** indicates that this class will inherit the **Apple** class. The **implements** keyword means that this class will include the methods defined in the **Fruit** interface. Table 4.3 lists the elements of a class declaration.

Table 4.3: Class declaration elements.

Element	Description
public	Indicates that the class can be used by all classes in all packages; if a class is not **public**, only other classes in the same package can use it.
abstract	Indicates that the class cannot be instantiated. It acts as a template.
final	Indicates that the class cannot be subclassed.
class	Tells the compiler that what follows is a class.
extends	Identifies the superclass of the class.
implements	Implements one or more interfaces (you supply a comma-separated list of interfaces).

The **public** modifier says that this class can be used by any class in any package. Without it, the class can be used by other classes in the same package only, not by classes from other packages. An **abstract** class is one that cannot be instantiated. Why would you do that? An **abstract** method is useful as a base class that represents something that isn't a real object. For example, an object for **dogs** and another for **cats** might share a common base class (**Animal**). There's no such creature called "animal." An animal is an abstract *category* of other, specific creatures (such as dogs and cats). You could make the **Animal** class **abstract** to prevent instantiation of the object.

The **final** modifier prevents a class from being extended. This enforces programming discipline and can help the compiler generate better code.

You can make a class abstract with the **abstract** modifier. The following code shows you how:

```
public abstract class Number implements java.io.Serializable
{
   //Returns the value as an int.
   public abstract int intValue();

   //Returns the value as a long.
   public abstract long longValue();

   //Returns the value as a byte.
   public byte byteValue()
   { return (byte)intValue(); }

   //Returns the value as a short.
   public short shortValue()
   { return (short)intValue(); }
}
```

An abstract class acts like a template for other classes. It becomes the superclass of classes that inherit it (said to be subclasses). This example is adapted from Java's **Number** class. This **Number** class is actually the superclass for the **BigDecimal**, **BigInteger**, **Byte**, **Double**, **Float**, **Integer**, **Long**, and **Short** classes. These are the wrappers for the actual primitive data types of the same name (such as **int** and **short**). There is no such thing as a **Number** type; there are only specific types of **Numbers** represented by subclasses of this abstract class.

Methods can also be **abstract**, and a class with **abstract** methods must be an **abstract** class. Marking a method as **abstract** means that derived classes must override the method, and you are not required to provide a definition for it in the abstract class.

Using Class Constructors

All Java classes have constructors. If you don't explicitly write one, Java will automatically provide a default constructor. It doesn't do anything, but it's there anyway. You use constructors to ensure that an object is prepared for use. Generally, you place initialization code in a constructor. When that class is instantiated somewhere, the constructor is automatically called as part of the instantiation. Because

this happens first, it's your chance to set up variables to your liking before the calling class can do any damage.

Suppose you have a class called **Car**. You can provide a constructor for it like this:

```
public Car extends Vehicle
{
   String tank = "empty";
   String tires = "empty";

   public Car(String condition)
   {
      tank = condition;
      tires = condition;
   }
}
```

This class has two class members: **tank** and **tires**. The constructor sets these variables based on a parameter (**condition**):

```
Car newCar = new Car("full");
```

You can control the visibility (which other objects can see it) of your class with an access modifier in the constructor's declaration. Table 4.4 lists these constructor modifiers.

Tip Use constructors to ensure that you control how your objects start life.

Table 4.4: Constructor access modifiers.

Modifier	Description
private	No other class can instantiate the constructor.
protected	Only subclasses and classes in the same package can instantiate the constructor.
public	Any class in any package can instantiate the constructor.
default	Any class within only this package can instantiate the constructor.

Declaring an Interface

An abstract class is different from an *interface*, which is a collection of method definitions or signatures without implementations or code in the body. You can also declare constants in an interface. The **implements** keyword indicates that the class you are defining will include all the methods in the interface it is implementing. Of course, you can also use other methods, but you must at least create the methods named in the interface. You can **implement** more than one interface, and that means you have to write code for each method described in all the interfaces.

Interfaces can extend other interfaces. Here's how you can declare an interface in your program:

```
public interface Document extends Node
{
    statement(s);
}
```

An interface differs from an abstract class in three primary ways:

- An abstract class can have method bodies (implemented), but an interface cannot.
- A class can have only one superclass (extends one abstract) but many interfaces (implements a comma-separated list of interfaces).
- An interface is not subclassed and so is not part of the class hierarchy. Any class can implement any interface.

If a class implements an interface, then the class *must* use all the methods exactly as they are found in the interface. The interface body includes a group of method declarations without a body (a semicolon terminates declaration without curly braces).

Tip You cannot use the **private**, **protected**, **transient**, **volatile**, or **synchronized** modifiers in a member declaration in an interface.

Using Packages

Every Java program uses at least one package. If you compiled the program listings in this book, you were automatically using the default package (not named). This is the way to group classes and interfaces (specialized classes) together. A package groups classes and interfaces together, allowing access and namespace management.

You create a package like so:

```
package fruit;

public class Apple
{   statement(s); }
```


Team-Fly®

```
public class Orange
{ statement(s); }

public class Pear
{   statement(s); }
```

You do not use braces around the package—it applies to the entire file. All classes and interfaces in this source file are considered part of the package declared at the top with the **package** keyword and the package name terminated with a semicolon.

> **Tip** Programmers strive to make package names unique, as Internet addresses are. Many programmers begin their package names with their reversed Internet domain name, like so: **com.coriolis.publishPackage**. This is done to avoid name collisions. The reversal prevents you from having many top-level directories that all have com subdirectories.

The compiler and the JVM (Java Virtual Machine) use the package name by simply looking in the directory of the same name. So, for example, a class in the **com.coriolis.publishPackage** package will be in the com\coriolis\publishPackage directory. Alternatively, you can store the class file in a subdirectory stored inside a JAR file.

The most important effect on classes lies in how the package controls access. As you have seen earlier in this chapter, the class access modifier **public** allows classes from any package to access the class; if you omit the **public** modifier, however, then only classes within the same package can access that class.

If you want to use classes outside of a given package, you'll usually **import** them like this:

```
import java.util.Dictionary;
import java.util.Enumeration;
```

Without these imports, you'd have to fully qualify the package name each time you use it. The previous **import** statements can be condensed to just the following:

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

The wildcard (*) tells Java to get all the classes in the package. This approach isn't as efficient at compile time, but it doesn't affect the runtime performance of your program. The compiler grabs what it needs from other packages only when it comes across the **import** statement. It is a good idea to import explicitly just the actual classes you need. Not only does it make your compiles faster, but it

also prevents ambiguity that might needlessly arise if you import two packages that have identically named classes.

> **Tip** Sometimes you need two classes from two packages that have the same name. In this case, just use the fully qualified name instead of using **import**.

Understanding Inheritance

A subclass inherits variables and methods from its parent or super-class. If the superclass is a subclass of another class, then the subclass of the superclass also inherits variables and methods from the super-class of the superclass (the grandparent). This inheritance works for all ancestors. A class that is a subclass can use the superclass's members (variables and methods) in one of three ways. It can use them as is, hide them (sometimes known as *shadowing*), or override them.

> **Tip** The **super** keyword allows a subclass to access hidden variables and overridden methods of the superclass, like this: **super.color()**.

Overriding Methods

Sometimes you inherit a method that doesn't meet the requirements for a subclass. When this happens, you can override the method. This means that you write a new method (the overriding method) with the same return type, name, and parameter list (same signature) as the inherited method (the overridden method). A method signature is the name of the method and its parameters (in other words, **foo(int)** is different from **foo(float)** or **foo()**). Listing 4.5 shows you how to override one method in a superclass with another method of the same signature in a subclass.

Listing 4.5: A method overriding demonstration program.

```
import java.lang.reflect.*;


class Apple

{

   public void appleMethod (String accessor)

   { System.out.println( accessor );    }

}


class ApplePie extends Apple

{
```

```
    public void appleMethod(String accessor)
   { System.out.println( accessor ); }

    public void superAppleMethod()
   { super.appleMethod("applePie.superAppleMethod()"); }
}

public class OverrideDemonstration extends Apple
{
   public static void main(String[] args)
    {
       Apple apple = new Apple();
       ApplePie applePie = new ApplePie();

        apple.appleMethod("MethodAccessLevel.main");
        applePie.appleMethod("MethodAccessLevel.main");
        applePie.superAppleMethod();
    }
}
//full source returns:
//Apple.appleMethod() called from MethodAccessLevel.main
//ApplePie.appleMethod() called from MethodAccessLevel.main
//Apple.appleMethod() called from applePie.superAppleMethod()
```

You can see that the two method signatures are the same. This approach is contrasted with overloading, in which the method name is the same but the parameter list differs by at least one parameter type or name.

Overloading Methods

Earlier in this chapter, we saw how the signature of a method is defined (name, parameters). Overloading allows you to have multiple methods with the same name but different parameters. The return value is not considered, so you can't write two methods that only differ in their return types. There is no explicit declaration for overloading. Java just figures it out like this:

```
final void countCustomers(int initialCount, int newCount)
{
    statements(s);
}
final void countCustomers(int initialCount, int newCount,
  int offset)
{
    statements(s);
}
```

You can see that the two method signatures are the same except for the extra **offset** parameter in the second one. That one difference is enough to create two entirely different methods in Java. This is very handy. If you need a method that can be called with various parameter lists, then overload it. A good example of overloading is the **System.out.print** method. You can pass it any primitive-type variable, and the method will create a string for it.

Chapter 5: Data Structures

In Brief

In earlier chapters, you examined various control structures such as **if** and **while**. These control how your program executes. Another crucial element of most programs is the way they represent the data they manipulate. Selecting the proper data structure can make your program virtually write itself. Conversely, an incorrect choice of data structures can make a simple program difficult to write.

In a way, Java's classes are super-sized data structures—they contain not only data but also the programming instructions that operate on the data. In fact, you can model any data structure with a class, and Java's library is full of data-structure classes.

In Chapter 2, you learned something about simple data types, such as **int**, and you learned about arrays. An array is a simple form of data structure, but arrays are not always the perfect data structure. For example, suppose you want to store a list of phone numbers and retrieve them by name. You could do this with arrays, but a more sophisticated data structure might do a better job.

Collections

Java has had collections classes for some time, but Sun beefed up support in Java 2 SDK 1.2—the version that added the collections framework. In particular, the framework provides a common interface to a variety of collections (sometimes known as *containers*). Having a common set of methods that apply to different collections makes it easy to interchange collections classes and reduces the learning curve because you have only a few unique methods to learn.

Java provides several collections classes. Each type has a special focus (such as quick insertion or quick lookup) that makes it better for some applications but a poor choice for others. For example, if you are building an electronic phonebook, you have two primary functional considerations: The creation and deletion of new entries can be slow, but the lookup must be extremely fast. Given these two constraints, **Hashtable** is the best collection to use for your phonebook. Using the correct data structures is crucial because each one is optimized for a different operation.

Table 5.1 lists methods common to most collections. Of course, each class might have specific features that relate to its special features. Many collections implement the methods that appear in Table 5.2, for example, but these don't apply to all collections.

Table 5.1: Methods common to most collections.

Method Call	Returns	Description
contains(Object o)	**boolean**	Returns **true** if this collection contains the specified element.

Table 5.1: Methods common to most collections.

Method Call	Returns	Description
containsAll(Collection c)	**boolean**	Returns **true** if this collection contains all of the elements in the specified collection.
equals(Object o)	**boolean**	Compares the specified object with this collection for equality.
hashCode()	**int**	Returns the hash code's value for this collection.
isEmpty()	**boolean**	Returns **true** if this collection contains no elements.
iterator()	**Iterator**	Returns an iterator over the elements in this collection.
retainAll(Collection c)	**boolean**	Discards elements in the target collection that don't appear in **Collection c**.
size()	**int**	Returns the number of elements in this collection.
toArray()	**Object[]**	Returns an array containing all of the elements in this collection.

Table 5.2: Methods that only some collections implement.

Method Call	Returns	Description
add(Object o)	**boolean**	Adds the specified element.
addAll(Collection c)	**boolean**	Adds all elements in the passed collection to this collection.
clear()	**void**	Removes all of the elements from this collection.
remove(Object o)	**boolean**	Removes a single instance of the specified element, if it is present.
removeAll(Collection c)	**boolean**	Removes all this collection's elements that are in other collections.

Table 5.2: Methods that only some collections implement.

Method Call	Returns	Description
sort	**void**	Places elements in natural order (**List** interface only).
shuffle	**void**	Places elements in random order (**List** interface only).
reverse	**void**	Places elements in reverse order (**List** interface only).
fill	**void**	Places elements in natural order (**List** interface only).
copy	**void**	Copies all of the elements from one **List** interface into another.
binarySearch	**int**	Searches the collection using a binary search algorithm (which means the collection must be a sorted list).
min	**Object**	Specifies the minimum element according to natural order.
max	**Object**	Specifies the maximum element according to natural order.

One limitation of ordinary arrays is that they are fixed in size. Arrays are very fast, but they have limitations. How do you insert an element into the array? How do you ensure that there are no duplicate elements? How do you sort all the elements? These operations require you to write code—often inefficient code (inserting an element, for example, requires that you move all the subsequent entries in the array).

Collections can do all of this and more. They fall into three broad categories. *List*-type classes store data in some sort of sequence. The *Set* grouping of classes creates objects that store unique values. You can't create duplicate entries. Finally, the *Map* grouping of classes creates objects that create a dictionary where your program can look up a value by providing a unique key (which might, for example, be a string that contains a name).

As you might expect, each of these broad categories has its own interface (see Chapter 4). In addition to the interfaces, a number of actual classes provides implementations for these interfaces.

A collection contains elements, just as an array does. The **Collection** interface generalizes behavior, such as adding, removing, and counting elements. Underneath this master interface are three subinterfaces:

List, **Set**, and **Map**. Within these three interfaces are many concrete implementations (classes), shown below with descriptions of their primary uses:

- ***List***—A collection that has a distinct order of elements.

    - ***LinkedList***—A list that makes it easy to insert and delete elements.
        - ***Vector***—Similar to an array, but able to grow larger on demand.
        - ***ArrayList***—The same as a **Vector**, but unsynchronized. (This makes it more efficient, but it is not useful when using multiple threads.)
- ***Set***—A collection with no duplicate entries.
    - ***HashSet***—A collection that is rapidly searchable, but with no particular ordering of elements.
    - ***TreeSet***—A set in which the items appear in ascending order.
- ***Map***—A collection that relates a key value to another value.
    - ***Hashtable***—A **Map** that is rapidly searchable.
    - ***HashMap***—The same as **Hashtable** but unsynchronized. (This makes it more efficient, but it is unsuitable for multiple thread use.)
    - ***TreeMap***—A **Map** that retains values in ascending key order.
    - ***WeakHashMap***—A **HashMap** that does not hold references to its contents. Items that are in a **WeakHashMap** and that are not referenced anywhere else are subject to garbage collection.
    - ***Properties***—A **Hashtable** that maps a key string to a string value used for setting options in Java and many other programs.

Some of the collections are actually arrays underneath, wrapped in a class that adds functionality. Other collections use a different internal representation, but, thanks to encapsulation, you needn't be concerned about that.

Immediate Solutions

Copying Arrays

Internally, many collections store data in an array. You can use some special techniques to copy arrays more efficiently (and the Java library does use these special techniques). For instance, when you want to copy an array, or only a portion of it, to another array, use the **arraycopy** method. The **arraycopy** method requires five arguments. Two of the arguments are the arrays you are copying to and from. The other three arguments indicate the starting location in both arrays and the number of elements to copy. Here is a template:

```
public static void arraycopy(Object copyFromArray,
                             int copyFromArrayIndex,
                             Object copyToArray,
                             int copyToArrayIndex,
                             int numberOfElementsToCopy);
```

The following program shows you how to copy a portion of an array (elements two, three, and four in the third row of **EmployeeNames**) into another array (**EmployeeNamesWhoGetRaises**):

```
public class ArrayCopyTest {
  public static void main(String[] args)
  {
     String[][] employeeNames =
     {
        { "Jan", "Frank", "Wess", "Pat", "Donald" },
        { "Stan", "Beth", "Harold", "Kevin" },
       { "Gary", "Greg", "Les", "Karen", "Tom", "Abe" },
        { "Pete", "Clari", "Seth", "Arnold", "Abdul" }
     };

    String[] employeeNamesWhoGetRaises = new String[3];

     System.arraycopy(employeeNames[2], 1,
                          employeeNamesWhoGetRaises, 0, 3);

    int employeeNames_Length = employeeNames[2].length;
     int employeeNamesWhoGetRaises_Length =
                              employeeNamesWhoGetRaises.length;

     System.out.print("Copy from - employeeNames[2]: ");
    for (int index = 0; index < employeeNames_Length; index++)
     {
         System.out.print(employeeNames[2][index] + " ");
     }

     System.out.println();
     System.out.print("Copy to - employeeNamesWhoGetRaises: ");
    for (int index = 0;
            index < employeeNamesWhoGetRaises_Length;
            index++)
      {
```

```
            System.out.print(employeeNamesWhoGetRaises
                [index] + " ");
        }
    }
}
//returns:
//Copy from - employeeNames[2]: Gary Greg Les Karen Tom Abe
//Copy to - employeeNamesWhoGetRaises: Gary Greg Les Karen
```

Related solution:	Found on page:
Declaring Variables and Arrays	45

Manipulating, Comparing, and Searching Arrays

With an array, you have a sequence of elements accessed using an integer index. You will often need to sort an array, fill each element with default values, or test for equality between arrays. The following portions of the program ArraysDemonstration.java (which can be found in full at http://w ww.inforobo.com/javacorelanguage/) demonstrate all the actions you can take on an array of primitive data types:

```
int integerArray[] = {27, 18, -553, 95, 62, -37, 783};
for (int index = 0; index < integerArray.length; index++)
{
    System.out.print(intege rArray[index] + " ");
}
System.out.println(" -- integerArray");
int[] integerArrayCopy;

integerArrayCopy = integerArray;//point to same object
for (int index = 0; index < integerArray.length; index++)
{
  System.out.print(integerArrayCopy[index] + " ");
}

boolean areArraysEqual = Arrays.equals(integerArrayCopy,
```


Team-Fly®

```
                                        integerArray);

int[] integerArrayCopy_2 = new int[7];
System.arraycopy(integerArray, 0, integerArrayCopy_2, 0, 7);
areArraysEqual = Arrays.equals(integerArrayCopy_2,
                                        integerArray);

Arrays.sort(integerArray);

Arrays.fill(integerArray, 3, integerArray.length-1, 99);

int position = Arrays.binarySearch(integerArray, 18);

//returns:
// 27 18 -553 95 62 -37 783 -- integerArray
//27 18 -553 95 62 -37 783 -- integerArrayCopy
//true -- Arrays.equals(integerArrayCopy, integerArray)
//true -- Arrays.equals(integerArrayCopy_2, integerArray)
//-553 -37 18 27 62 95 783 -- integerArray
//true -- Arrays.equals(integerArrayCopy, integerArray)
//false -- Arrays.equals(integerArrayCopy_2, integerArray)
//-553 -37 18 99 99 99 783 -- integerArray
//true -- Arrays.equals(integerArrayCopy, integerArray)
//519 332 172 993 492 789 994 180 414 521 -- randomValues
//172 180 332 414 492 519 521 789 993 994 -- Arrays.sort()
//false -- Arrays.equals(randomValues, integerArray)
// 2 -- Arrays.binarySearch(integerArray, 18)
```

Tip If you assign one array to another, they remain equal. What you do to one will appear to be done to the other. This occurs because they actually point to the same object.

Using Vectors

The **Vector** class is the most like a normal array. Remember that ordinary arrays are fixed in size, but you can easily change the size of a **Vector** object whenever you want. As you can with an array, you can access items in a **Vector** object by using an integer index.

The **Vector** class holds objects, not primitive types. Java provides wrapper classes for all the primitive types you could use to store those types in a **Vector** class (for example, the **Integer** class wraps an **int**). Listing 5.1 shows an example of using a **Vector** class.

Listing 5.1: Using the Vector class.

```
import java.util.* ;

public class VectorAddGetTest {

    public static void main(String args[])

     {

    String firstArray[] = {"Violin", "flute",

                                    "trumpet", "piano"};

    String lastArray[] = {"Strings", "Woodwind",

                                   "Brass", "Percussion"};


     Vector band = new Vector();

      band.addElement(firstArray);

      band.addElement(lastArray);


     String instruments[] = (String[])band.firstElement();

    String instrumentTypes[] = (String[])band.lastElement();


    int instrumentsLength = instruments.length;

   for (int index = 0; index < instrumentsLength; index++)

     {

        System.out.print(instruments[index] + " ");

     }

     System.out.println();
```

```
    int instrumentTypesLength = instrumentTypes.length;
    for (int index = 0; index<instrumentTypesLength; index++)
     {
       System.out.print(instrumentTypes[index] + " ");
     }
     System.out.println();
   }
}
//returns:
//Violin flute trumpet piano
//Strings Woodwind Brass Percussion
```

As you add and remove items from a **Vector** object, it resizes to fit. You will need tomanage this resizing yourself for projects where performance is crucial (using fewer resizes results in faster programs). Here are some example **Vector** manipulations, taken from the VectorDemonstration.java program (found at the previously citedhttp://www.inforobo.com Web site):

```
import java.util.* ;

class VectorDemonstration
{
  public static void main ( String[] args)
   {
     Vector employeeNames = new Vector( 5, 3 );

      System.out.print("empty vector" );
     System.out.print("[capacity: " + employeeNames.capacity() );
     System.out.println(", size: " + employeeNames.size() + "]");
```

```
  employeeNames.addElement("Laura");
  employeeNames.addElement("Patricia");
  employeeNames.addElement("Devyn");
  employeeNames.addElement("Marianne");
  employeeNames.addElement("Shane");

for ( int index = 0; index < employeeNames.size(); index++ )
{  //the String element is automatically cast by System
      System.out.print(  employeeNames.elem entAt(index) + " ");
  }
 System.out.print("[capacity: " + employeeNames.capacity() );
System.out.println(", size: " + employeeNames.size() + "]");

  employeeNames.addElement("Kasienne");
  employeeNames.addElement("Ann");

 for  (  int index = 0; index < employeeNames.size(); index++ )
  {
     System.out.print( employeeNames.elementAt(index) + " ");
  }
 System.out.print("[capacity: " + employeeNames.capacity() );
System.out.println(", size: " + employeeNames.size() + "]");

  employeeNames.trimToSize();
 System.out.print("[capacity: " + employeeNames.capacity() );
System.out.println(", size: " + employeeNames.size() + "]");
  //employeeNames.contains("Devyn")  );
  //employeeNames.indexOf("Devyn")  );
   //employeeNames.removeElement("Devyn") );
  //employeeNames.contains("Devyn")  );

  //employeeNames.lastElement()  );
  //employeeNames.firstElement()  );
//employeeNames.isEmpty() ? "empty." : "not empty.";
```

```
    }
}
//returns:
//e empty vector [capacity: 5, size: 0]
//Laura Patricia Devyn capacity: 5, size: 5]
//Laura Patricia Devyn Ann [capacity: 8, size: 7]
//After .trimToSize() : [capacity: 7, size: 7]
//.contains("Devyn"): true
//.indexOf("Devyn"): 2
//.removeElement("Devyn"): true
//.contains("Devyn"): false
//.lastElement(): Ann
//.firstElement():Laura
//Vector is not empty.
```

In the following code, the three highlighted lines show you where the new elements were added. The total number of elements is now seven, two more than the **Vector**'s size when it was created. How does Java manage this when we know there's really an array underneath that can't change its size? Let's look at how a **Vector** changes its size internally:

```
private void ensureCapacityHelper(int minCapacity)
{
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity)
    {
        Object oldData[] = elementData;
        int newCapacity = (capacityIncrement > 0) ?
                          (oldCapacity + capacityIncrement) :
                          (oldCapacity * 2);
        if (newCapacity < minCapacity)
        {
            newCapacity = minCapacity;
        }

        elementData = new Object[newCapacity];
        System.arraycopy(oldData, 0, elementData, 0, elementCount);
    }
```

```
}
```

The previous routine illustrates how Java grows a **Vector**. The highlighted lines show where Java figures out how big to make the **Vector**. Remember that underneath, a **Vector** is just an array. This routine is designed to calculate how big to make the array. When the **Vector** runs out of room, it doubles in size. You can plainly see there is no magic here. A new array is created that is double the size of the old one. Then Java calls the **System.arraycopy** method to copy the elements from the old array to the new one.

The following program shows how Java inserts an element:

```
public synchronized void insertElementAt(Object obj, int index)
{
    modCount++;
   if (index >= elementCount + 1)
    {
        throw new ArrayIndexOutOfBoundsException(index
     + " > " + elementCount);
    }

    ensureCapacityHelper(elementCount + 1);
   System.arraycopy(elementData, index, elementData, index + 1,
                        elementCount - index);
   elementData[index] = obj;
    elementCount++; }
```

The previous method reveals the **Vector**'s internal algorithm. When you insert an element into a **Vector**, all the elements at the point of insertion are moved up one spot (copied with **System.arraycopy**), leaving an open spot into which the new element is placed. This is straightforward, but you can see how this operation could affect your program's performance. The **ArrayList** class (covered shortly) works in nearly the same way as a **Vector**.

Tip Inserting elements into a **Vector** is a slow process, so use one of the **List** collections.

Processing Each Element in a Collection

You often need a way to process each element in a collection. Java provides an **Iterator** interface to simplify this task. Each collection has an **iterator** method that returns some object that implements the **Iterator** interface. You don't know the exact type of the object, but then again you don't care—it implements **Iterator**, so you can treat it as though it were an **Iterator** object.

Armed with this object, you can call the **hasNext** method to see if there are more elements to process, and then call **next** to retrieve the next element. In addition, you can use **remove** to ask the object that implements **Iterator** to delete the current element from the collection. Sure, you could remove the

item directly, but that might confuse the **Iterator**; asking the **Iterator** to delete the item provides consistent results.

Listing 5.2 shows how you can iterate through each element of a **Vector**. Regardless of the collection, the code remains essentially the same.

Listing 5.2: Iterating a Vector.

```
import java.util.*;


public class VectIt

{

   public static void main(String [] args)

    {

       Iterator i;

       Vector v = new Vector();

      v.addElement(new Integer(10));

      v.addElement(new Integer(5));

      v.addElement(new Integer(40));

    //iterate

      for (i=v.iterator();i.hasNext();)

       {

         Integer n = (Integer)i.next();

           System.out.println(n);

       }

    }

}
```

Using a Stack

Stack is a class that extends a **Vector** class and acts like a last-in-first-out stack of objects. It does this with the following five additional stack methods:

- ***empty()***—Tests whether the stack is empty.
- ***peek()***—Looks at the object at the top but doesn't remove it.
- ***pop()***—Returns the top object and removes it.
- ***push()***—Adds an object to the top of the stack.
- ***search()***—Returns an element's position (the first position is 1, not 0).

If you examine the code that performs the **pop** operation, you'll see that **Stack** is little more than an enhanced **Vector** class:

```
return elementAt(len - 1);

removeElementAt(len - 1);
```

The StackDemonstration.java program (from the previously cited Web site) demonstrates how to use a stack. The following is an excerpt:

```
import java.util.* ;


class StackDemonstration
{
  public static void main ( String[] args)
   {
     Stack beefedUpVector = new Stack();
      beefedUpVecto r.push(new Integer(1));
     beefedUpVector.push(new  Integer(3));
     beefedUpVector.push(new  Integer(2));
     beefedUpVector.push(new  Integer(5));
      beefedUpVector.push(new String("Patricia"));
      beefedUpVector.push(new String("Kasienne"));


      //beefedUpVector.empty();
      //beefedUpVector.size();
     //beefedUpVector.search(new  Integer(1));
     //beefedUpVector.search(new  Integer(4));
      //beefedUpVector.pop();
      //beefedUpVector.pop();
      //beefedUpVector.peek();
   }
```

```
}
//returns:
// .push 1: [1]
//.push 3: [1, 3]
//.push 2: [1, 3, 2]
//.push 6: [1, 3, 2, 5]
//.push "Patricia":
//.push "Kasienne":
//[1, 3, 2, 5, Patricia, Kasienne]
//.empty() : false
//.size(): 6
//.search 3: 6
//.search 4: -1
//.pop(): Kasienne
//.pop(): Patricia
//[1, 3, 2, 5]
//.peek(): 5
//[1, 3, 2, 5]
```

Stacks are common in many programs such as RPN (Reverse Polish Notation) calculators, certain types of queues, and some areas of accounting programs.

Using an ArrayList or LinkedList Class

You can use the **ArrayList** or **LinkedList** classes when the order of the collection's elements is important. These lists have more methods than do the other collections. An **ArrayList** is just a **Vector** except that the **ArrayList** methods are not synchronized, whereas most of the **Vector** methods are synchronized. Synchronization makes the **Vector** thread-safe, but it also makes **Vector** somewhat slower.

Tip If you need a thread-safe **LinkedList**, use a synchronized wrapper from **Collections.synchronizedList(List)**.

As a general rule, the **ArrayList** class is better at providing random element access than a **LinkedList** is. The **LinkedList** class has more methods (**addFirst**, **addLast**, **getFirst**, **getLast**, **removeFirst**, **removeLast**) and is better at inserting and deleting elements from the middle of the **List**. The following program demonstrates the primary methods that are available for **List** collections but not available for other collections:

```
import java.util.* ;


class ListAlgorithms
```

```
{
  public static void main ( String[] args)
   {
       final int max = 15;
       List aList = new ArrayList();
        for (int index = 0; index < max; index++)
        {
            aList.add(new Integer(index));
        }
// The shuffle method randomly reorders a collection
       Collections.shuffle (aList, new Random());
        System.out.println("shuffle: " + aList);

        Collections.sort(aList);
       System.out.println("sort: " + aList);

        Collections.reverse(aList);
       System.out.println("reverse: " + aList);

       Integer i = (Integer)Collections.min(aList);
        System.out.println("min: " + i);

       i = (Integer)Collections.max(aList);
       System.out.println("max: " + i);
    }
}
```

The previous program uses the **ArrayList** class, but the program would work the same way if it used a **LinkedList** class in the declaration on the highlighted line. In practice, you'd select one or the other depending on which operation on the collection will be most common. An**ArrayList** is fast to access, but insertions and deletions are faster in a **LinkedList**. The following code snippet is from Java's **ArrayList** class:

```
public void add(int index, Object element)
{
   //resizing and capacity code removed for clarity
```


Team-Fly®

```
    System.arraycopy(elementData, index, elementData,
                     index + 1, size - index);
    elementData[index] = element;
}
```

Notice that the **ArrayList** class simply makes sure there is room (adding a bucket, if necessary), shifts the array, starting from the index and moving to the end, and then adds the object to the now-open element. The **remove** method does the same thing, only in reverse. This is why the **ArrayList** isn't fast at inserting or removing into the middle.

The following code from the ListDemonstration.java program (which you will find in full on the previously cited http://www.inforobo.com Web site) shows the performance of some **ArrayList** operations:

```
import java.util.* ;


class ListDemonstration
{
  public static void main ( String[] args)
   {
       final int max = 30000;


       List aList = new ArrayList();
        long startTime = System.currentTimeMillis();
        for (int index = 0; index < max; index++)
        {
            aList.add(new Integer(index));
        }
        elapsedTime(startTime, max, "aList.add");


        //repeat above construction for:
       // List lList = new LinkedList();
         // lList.add(new Integer(index));
       // Vector testVector = new Vector();
        // testVector.addElement(new Integer(index));


        for (int index = 0; index < max; index++)
        {
            aList.get(index);
```

```
        }

        //repeat above construction for:
        //    lList.get(index);
        //    testVector.contains(new Integer(index));

       //repeat above construction for:
        //    aList.remove(0);
        //    lList.remove(0);
        //    testVector.removeElement(new Integer(index));

//1 GHz Dell Laptop returns:
//aList.add (30000) = 0.08 seconds
//lList.add (30000) = 0.18 seconds
//testVector.addElement (30000) = 0.09 seconds
//aList.get (30000) = 0.06 seconds
//lList.get (30000) = 21.391 seconds
//testVector.contains (30000) = 47.748 seconds
//aList.remove (30000) = 1.542 seconds
//lList.remove (30000) = 0.06 seconds
//testVector.removeElement(0) (30000) = 1.563 seconds
```

A simple array is extremely fast for adding, deleting, and looking up elements if you know the index, but the whole point of the **List** collections is that you usually don't know the index. The previous program shows that an **ArrayList** is faster than a **Vector** overall, which is something that is especially true for lookups. You should try to add and delete elements from the end of a **Vector**. The **ArrayList** is slightly faster than the **LinkedList** when you insert elements but is much faster on lookups.

We'll look at one more **List** code snippet. By using the **BinarySearch** method, you can search a **List** that has been put in natural order. Look at how Java conducts the search:

```
int low = 0;
int high = List.size()-1;

while (low <= high)
{
  int mid =(low + high)/2;
  Object midVal = List.get(mid);
  int cmp = ((Comparable)midVal).compareTo(key);
```

```
  if (cmp < 0)
      low = mid + 1;
   else if (cmp > 0)
      high = mid - 1;
   else
  return mid; // key found
   }

   return -(low + 1); // key not found
}
```

The algorithm compares the target element to the element in the middle. If the middle value is too high, then the algorithm searches the middle of the lower half, and continues this divide-and-conquer approach until it finds a match. Of course, this assumes the list is sorted. The advantage of this standard search technique is that the speed is good (log n time). The **BinarySearch** code is a nice example of how you can traverse **List** collections.

Using a Hashtable

A **Hashtable** maps keys to values where the key and the value can be any non-null object. This map acts like a dictionary (actually extending the **Dictionary** class) in which you use a key to look up a corresponding value. We use this type of associative data structure every day in the form of phonebooks and Web addresses.

Hash tables use an algorithm to transform the key into an index. However, some keys might generate the same hash index (this duplication is known as a *collision*). The larger the hash table, the less likely it is that collision will occur. Collisions require special handling, which slows performance, so you'd like to minimize them if possible.

A hash table has two parameters that affect its performance: capacity and load factor. *Capacity* refers to the number of entries available, and *load factor* is a measure of how full the hash table is allowed to get. When the table level begins to fill up, it is automatically resized (or, in other words, rehashed). The more empty entries are available, the less likely it is that two keys will clash. But there is a lot of wasted space. Increasing the load factor trims space but slows execution. Java uses a default load factor (.75) as a fair tradeoff between performance and space. You can override the default by choosing your own load factor. If you are not concerned about using too much memory, use a low load factor. If space is paramount, though, use a high load factor.

The following program demonstrates most of the methods you will use with **Hashtable**:

```
import java.util.* ;

public class HashTableDemonstration
{
```

```
    public static void main(String args[])
     {
        String[] HTMLtableTags = { "<table>","<th>","<tr>",
//--code removed for space–      };
       String[] HTMLtableTagDefinitions = { "Defines a table",
                                            "Defines a table header",
                                              "Defines a table row",
//--code removed for space–                   };
         Hashtable HTMLtableElements = new Hashtable();
         for (int index = 0; index < HTMLtableTags.length; index++)
         {
             HTMLtableElements.put(HTMLtableTags[index],
                                        HTMLtableTagDefinitions[index]);
         }

         // use Enumeration to get all values
         Enumeration eTags = HTMLtableElements.keys();
         Enumeration eDefinitions = HTMLtableElements.elements();
         while (eTags.hasMoreElements())
         {
           System.out.print( eTags.nextElement() + " = " );
             System.out.println( eDefinitions.nextElement() );
         }

         String tag = "<colgroup>";
         String definition = (String)HTMLtableElements.get(tag);
         if (definition != null)
         {
           System.out.println(tag + " = " + definition );
         }

         System.out.println( HTMLtableElements.size() );
         HTMLtableElements.remove(tag);
         System.out.println( HTMLtableElements.size() );
```

```
        definition = (String)HTMLtableElements.get(tag);
        System.out.println(tag + " = " +
                           (definition!=null?definition:"gone") );
        //optional: print entire hashTable as string
        //System.out.println("HTML table tags: " );
        //System.out.println( HTMLtableElements.toString() );
    }
}
//returns:
// <td> = Defines a table cell
//<tr> = Defines a table row
//<thead> = Defines a table head
//--code removed for space–
//10
//9
//<colgroup> = gone
```

The **Hashtable** is superseded by the **HashMap**. Even though the **HashMap** is really using a **Hashtable** underneath, Sun recommends using **HashMap** in new programs.

Creating a HashSet Collection

You use **HashSet** for a fast collection of unordered but unique values. The hash is computed for each key and stored for fast retrieval, as is shown in the following code snippet:

```
String testString = args[0];
System.out.println("hashcode for " + testString + " is " +
    args[0].hashCode()); }
```

The command **java HashCodeTest "Little Black Book"** yields:

```
hashcode for Little Black Book is -574434828
```

All the hash-based containers, such as **HashSet** and **Hashtable**, use a simple scheme of computing a hash for storage. The following program (from the previously cited http://www.inforobo.com Web site) shows you how to use a **HashSet** to do unions and intersections for a hypothetical annual employee picnic:

```
import java.util.*;

class HashSetDemonstration
{
  public static void main ( String[] args)
```

```
    {
        System.out.println("Annual employee picnic:");
        HashSet picnicSigned = new HashSet();
        picnicSigned.add( "Kasienne" );
        picnicSigned.add( "Patricia" );
//--code removed for space–


        HashSet picnicPaid = new HashSet();
        picnicPaid.add( "Patricia" );
        picnicPaid.add( "Devyn" );
//--code removed for space–

        HashSet picnicInterested = new HashSet();
        picnicInterested.addAll( picnicSigned );
        picnicInterested.addAll( picnicPaid );//union
        System.out.println( "Interested = " + picnicInterested );

        HashSet picnicStillOweFee = new HashSet();
        picnicStillOweFee.addAll( picnicSigned );
        picnicStillOweFee.removeAll( picnicPaid );
        System.out.println( "You owe = " + picnicStillOweFee );
        HashSet picnicPaidButNeedToSign = new HashSet();
        picnicPaidButNeedToSign.addAll( picnicPaid );
        picnicPaidButNeedToSign.removeAll( picnicSigned );
        System.out.println( "Paid, need to sign = " +
                                    picnicPaidButNeedToSign );

        HashSet picnicAttending = new HashSet();
        picnicAttending.addAll( picnicInterested );
        picnicAttending.removeAll( picnicStillOweFee );
        picnicAttending.removeAll( picnicPaidButNeedToSign );
        System.out.println( "Approved = " + picnicAttending );
    }
```

```
}
//returns:
//Annual employee picnic:
//Signed = [Ann, Guy, Devyn, Patricia, Kasienne, Jean, Joseph]
//Paid = [Laura, Devyn, Patricia, Donna]
//Interested = [Ann, Jean, Kasienne, Patricia, Guy, Devyn,
//Donna, Joseph, Laura]
//You owe = [Ann, Guy, Kasienne, Jean, Joseph]
//Paid, need to sign = [Laura, Donna]
//Approved = [Patricia, Devyn]
```

You can see that using the idea of mathematical sets is useful when you need to obtain the union and intersection between lists.

Creating a TreeSet Collection

A **TreeSet** is the same thing as a **HashSet** except that, in a **TreeSet**, the keys are kept in order. The following adaptation of the program given in the **HashSet** section demonstrates the difference between the two collections:

```
import java.util.* ;


class TreeSetDemonstration
{
  public static void main ( String[] args)
   {
        System.out.println("TreeSet() sorts HashSet():");
       HashSet picnicSigned = new HashSet();
        picnicSigned.add( "Kasienne" );
        picnicSigned.add( "Patricia" );
        picnicSigned.add( "Ann" );
        picnicSigned.add( "Devyn" );
        picnicSigned.add( "Joseph" );
        picnicSigned.add( "Donna" );
       System.out.println( "HashSet() = " + picnicSigned );

        TreeSet sortedEmployees = new TreeSet();
        sortedEmployees.addAll( picnicSigned );
```

```
        System.out.println( "TreeSet() = " + sortedEmployees);
    }
}
//returns:
// TreeSet() sorts HashSet():
//HashSet() = [Ann, Devyn, Patricia, Kasienne, Joseph, Donna]
//TreeSet() = [Ann, Devyn, Donna, Joseph, Kasienne, Patricia]
```

Using a HashMap or a WeakHashMap

Both the **HashMap** and **WeakHashMap** require the same steps except for the subtle differences that occur when garbage collection (Java's automatic reclamation of unused memory) cleans up the map. Each map has the same methods; the only difference is the way garbage collection occurs. A **WeakHashMap** does not hold a reference on its keys. So if the key isn't used anywhere else, the garbage collector may reclaim it even though it is still a key in the **WeakHashMap**. This does not hold true for values, by the way—the **WeakHashMap** holds a reference to the values, so they are not subject to reclamation.

The following program (Listing 5.3), using methods that work for both maps, shows you how to place a value pair into a **HashMap**. Listing 5.3 also demonstrates how you can print the contents of the **HashMap** by using **System.out.print**. Finally, the listing shows you how to get the pair value for a given key value.

Listing 5.3: Using a HashMap.

```
import java.util.*;


public class wordIntegerMap

{

    public static void main(String args[])

    {

        Map wordIntegerMap = new HashMap();

        wordIntegerMap.put("One", new Integer(1));

        wordIntegerMap.put("Two", new Integer(2));

        wordIntegerMap.put("Three", new Integer(3));

        wordIntegerMap.put("Four", new Integer(4));
```

```
        wordIntegerMap.put("Five", new Integer(5));
         System.out.println(wordIntegerMap);
         System.out.print("{One=" +wordIntegerMap.get("One"));
        System.out.print(", Two=" + wordIn tegerMap.get("Two"));
        System.out.print(", Three=" +
                          wordIntegerMap.get("Three"));
        System.out.print(", Four=" +
                          wordIntegerMap.get("Four"));
        System.out.print(", Five=" +
                         wordIntegerMap.get("Five") + "}");
    }
}
//returns:
//{One=1, Three=3, Four=4, Five=5, Two=2}
//{One=1, Two=2, Three=3, Four=4, Five=5}
```

[redacted]

These maps store a hash of the key and then discard the key. When you perform a lookup, the map does the reverse—it hashes the provided key and looks for the hash value that was calculated upon insertion. If the map finds a match, it returns the associated value (for example, the hash of "Four" gets the integer 4).

The following code uses two maps (**WeakHashMap**) to create a constants table. The first map pairs the name of the constant with its numerical value, and the second map pairs the name with the symbol. This code demonstrates the methods you normally need to use for maps:

```
import java.util.*;

public class WeakHashMapTest
{
   public static void main(String args[])
    {
```

```
        Map usefulConstants = new WeakHashMap();
         usefulConstants.put("Avogadro", new Float(6.022e+23F));
         usefulConstants.put("Boltzmann", new Float(1.381e-16F));
//--code removed for space–

        Map usefulConstantUnits = new WeakHashMap();
         usefulConstantUnits.put("Avogadro",
                                 new String("molecules/mole"));
         usefulConstantUnits.put("Boltzmann",
                                 new String("erg/oK "));
//--code removed for space–
        System.out.println("NAME, Symbol, Constant");
        Float constant = new Float(0f);
        String symbol = "";

       String[] names = {"Avogadro", "Electron mass",
                         "Planck constant"};
        for (int index = 0; index < names.length; index++)
        {
           constant = (Float)usefulConstants.get(names[index]);
           symbol = (String)usefulConstantUnits.get(names[index]);
          System.out.println(names[index] + ", " + symbol +
                             ", " + constant);
        }
    }
}
//returns:
// NAME, Symbol, Constant
//Avogadro, molecules/mole , 6.022E23
//Electron mass, null, 9.109E-28
//Planck constant, erg/s , 6.625E-27
```

TEAMFLY

Team-Fly®

Chapter 6: Files and Streams

In Brief

Without files, computers wouldn't be nearly as useful as they are. Files store data in a persistent way and give users a way to exchange data with other users. From a programming point of view, files represent data storage that is not part of the program. Because of this, many things that are not actual files present themselves as files to a program. For example, a network connection or a printer might look like a file.

Java allows you to work with files, as you'd expect. In addition, Java uses the idea of streams—sequences of bytes or characters—to allow easy access to files or anything that might appear to be a file.

File Objects

One of the hallmarks of object-oriented programming systems is that they abstract common functionality into base classes. A common operation in any programming language is to process or emit a sequence of characters. For example, your program might read data from a disk file. Programs often want to write data to a network socket or a printer.

Java abstracts a source of characters or a destination for characters in its stream classes. At an abstract level, a *stream* is just the processing logic used to read or write a sequence of characters. Other classes provide features such as buffering portions of the file in memory. Finally, a set of core implementation classes associates streams with particular items, such as files, strings, the user's console, or network sockets. You can connect these objects together to fit your needs. Therefore, a stream that reads a file might feed a stream that performs buffering. That stream might, in turn, feed a stream that interprets basic data types (such as **int** of **float**).

All the stream classes (and many other related classes) are in the **java.io** package. This package contains two major groups of stream-related classes. The original set of streams (based on **InputStream** and **OutputStream**) deals with byte-oriented entities such as ASCII files or network sockets. A newer set of streams (based on the **Reader** and **Writer** base classes) deals with Unicode characters. Because it is possible to map bytes into Unicode, some classes will convert an **InputStream** into a **Reader** and some will convert an **OutputStream** into a **Writer**.

At first, this abstraction of streams from actual devices might seem superfluous. After all, most operating systems already abstract devices as files, right? So printers and consoles look like files anyway. Thanks to the way Java handles streams, however, a stream can even read from or write to a **String** object, a network socket, or even another program, for example.

Streams and Strings

Consider the **StringReader** class. This class is just a wrapper around a string with extra methods to help you march along the string, retrieving or skipping characters. Remember, a string is really just an array of characters with some built-in methods. The following snippet is an excerpt from the **StringReader.read**method:

```
public int read() throws IOException
{
    synchronized (lock)
    {
         ensureOpen();

        if (next >= length)
         {
            return -1;
         }

        return sourceString.charAt(next++);
     }
}
```

This method returns a single character from the stream or returns **-** 1 if it reaches the end of file. The expression **sourceString.charAt (next++)** simply picks off the character at the **next** position in the string. Keep in mind that any sequence of characters, no matter the source, can be a stream.

One stream class you have been using throughout this book is the **System.out** stream. In particular, it is a special type of stream that handles many different data types. At the top of the next page, you will see some code that shows how versatile this stream is:

```
public class PrintStream
{
   public static void main(String[] args)
    {
        System.out.println(Byte.MAX_VALUE);
        System.out.println(Short.MAX_VALUE);
        System.out.println(Integer.MAX_VALUE);
        System.out.println(Long.MAX_VALUE);
        System.out.println(Float.MAX_VALUE);
        System.out.println(Double.MAX_VALUE);
    }

}
```

The **println** method has overloaded versions that handle a wide variety of data types. In practice, the one you use most is the one that accepts a **String**. When you write:

```
int x;
```

```
x=somefunc();
System.out.println("The answer is " + x);
```

Java recognizes that you are trying to combine a string and an integer, so Java calls **Integer.toString** to transform the number into a string. The result is that the call to **println** simply receives a **String** object to print.

Table 6.1 shows the basic types of readers, writers, and streams. These classes represent streams that actually access the hardware (or memory) for characters. Table 6.2 shows the classes that act as wrappers over the classes in Table 6.1. Although these classes are also streams, they operate on other streams.

Table 6.1: Stream sources and sinks.

Type	Classes	Description
File	**FileReader**	Reads characters from a file.
	FileWriter	Writes characters to a file.
	FileInputStream	Reads bytes from a file.
	FileOutputStream	Writes bytes from a file.
Printing	**PrintWriter**	Writes formatted data to a Unicode stream (known as a *writer*).
	PrintStream	Writes formatted data to a stream.

Table 6.1: Stream sources and sinks.

Type	Classes	Description
Memory	**CharArrayReader**	Reads from a character array.
	CharArrayWriter	Writes to a character array.
	ByteArrayInputStream	Reads from a byte array.
	ByteArrayOutputStream	Writes to a byte array.
	StringReader	Reads from a string.
	StringWriter	Writes to a string.
	StringBufferInputStream	Reads from a string buffer.
Pipe	**PipedReader**	Reads characters from a pipe (a *pipe* is a stream connecting two threads).
	PipedWriter	Writes

Table 6.1: Stream sources and sinks.

Type	Classes	Description
		characters to a pipe.
	PipedInputStream	Reads bytes from a pipe.
	PipedOutputStream	Writes bytes to a pipe.
Serialization	**ObjectInputStream**	Reads objects from a stream.
	ObjectOutputStream	Writes objects to a stream.

Table 6.2: Manipulating streams.

Type	Classes	Description
Buffering	**BufferedReader**	Buffers data to improve performance (resulting in fewer actual hardware accesses).
	BufferedWriter	A writer that buffers output data in a memory buffer to improve performance.
	BufferedInputStream	Similar to

Table 6.2: Manipulating streams.

Type	Classes	Description
		BufferedReader, but operates on byte streams.
	BufferedOutputStream	Similar to **BufferedWriter** , but operates on byte streams.
Peeking ahead	**PushbackReader**	Allows the caller to read characters and then decide to put them back so that later reads will return the same characters.
	PushbackInputStream	Does the same thing as **PushbackReade r**, but uses a byte stream.
Filtering	**FilterReader**	Is a base class that simplifies writing filters that process one stream and produce a second stream and attach to a reader.
	FilterWriter	Is a base class that simplifies writing filters that process one stream and produce a second

Table 6.2: Manipulating streams.

Type	Classes	Description
		stream and attach to a writer.
	FilterInputStream	Is a base class that simplifies writing filters that process one stream and produce a second stream and attach to an input stream.
	FilterOutputStream	Is a base class that simplifies writing filters that process one stream and produce a second stream and attach to an output stream.
Concatenation	**SequenceInputStream**	Concatenates multiple input streams into a single stream.
Counting	**LineNumberReader**	Tracks line numbers while reading.
	LineNumberInputStream	Is a stream version of **LineNumberReader**.

Working with Java input/output (I/O) reminds me of plumbing. You take two pipes, put them together, and then find an adapter that makes them fit. You can join many pipes together if you have enough different adapters. It's perfectly acceptable to mix more than one adapter to fit the plumbing pipes together. For example, suppose you're writing a network program that uses the **Socket** class. You can read from or write to the socket by using streams (you can get a stream that corresponds to a socket using the **getInputStream** and **get-OutputStream** methods). You might use an **InputStreamReader** to convert the stream to a **reader** object. Then you would pass the **InputStreamReader** to the constructor for a **LineNumberReader**.

If you're a Unix user, this technique might remind you of the way you perform many tasks in Unix: by using pipes. So while the Unix command **cat foo|more** shows you a file, **cat foo|sort|more** shows you the same file with its lines sorted. Using small building blocks together to perform large tasks is such a powerful idea that you'll probably want to write your own plug-in modules that can alter a stream of input or output.

Filtering modules are so commonplace that Java makes it easy to produce them via special prototype classes that you can subclass. In particular, the **FilterInputStream**, **FilterOutputStream**, **FilterReader**, and **FilterWriter** classes preprocess data for **InputStream**, **OutputStream**, **Reader**, or **Writer** objects, respectively.

You won't use the prototype classes (such as **FilterReader**) directly. Instead, you'll create a subclass to do whatever specialized processing you need. Although each class has several functions, you'll mostly need to override the **read** and **write** methods. You can usually write one function to handle single characters, and then you can define the remaining functions in terms of those two character-oriented functions.

For example, you can create a class called **UCWriter** that forces all of the output to a writer into uppercase letters. When the filter class wants to write to the underlying writer, it uses the protected **out** field from **FilterWriter**. Nothing actually appears on the output stream until the class specifically writes to the **out** writer. You can find **UCWriter** in Listing 6.1.

Listing 6.1: This filter class converts characters to uppercase.

```
import java.io.*;


public class UCWriter extends FilterWriter

{

    public UCWriter(Writer out)

    {

     super(out);

    }

// This is the only method that actually converts to uppercase
```

```
// All the other methods call this one to do their work
    public void write(int c) throws IOException
     {
       super.write(Character.toUpperCase((char)c));
     }
    public void write(char[] cbuf,int off,int len)
        throws IOException
     {
     while (len--!=0)
         {
          write((int)cbuf[off++]);          }
     }
    public void write(String str,int off,int len)
          throws IOException
     {
                while (len--!=0)
                {
                      write((int)str.charAt(off++));
                }
     }
     public static void main(String args[])
     {
     PrintWriter console = new PrintWriter(
          new UCWriter(new OutputStreamWriter(System.out)));
          console.println("hello there bunkie!");
```

```
        console.flush();
      }
}
```

The test **main** method converts the **System.out** stream to an **OutputStreamWriter** object and then attaches it to a **UCWriter**. Finally, the program attaches the entire set of writers and streams to a **PrintWriter** (which performs the formatting). The result is a stream that prints everything in uppercase to the system console.

Immediate Solutions

Creating a File Stream

Many programs need to open files for reading or writing. Of course, a file might really map to a device (such as a printer or the console); that depends on your operating system. Listing 6.2 shows how to use a **FileReader** object to read a file (and, in this case, display the contents on the console).

Listing 6.2: A simple file-reader program.

```
import java.io.*;


public class SimpleFileReader
{
   public static void main(String []args)
    {
       if (args.length != 1)
        {
           System.out.println("Please provide a file name.");
            System.exit(1);
        }
        String fileName = args[0];
```

TEAMFLY

Team-Fly®

```
        try
        {
            String fileLine;
          FileReader fileReader = new FileReader(fileName);
          BufferedReader bufferedReader =
              new BufferedReader(fileReader);
          while ((fileLine = bufferedReader.readLine()) != null)
          {
              System.out.print(fileLine);
          }
      }
     catch (IOException e) {
          System.out.println(e);
      }
  }
}
```

The highlighted lines are where all the stream action happens. You use the **FileReader** class to open up the file (**filename**) and read characters. Although they are all considered streams, remember that those classes that have **Reader** or **Writer** in their names take characters, whereas classes with **Stream** in their names take bytes.

To improve performance, the code connects a **BufferedReader** object to the **FileReader**. The program would work without this layer, but it would be less efficient because the **BufferedReader** object reads large chunks from the **FileReader** and buffers them for faster access.

The program in Listing 6.3 shows an example of both reading and writing to a file. The program requires two file names at the command prompt. The program then copies the contents from the first file into a newly created second file. If the second file already exists, it gets overwritten. Finally, the program reports the details on the second file, demonstrating a few of the **File** class methods:

Listing 6.3: A file stream demonstration program.

```
import java.io.*;


public class FileStreamDemonstration
{
   public static void main(String[] args) throws IOException
    {
         String readFileName = "";
         String writeFileName = "";
         readFileName = args[0];
        writeFileName = args[1];


         File readFile = new File(readFileName);
         File writeFile = new File(writeFileName);


//text files
        FileReader fileReader = new FileReader(readFile);
//text files
        FileWriter fileWriter = new FileWriter(writeFile);


//byte files
         //FileInputStream in = new FileInputStream(readFile);
//byte files
        //FileOutputStream out = new FileOutputStream(writeFile);
```

```
        int character = fileReader.read();
        while (character != -1)
        {
            fileWriter.write(character);
            character = fileReader.read();
        }
        if( writeFile.exists() )
        {
            fileReport(writeFile);
        }

        fileWriter.close();
        fileReader.close();
    }

    public static void fileReport(File fileObject)
    {
        System.out.println(fileObject.getName() +
                " has been created.");
        System.out.println(fileObject.getParent() +
                " = getParent()");
        System.out.println(fileObject.getPath() +" = getPath()");
        System.out.println(fileObject.canRead() +" = canRead()");
        System.out.println(fileObject.canWrite() +
                " = canWrite()");
```

```
        System.out.println(fileObject.isAbsolute() +
            " = isAbsolute()");
        System.out.println(fileObject.isDirectory() +
            " = isDirectory()");
        System.out.println(fileObject.isFile() +" = isFile()");
        System.out.println(fileObject.isHidden() +
            " = isHidden()");
       System.out.println(fileObject.length() +" = length()");

        System.out.println(fileObject.lastModified() +
            " = lastModified()");
       long currentTime = System.currentTimeMillis();
        System.out.println(fileObject.setLastModified(currentTime)
              + " = setLastModified(" + currentTime +")");
        System.out.println(fileObject.setReadOnly() +
                 " = setReadOnly()");
        System.out.println(fileObject.toString() +
                 " = toString()");
// Uncomment these lines to try renaming, mkdir, delete,
// creation, and directory listings

      //File newFile = new File("files/newFileName.txt");
       //fileObject.renameTo(newFile);
      //System.out.println(fileObject.mkdir() +" = mkdir()");
```

```
        //System.out.println(fileObject.delete() +
        //              " has been deleted.");
        //System.out.println(fileObject.createNewFile() +
        //                              " file has been created.");
       //String[] fileList = fileObject.list();
                    //for directory only
    }
}
//java FileStreamDemonstration
                    "files/inputfile.txt""files/outputfile.txt"
//copies inputfile.txt to outputfile.txt
//java FileStreamDemonstration
                    "files/inputfile.txt""files/outputfile.txt"
//copies inputfile.txt to outputfile.txt\
//returns:
//outputfile.txt has been created.
//files = getParent()
//files\outputfile.txt = getPath()
//true = canRead( )
//true = canWrite()
//false = isAbsolute()
//false = isDirectory()
//true = isFile()
//false = isHidden()
//0 = length() - Zero because file not closed yet!
```

```
//1000265656000 = lastModified()/ //false = setLastModified(1000265654434)

//true = setReadOnly()

//files\outputfile.txt = toString()
```

The file stream classes handle all the dirty work of figuring out the native file directory system. You can append text to a file by passing **true** as the **FileWriter** constructor's second argument. This indicates that you want to append the data you write to the existing file. Here's an example:

```
BufferedWriter bufferedWriter = new BufferedWriter(
        new  FileWriter("output.txt", true));
PrintWriter fileWriter = new PrintWriter(bufferedWriter);
fileWriter.println("This text is appended.");
fileWriter.write("neurosymphany", 5, 3);// append to file 'sym'.
fileWriter.close();
```

As a final example, consider Listing 6.4. This program combines many techniques and classes covered previously in the book. The program counts all the unique words in all the files in a directory (you give the directory name as a command-line argument):

Listing 6.4: A SequenceInputStream demonstration program.

```
import java.util.*;


import java.io.*;


public class DirectoryWordFrequency

{

   public static void main(String[] args) throws IOException

    {

       if (!(args.length > 0))

        {

          System.out.println("Please provide a directory name.");
```

```
        System.exit(1);
    }

    String directoryName = args[0];
    File readFile = new File(directoryName); // directory
    String[] files = readFile.list(); // files in directory
    for (int i=0; i<files.length; i++)
    {
        files[i] = directoryName +"\\" + files[i];
        System.out.println(files[i]);
    }

    FileEnumerator fileEnumerator = new FileEnumerator(files);
    SequenceInputStream allFiles =
            new SequenceInputStream(fileEnumerator);

    HashMap wordMap = new HashMap();
    StringBuffer subToken = new StringBuffer();
    String token ="";
    boolean isWord = false;
    int wordCount = 0;
    int nextLetter;

    while ((nextLetter = allFiles.read()) > -1)
    {
```

```
            char letter = (char)nextLetter;


             if ( (letter >= 'a' && letter <= 'z') ||
                        (letter >= 'A' && letter <= 'Z') )
            { // its a letter
                 isWord=true;
                 subToken.append(letter);
            } else
             {
                if (isWord)
               { //put word into hashtable
                    token = subToken.toString();
                     subToken.setLength(0);
//only I or a are single-letter words
                     if (token.length()==1)                    {
                        if (!(token.equalsIgnoreCase("i") ||
                             token.equalsIgnoreCase("a"))) token="";
                     }


                    if (token!=null & token!="")
                    {//token==null shouldn't happen
//faster than csounting from the Map directly
                         ++wordCount;
                         if(wordMap.containsKey(token))
                         {
```

```
                    ((Frequency)wordMap.get(token)).frequency++;
                }  else
                {
                    wordMap.put(token, new Frequency());
                }
            }
        }

        isWord=false;
      }
    }//end while
    System.out.println("wordCount =" + wordCount);
     System.out.println(wordMap);
  }
}

class Frequency
{
  int frequency = 1;
  public String toString()
  {
     return Integer.toString(frequency);
  }
}
//java DirectoryWordFrequency files
```

```
//returns:

//files\genesis_1.txt

//files\Matthew_1.txt

//wordCount = 1263

//{gathered=2, appeared=1, Hezekiah=2, Shealtiel=2,

                quietly=1,  father=39,

//All=1, how=1, exile=4, mark=1, record=1, mother=5,

                land=5,  Christ=4,  air=3,

//... removed for clarity

//r=1, generations=1, And=25, Jesus=5, Tamar=1,

                creatures=7,  Eliud=2,  whom=1}
```

TEAMFLY

This listing used the **SequenceInputStream** class. This stream simply joins multiple streams together sequentially. Using this object allows the rest of the program to process one long file instead of handling each input file separately.

The class in Listing 6.5 is adapted from a sample in the Java documentation. This class is required by the word-counting program. The **FileEnumerator** class allows you to enumerate the files in a directory. The enumeration does not continue into subdirectories, although you could write the program recursively to descend into subdirectories.

Listing 6.5: This class enumerates the files in a directory.

```
import java.util.*;

import java.io.*;


public class FileEnumerator implements Enumeration

{

    private String[] listOfFiles;

   private int fileCount = 0;
```


Team-Fly®

```
public FileEnumerator(String[] listOfFiles)
{
    this.listOfFiles = listOfFiles;
}

public boolean hasMoreElements()
{
    if (fileCount < listOfFiles.length)
    {
        return true;
    } else
    {
        return false;
    }
}
public Object nextElement()
{
   InputStream fileStream = null;

    if (!hasMoreElements())
    {
      throw new NoSuchElementException("No more files.");
    } else
    {
```

```
            String nextElement = listOfFiles[fileCount];
             fileCount++;

             try
             {
                fileStream = new FileInputStream(nextElement);
            } catch (FileNotFoundException e)
             {
                 System.err.println("Can't open" + nextElement);
             }
          }

          return fileStream;
      }
}
```

Rewinding a Stream

Typically, once you read data from a stream, the data is gone. You can't randomly access the stream's data at an arbitrary position, but some streams do allow you to mark a position and return to it later. The **mark** method marks the current position of the stream for later use. Later, you can call **reset** to return to that mark. The **mark** method takes a limit argument (an integer). This limit tells Java to invalidate the mark if the program reads the specified number of characters past the mark. This is important because Java may have to buffer everything from the mark to the current position in order to make **reset** work properly.

Not all streams support the **mark** method. Use **markSupported** to determine if you can use **mark** at runtime. A return value of **true** means you can use **mark** safely.

The program in Listing 6.6 illustrates the use of **mark** and **reset**. The program opens the file and determines its size. The program then uses the **skip** method to jump to a point you specify on the command line (use a percentage). At the specified position, the stream is marked using the **mark**

method. Now, the program prints the rest of the file to the console. Finally, the program issues a **reset** call to jump back to the mark, and then it prints the entire file.

Listing 6.6: A skip and mark program.

```
import java.io.*;


public class SkipMarkDemonstration {

    public static void main(String args[]) throws Exception {

        if (args.length != 2)

         {

            System.out.println("Type <fileName> <percentage>.");

             System.exit(1);

         }

          String fileName = args[0];

        int percentage = Integer.parseInt(args[1].trim());


        File file = new File(fileName);

         BufferedInputStream inputStream =

            new BufferedInputStream(new FileInputStream(file));


        int length= (int) file.length();

          System.out.println(fileName +" length =" + length);

         int skipTo = (int)((float)length * (percentage * .01F));

        System.out.println("skipTo=" + skipTo);

          inputStream.skip(skipTo);
```

```
        if (inputStream.markSupported())
        {
              inputStream.mark(length);
        }

        int tempCharacter;
       while ((tempCharacter = inputStream.read()) > 0)
        {
            System.out.print((char) tempCharacter);
        }

        System.out.println("\nread from mark:");
        inputStream.reset();
       while ((tempCharacter = inputStream.read()) > 0)
        {
            System.out.print((char) tempCharacter);
        }
    }
}
//java SkipMarkDemonstration"files\genesis_1xs.txt" 99
//returns:
//files\genesis_1.txt length = 4104
//skipTo=4062
//, and there was morning--the sixth day.
//
```

```
//read from mark:

//, and there was morning--the sixth day.
```

Filtering Streams

One common stream operation is creating a filter class. This class is really no different from any other stream class except that a filter class's input is one stream and its output is another stream. For example, you might want to connect a **PrintWriter** object to a special class that converts all the characters written to uppercase. The uppercase class then connects to a **FileWriter** that actually stores the data in a file. The class in the middle is a filter.

Listing 6.7 shows a class, **HackOutputStream**, that performs this filtering on a byte stream (you can find a similar class for writers in Listing 6.1). The example program (in **StreamFilterDemonstration**) writes its output to the file outputcase.txt.

Listing 6.7: A filtering stream program.

```
import java.io.*;


public class StreamFilterDemonstration

{

    public static void main(String args[]) // test driver

    {

       int caseChoice = HackOutputStream.CAPITAL;

       if (args.length != 2)

        {

           System.out.println("Type <fileName> <case choice>.");

            System.exit(1);

        }

        String fileName = args[0];

       caseChoice = Integer.parseInt(args[1].trim());
```

```
if ( (caseChoice > 3) || (caseChoice < 1) )
 {
     caseChoice = HackOutputStream.CAPITAL;
 }


 try
 {
    FileOutputStream output =
                        new FileOutputStream("outputCase.txt");
    HackOutputStream hackStream =
                       new HackOutputStream(output, caseChoice);
   FileInputStream fis = new FileInputStream(fileName);
   BufferedInputStream bs = new BufferedInputStream(fis);
   byte[] b = new byte[1000];
    bs.read(b, 0, 1000);


     hackStream.write(b);
     hackStream.flush();
     hackStream.close();
 }
catch(FileNotFoundException e)
 {
     System.out.println(e);
 }
```

```
        catch(IOException e)
        {
            System.out.println(e);
        }
    }
}

// Output stream that can force different cases
class HackOutputStream extends FilterOutputStream
{
   public static final int UPPERCASE = 1;
    public static final int LOWERCASE = 2;
   public static final int CAPITAL = 3;
    private int changeFlag;
   private boolean lastByteWasNonLetter = true;

   public HackOutputStream(OutputStream stream, int choice)
    {
       super(stream);//send to superclass

        this.changeFlag = choice;
    }

// All the work occurs here
   public void write(int streamByte) throws IOException
```

```
{
    switch (changeFlag) // do the requested transform
    {
        case UPPERCASE:
            out.write( (int)Character.toUpperCase(
                (char)streamByte ) );
            break;
        case LOWERCASE:
            out.write( (int)Character.toLowerCase(
                (char)streamByte ) );
            break;
        case CAPITAL:
            if (Character.isLetter( (char)streamByte ) )
            {
                if (lastByteWasNonLetter)
                {
                    out.write( (int)Character.toUpperCase(
                        (char)streamByte ) );
                } else
                {
                    out.write( (int)Character.toLowerCase(
                        (char)streamByte ) );
                }

                lastByteWasNonLetter = false;
```

```
            } else
            {
                lastByteWasNonLetter = true;
                out.write( streamByte );
            }
            break;
        }
    }
}
```

//use: java StreamFilterDemonstration"files\input.txt" 3

Related solution:	Found on page:
Understanding Inheritance	108

Reading Words or Groups

When you are dealing with files, you often don't want to handle bytes or characters. Instead, you want to read either Java objects (covered later) or groups of characters. For example, if you were writing a program to format Java code, you'd want to read the Java keywords, variable names, and so forth. Reading groups of characters is often known as *tokenizing*, and Java provides the **StreamTokenizer** class for this purpose. By default, this class splits words on white-space boundaries. It also differentiates between words and numbers. You can even make the class handle quoted strings and Java comments. Listing 6.8 shows a word-counting program that uses the **StreamTokenizer** class.

Listing 6.8: A StreamTokenizer demonstration program.

```
import java.util.*;

import java.io.*;
```

```
public class streamTokenizerDemonstration
{
   public static void main(String arg[])
   {
       BufferedReader br = null;
       StreamTokenizer token = null;

        try
        {
           br = new BufferedReader(new
                 FileReader("files\\genesis_1.txt"));
          token = new StreamTokenizer(br);
       } catch(FileNotFoundException e)
        {
           System.out.println("File Not Found");
        }
        HashMap wordMap = new HashMap();
       int wordCount = 0;

        try
        {
// Check for End of File (EOF)
       while (token.nextToken() != StreamTokenizer.TT_EOF)
        {
```

TEAMFLY

Team-Fly®

```
// See if the token is a word
            if (token.ttype == StreamTokenizer.TT_WORD)
             {
                 ++wordCount;
                     //faster than counting from the Map directly
                 if(wordMap.containsKey(token.sval))
                 {
                       ((Frequency)wordMap.get(token.sval)).frequency++;
                 } else
                 {
                       wordMap.put(token.sval, new Frequency());
                 }
           }// else if (token.ttype == StreamTokenizer.TT_NUMBER)
             //{
            //    do something with number: token.nval
             //}
         }
    } catch(IOException e)
    {
     e.printStackTrace();
    }
    int uniqueWords = wordMap.size();
    int newWordRate = wordCount/uniqueWords;
    System.out.println("total words =" + wordCount);
    System.out.println("unique words =" + uniqueWords);
```

```
    if(newWordRate > 5)
    {
        System.out.println("Please use more vocabulary");
    }
   }
}

class Frequency
{
    int frequency = 1;
    public String toString()
    {
        return Integer.toString(frequency);
    }
}
//returns:
//total words = 469
//unique words = 136
// and the individual word counts
```

Processing Binary Data

Although you usually think of streams as containing characters (either bytes or Unicode characters), you can also process raw data in streams. Of course, it doesn't make sense to deal with Unicode streams and binary data (unless you plan to write textual representations of the numbers). Instead, you'll use **DataInputStream** and **DataOutputStream**.

These two classes are filters; you'll use an **InputStream** or **OutputStream** in the constructor when you instantiate the class. Then you can use methods like **readByte**, **readFloat**, or **writeInt**. Here's a simple example:

```
try {
   FileInputStream fis = new FileInputStream("test.txt");
    DataInputStream dis = new DataInputSTream(fis);
    int x;
    float y;
     x=dis.readInt();
     y=dis.readFloat();
  }
```

Naturally, it is up to you to read and write the data in the correct order. Java will dutifully read an integer when you ask it to, even if the program that wrote the file placed a floating-point number at that point in the file.

Of course, a more object-oriented method for writing raw data to a file is serialization, which is coming up next.

Serializing Objects

Java I/O can also serialize objects. *Serialization* is the process of saving to disk the state of an object with the intention of reconstructing that object in the same state. The serialization classes can read or write a whole object to and from a raw byte stream. Serialization allows Java objects and primitives to be stored and rebuilt.

The **Serializable** interface (**java.io.Serializable**) identifies a class that supports serialization. Interestingly, this interface is just a marker, so it doesn't have any methods. Any class that says it implements **Serializable** does implement it because it has no methods at all. The code in Listing 6.9 demonstrates a simple example of serializing a few objects and then deserializing (reconstructing) them.

Listing 6.9: A serialization demonstration program.

```
import java.io.*;

import java.util.*;



public class SerializeDemonstration

{

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

```
{
    try
    {
        String[][] employeeNames =
        {
            {"Jan","Frank","Wess","Pat","Donald" },
            {"Stan","Beth","Harold","Kevin" },
            {"Harold","Greg","Les","Karen","Tom","Abe" },
            {"Pete","Claire","Seth","Arnold","Abdul" }
        };

      FileOutputStream f = new FileOutputStream("files \\tmp");
       ObjectOutputStream oldObject =
                new ObjectOutputStream(f);
      oldObject.writeObject("I have been serialized...");
        oldObject.writeObject(employeeNames);
       oldObject.writeObject(new Date());
        oldObject.flush();

      //deserialize objects from file
      FileInputStream in = new FileInputStream("files \\tmp");
      ObjectInputStream newObject = new ObjectInputStream(in);
      String oldString = (String)newObject.readObject();
       String stringArray[][] =
           (String[][]) newObject.readObject();
```

```
          Date date = (Date)newObject.readObject();
          System.out.println(oldString);
          System.out.println(date);

          int dim_1_Length = stringArray.length;
        for (int dim_1 = 0; dim_1 < dim_1_Length; dim_1++)
          {
            int dim_2_Length = stringArray[dim_1].length;
           for (int dim_2 = 0; dim_2 < dim_2_Length; dim_2++)
             {
                System.out.print(stringArray[dim_1][dim_2] +"");
             }
             System.out.println();
          }
      } catch (Exception ex)
       {
          ex.printStackTrace();
       }
   }
}
//returns:
//I have been serialized...
//Thu Sep 13 01:05:54 PDT 2001
//Jan Frank Wess Pat Donald
//Stan Beth Harold Kevin
```

```
//Harold Greg Les Karen Tom Abe

//Pete Claire Seth Arnold Abdul
```

You should see that the class is doing a lot behind the scenes. Serialization is important anytime you want to store an object and reconstitute it later. Of course, this includes saving and reading from files. Serialization is also useful for passing objects across a network. Because an ordinary stream can write to a **String**, you can create a **String** that represents a serialized object that you can store, for example, in a database.

Related solution:	**Found on page:**
Serializing an Object	284

Tracking Line Numbers

Java's **LineNumberReader** class reads a file and tracks the line number it is reading. This is useful when you want to report an error in the file, for example. You can include the offending line number with the error. The class counts lines by marking the line terminators. A line terminator is a line feed (**'\n'**), a carriage return (**'\r'**), or a carriage return followed immediately by a line feed. Otherwise, this class is identical to a **BufferedReader** class (which it extends). Listing 6.10 shows how to use the **LineNumberReader** class.

Listing 6.10: A StreamLineNumber demonstration program.

```
import java.io.*;


public class StreamLineNumber
{
   public static void main(String args[])
    {
      if (args.length != 1)
       {
          System.out.println("Please provide a file name");
```

```
            System.exit(1);
        }


        try
        {
            LineNumberReader reader =
                    new LineNumberReader(new FileReader(args[0]));
            String text;
           while ((text = reader.readLine()) != null) {
                    System.out.println(reader.getLineNumber() +
                       " " + text);
            }
        }
        catch(IOException e) {
            System.out.println(e);
        }
    }
}
// java StreamLineNumber StreamLineNumber.java
//returns:
//1
//2   import java.io.*;
//3
//4   public class StreamLineNumber
//5   {
```

```
//6      public static void main(String args[])
//7       {
//8            if (args.length != 1)
//... remainder removed to save space
```

Using Random File Access

You can use the **RandomAccessFile** class for random reading and writing of files. The program in Listing 6.11 shows you how to use this class. Unlike other classes in this chapter, **RandomAccessFile** does not extend any of the other stream classes. It is a completely separate way to handle files.

Listing 6.11: A random access demonstration program.

```
import java.io.*;


class RandomAccessFileDemonstration
{
   public static void main (String[] args)
    {
        File  file;
        RandomAccessFile raf;


    try
    {
        file = new File ("files\\append.txt");
        raf = new RandomAccessFile (file,"rw");
          raf.writeBytes(
```

```
            "The RandomAccessFile class implements both \r\n");
          raf.writeBytes(
           "the DataInput and DataOutput interfaces and \r\n");
          raf.close ();

         file = new File ("files\\append.txt");
         raf = new RandomAccessFile (file,
              "rw");
          raf.seek (raf.length ());
        raf.writeBytes("therefore can be used for both reading" +
                          " and writing.\r\n");
           raf.close ();
    } catch (Exception e)
     {
         System.out.println ("RandomAccessFile:" + e);
     }
 }
}
//Creates a file with this in it:
// The RandomAccessFile class implements both
//the DataInput and DataOutput interfaces and
//therefore can be used for both reading and writing.
```

The **RandomAccessFile** class uses the same methods to read and write that **DataInputStream** and **DataOutputStream** use. In fact, it implements the same interfaces: **DataInput** and **DataOutput**. However, this class also allows you to call **seek** to set the current file position to any arbitrary position.

Chapter 7: Java Database Connectivity

In Brief

Computer programs manipulate data. Because data typically resides in databases, it's a good bet you'll need to write a program that accesses a database. The problem is: which database? There are probably a dozen major players in the database game and many lesser-known databases in use. Do you really want to learn all of them?

JDBC Overview

Fortunately, you don't have to learn all of the databases. Java provides a package known as Java Database Connectivity (JDBC), which acts as an object-oriented wrapper around most common databases. If you know how to work with JDBC, you can work with practically every database you might encounter.

JDBC under Windows often cooperates with another high-level wrapper: the Open Database Connectivity (ODBC) standard. ODBC is the Windows abstraction of databases. This standard increases the number of databases that JDBC can use because many database vendors provide ODBC drivers.

This chapter focuses on the more important classes and techniques of JDBC and on how to use Java with JDBC; the chapter will not focus on specific database issues. General database theory deserves at least a book of its own (and many database books are available). To get the most from this chapter, you should already know something about tables, SQL (Structured Query Language), and simple queries, which are database issues and are not really related to Java.

Consider a real program. The code in Listing 7.1 connects to a database. You simply supply an ODBC data source name on the command line. For example, you might use the Northwind example database that ships with Microsoft Access (if you are using Windows).

Listing 7.1: Connecting to a database.

```
import java.sql.*;


public class JDBCsimple

{

   public static void main (String args[])

    {
```


Team-Fly®

```
if (args.length != 1)
 {
      System.out.println("Please provide an ODBC name.");
      System.exit(1);
 }
 String dataSource = args[0];

 String username = " ";
 String password = " ";
 String source = "jdbc:odbc:";
source += dataSource;

 try
 {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (Exception e)
 {
   System.out.println("JDBC/ODBC driver failure.");
    return;
 }

 try
 {
     Connection conn =
                        DriverManager.getConnection(source,
```

```
                        username, password);
            System.out.println("Connected to " + dataSource);
            DatabaseMetaData dmd = conn.getMetaData();

            if (dmd == null)
             {
             System.outprintln ("No database meta data available");
            } else
             {
                 System.out.println ("Database Product: " +
                                        dmd.getDatabaseProductName());
                 System.out.println ("Product Version: " +
                                      dmd.getDatabaseProductVersion());
             }

             conn.close();
        } catch (Exception e)
         {
             e.printStackTrace();
         }
    }
}
```

The previous program constructs a database URL of the form **jdbc: subprotocol:subname**. This URL allows access to the database.

Not all Java installations include JDBC, and not all JDBC installations have the particular driver you want. Therefore, the program tests to be sure that the ODBC-to-JDBC bridge is present. That's the purpose of the following line:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

If the JDBC class is not present, this statement will throw a **ClassNot-FoundError** exception.

The actual database code begins in the second **try** block. The **Connection** object creates a session with the database. Through this connection, you can execute SQL statements, retrieve results, and get information describing the database. For example, you might want to know if the database supports a particular operation (such as **UPDATE**).

One thing to beware of with connection objects: They automatically commit changes after executing each statement. You can override this behavior by turning off the autocommit feature (simply pass **false** to the **setAutoCommit** method). If you do disable this feature, you'll have to call the **commit** method to save any database changes.

> **Note** If the autocommit feature is turned off, you will lose all changes unless you explicitly commit transactions.

Next, the program calls the **getMetaData** method. This call provides the information about this particular database. The call returns a **DatabaseMetaData** object, which has many methods describing details about the database (for example, the database engine in use and the version number). Notice that the program isn't querying the data itself—that will come shortly.

JDBC 3

Sun Microsystems ships JDBC 3 with Java 1.4. JDBC 3 supersedes both the JDBC 1.22 and JDBC 2.1 specifications. Most companies are using JDBC 2.1 or even 1.22. If you are using one of these older versions, dump it and go with JDBC 3 (which is compatible with JDBC 2). When you upgrade, you get three things:

- *Performance*—JDBC 3 is faster, is more reliable, and fixes bugs.
- *SQL-99 compliance*—JDBC doesn't support the entire standard, but the SQL subset that it does support complies with the standard.
- *More features*—JDBC 3 offers advanced features (such as named parameters and connection pooling) not available in earlier releases.

Remember that JDBC will help you talk to the database. JDBC is not a database itself. Therefore, JDBC addresses what you can do with the data after the database hands it to you. Java can't do anything about what the database does with the data you give it. This brings up a subtle, but significant, issue. Is database functionality in the database or in the application? To answer that question, you have to look at the SQL-99 specification.

The SQL Standard

Starting in 1978, the H2 committee (part of ANSI, the American National Standards Institute) started working on a standard for databases. After much bureaucratic processing, the standard became an American National Standard in 1986. Since then, this committee has given us SQL-89, SQL-92, and the latest standard, SQL-99.

The purpose of a standard like SQL-99 is to allow database users to switch between different database systems without fear of incompatibility. Most major database vendors (including Oracle, Microsoft, and IBM) comply with SQL-99, although many offer their own nonstandard extensions.

What this means is that JDBC is like a telephone—you can use a telephone to call anyone in the world. Once you connect, though, there is no guarantee that you'll understand the language of the person on the other end. Consider this query:

```
SELECT TOP 10 PERCENT
  LastName, Department
FROM Employees
WHERE BirthYear > 1970
ORDER BY Salary DESC;
```

This query selects names and departments of employees who were born after 1970. The **ORDER BY** clause sorts the results in descending order (because of the **DESC** keyword). The **SELECT TOP 10 PERCENT** statement selects only the top 10 percent of the results.

This query works with Microsoft SQL Server and Access but not with most other database systems. If you stick to the SQL-99 standard (which does not include keywords like **PERCENT**), your database operations should be portable. If you must use nonstandard operations, you can abstract some of your JDBC code into vendor-neutral base components and then extend these with vendor-specific utility components.

Note Please see ISO/IEC 9075:1999(E) Information technology—Database languages— SQL. This document is the standard referred to as SQL-99. Go to http://www.iso.org/iso/en/prodsservices/catalogue/intstandards/CatalogueListPage.CatalogueList and search for SQL. The results will list all the standards you might be interested in, but you have to pay to view them. On the Java side, you can look at the JDBC specification, titled "JSR-000054 JDBC 3.0 Specification," at http://www.jcp.org/jsr/detail/54.jsp.

Driver-Database Discrepancies

One other problem you might encounter is that of slight syntax differences. The SQL that a driver understands is sometimes different from what the database is expecting. The driver and the database don't always speak the same language. Mostly they do, of course, but not exactly. For example, Access uses an asterisk (*) for a wildcard character. The wildcard matches any number of characters ("wh*" finds "what," "white," and "why"). But the ODBC driver that ships with Windows doesn't accept this character. Instead, you must use the percent sign (%). So code you develop for a native database application might require modifications before passing through an ODBC driver (or through a JDBC program that uses an ODBC driver).

You might wonder why you'd even bother writing native SQL code instead of ODBC code. There are several reasons. First, ODBC is not especially efficient. You might have a choice of using a native JDBC driver that directly interfaces with the database. Also, it is possible to partition your program so that some of the database work occurs on the database server.

For example, most modern database programs support stored procedures. Your JDBC program might do little more than trigger a stored procedure to carry out some operation. The stored procedure would use the native SQL syntax because it executes directly on the server.

When you are executing a program in a client/server environment, placing code in stored procedures might increase performance because less network traffic is required for this approach. However, stored procedure code makes your program more of a hybrid and less of a Java program. This can make maintenance more complex.

Types of Drivers

You'll often hear people refer to a *JDBC driver type*. This is an arbitrary division of drivers based on their internal workings. A type 1 driver, for example, is the JDBC-to-ODBC bridge we discussed earlier in this chapter.

A type 2 driver, on the other hand, is a Java class that calls a native database API (for example, client APIs for Oracle or Sybase). These are not very different from type 1 drivers; they still use an external, non-Java bridge to work with the database.

Type 3 drivers are all Java and use a network transport to communicate with a database server using a database-independent protocol. Since many database vendors offer this network service, a type 3 driver is a flexible option. The driver can be pure Java and still interface with a proprietary database format.

Finally, there are type 4 drivers. These classes directly interface to a proprietary network protocol. They are specific for a particular database, as you would expect.
You can search for different drivers based on their type and levels of support at the Sun Web site (see http://www.industry.java.sun.com/products/jdbc/drivers).

Immediate Solutions

Interrogating a Database with the DatabaseMetaData Class

JDBC 3 has greatly expanded the database interrogation capabilities of Java. The program in Listing 7.2 doesn't need much commentary thanks to Sun's descriptive method names. The program connects to the database and then prints what information it can find. Calling **getMetaData** returns a **DatabaseMetaData** object that contains plenty of information about the database. You simply make method calls to retrieve the information you want.

In a real program, you'd probably use only a few of these methods to find the information you needed to tailor your processing to the database you were using.

Listing 7.2: Interrogating a connection.

```
import java.sql.*;
```

```
public class DatabaseMetaDataDemo
{
   public static void main (String args[])
    {
       if (args.length != 1)
        {
              System.out.println("Please provide an ODBC name.");
            System.exit(1);
        }
        String dataSource = args[0];

        String username = " ";
        String password = " ";
        String source = "jdbc:odbc:";
       source += dataSource;

        try
        {
           Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       } catch (Exception e)
        {
              System.out.println("JDBC/ODBC driver failure.");
           return;
        }
```

```
try
{
   Connection conn = DriverManager.getConnection(source,
         username, password);
      System.out.println("Connected to " + dataSource);
   DatabaseMetaData dmd = conn.getMetaData();

   if (dmd == null)
    {
          System.out.println(
           "No database meta data available");
    } else
    {
        /*
         * Constants indicating the cursor type for
         * a ResultSet object cursor may move only forward
         * use ResultSet.TYPE_FORWARD_ONLY
        */      final int TYPE_FORWARD_ONLY = 1003;

    /* use ResultSet.TYPE_SCROLL_INSENSITIVE
     * scrollable but generally not sensitive to changes
     * made by others.
     */       final int TYPE_SCROLL_INSENSITIVE = 1004;
```

```
            /* use ResultSet.TYPE_SCROLL_SENSITIVE
             * scrollable and generally sensitive to
             * changes made by others.
            */      final int TYPE_SCROLL_SENSITIVE = 1005;



          // display all database properties
           displayDBproperties(dmd, TYPE_FORWARD_ONLY );
        }

         conn.close();
    } catch (Exception e)
     {
         e.printStackTrace();
     }
}
static void displayDBproperties (DatabaseMetaData dbMetaData,
       int cursor_type)
{
     try
     {
        boolean true_false = false;

         System.out.println ("Database Product Name : " +
                                dbMetaData.getDatabaseProductName());
```

```
 true_false = dbMetaData.isCatalogAtStart();
 System.out.println( true_false +
            ": Does a catalog appear at " +
            " the start of a qualified table name? " +
            " (Otherwise it appears at the end) ");
true_false = dbMetaData.isReadOnly();
System.out.println( true_false +
                        ": Is the database in " +
                          "read-only mode?");
true_false = dbMetaData.nullPlusNonNullIsNull();
System.out.println( true_false +
      ":    Are concatenations between " +
        " NULL and non-NULL values NULL? For SQL-92 " +
      " compliance, a JDBC technology-enabled " +
      " driver will return true.");
true_false = dbMetaData.nullsAreSortedAtEnd();
System.out.println( true_false +
       ":  Are NULL values sorted at the " +
         " end regardless of sort order?");
 true_false = dbMetaData.nullsAreSortedAtStart();
System.out.println( true_false +
          ":  Are NULL values sorted at " +
          "the start regardless of sort order?");
true_false = dbMetaData.nullsAreSortedHigh();
 System.out.println( true_false +
```

```
        ": Are NULL values sorted high?");
    true_false = dbMetaData.nullsAreSortedLow();
    System.out.println( true_false +
        ":    Are NULL values sorted low?");
    true_false = dbMetaData.storesLowerCaseIdentifiers();
    System.out.println( true_false +
         ":  Does the database treat mixed " +
        " case unquoted SQL identifiers as case " +
         " insensitive and store them in lower case?");
     true_false =
       dbMetaData.storesLowerCaseQuotedIdentifiers();
    System.out.println( true_false +
      ":  Does the database treat mixed " +
     " case quoted SQL identifiers " +
     "as case insensitive and store them in lower case?");
    true_false = dbMetaData.storesMixedCaseIdentifiers();
    System.out.println( true_false +
              ": Does the database treat mixed " +
                 " case unquoted SQL identifiers as case " +
                 " insensitive and store them in mixed case?");
     true_false =
         dbMetaData.storesMixedCaseQuotedIdentifiers();
    System.out.println( true_false +
        ": Does the database treat mixed " +
        " case quoted SQL identifiers as case" +
```

TEAMFLY

Team-Fly®

```
        " insensitive and store them in mixed case?");
      true_false = dbMetaData.storesUpperCaseIdentifiers();
      System.out.println( true_false +
        ": Does the database treat mixed " +
        " case unquoted SQL identifiers " +
         "as case insensitive and " +
         "store them in upper case?");
      true_false =
        dbMetaData.storesUpperCaseQuotedIdentifiers();
      System.out.println( true_false +
        ": Does the database treat mixed " +
         " case quoted SQL identifiers " +
         "as case insensitive and store them in " +
         "upper case?");
      true_false =
         dbMetaData.supportsAlterTableWithAddColumn();
      System.out.println( true_false +
              ": Is \"ALTER TABLE\" with add " +
             " column supported?");
      true_false =
         dbMetaData.supportsAlterTableWithDropColumn();
      System.out.println( true_false +
              ": Is \"ALTER TABLE\" with drop " +
             " column supported?");
    true_false = dbMetaData.supportsANSI92EntryLevelSQL();
```

```
System.out.println( true_false +
                    ":  Is the ANSI92 entry level SQL " +
                     " grammar supported? All JDBC " +
                      "Compliant drivers must return true.");
true_false = dbMetaData.supportsANSI92FullSQL();
System.out.println( true_false +
      ":  Is the ANSI92 full SQL grammar supported?");
 true_false =
    dbMetaData.supportsANSI92IntermediateSQL();
System.out.println( true_false +
          ": Is the ANSI92 intermediate " +
           " SQL grammar supported?");
true_false = dbMetaData.supportsBatchUpdates();
System.out.println( true_false +
             ": Indicates whether the driver " +
            " supports batch updates.");
 true_false =
        dbMetaData.supportsCatalogsInDataManipulation();
System.out.println( true_false +
                   ":   Can a catalog name be used in " +
                " a data manipulation statement?");
 true_false =
     dbMetaData.supportsCatalogsInIndexDefinitions();
System.out.println( true_false +
        ":  Can a catalog name be used in " +
```

```
            " an index definition statement?");
        true_false =
            dbMetaData.supportsCatalogsInPrivilegeDefinitions();
        System.out.println( true_false +
            ": Can a catalog name be used in " +
            " a privilege definition statement?");
        true_false =
            dbMetaData.supportsCatalogsInProcedureCalls();
        System.out.println( true_false +
            ": Can a catalog name be used in " +
            " a procedure call statement?");
        true_false =
            dbMetaData.supportsCatalogsInTableDefinitions();
        System.out.println( true_false +
            ": Can a catalog name be used in " +
            " a table definition statement?");
        true_false = dbMetaData.supportsColumnAliasing();
        System.out.println( true_false +
            ": Is column aliasing supported?");
        true_false = dbMetaData.supportsConvert();
        System.out.println( true_false +
                ": Is the CONVERT function " +
                " between SQL types supported?");
        true_false = dbMetaData.supportsCoreSQLGrammar();
        System.out.println( true_false +
```

```
            ":  Is the ODBC Core SQL grammar supported?");
        true_false = dbMetaData.supportsCorrelatedSubqueries();
        System.out.println( true_false +
                ": Are correlated subqueries " +
                " supported? A JDBC Compliant" +
                " driver always returns true.");
        true_false =
          dbMetaData.
    supportsDataDefinitionAndDataManipulationTransactions();
        System.out.println( true_false +
                ": Are both data definition and " +
                  " data manipulation " +
                 "statements within a transaction supported?");
        true_false =
            dbMetaData.supportsDataManipulationTransactionsOnly();
        System.out.println( true_false +
                ": Are only data manipulation " +
                  " statements within a transaction supported?");
        true_false =
            dbMetaData.supportsDifferentTableCorrelationNames();
        System.out.println( true_false +
                ": If table correlation names " +
               " are supported, are they restricted to be " +
               " different from the names of the tables?");
        true_false = dbMetaData.supportsExpressionsInOrderBy();
```

```
System.out.println( true_false +
        ": Are expressions in \"ORDER " +
         "BY\" lists supported?");
true_false = dbMetaData.supportsExtendedSQLGrammar();
System.out.println( true_false +
        ": Is the ODBC Extended SQL " +
         " grammar supported?");
true_false = dbMetaData.supportsFullOuterJoins();
System.out.println( true_false +
                    ":  Are full nested outer "
                      " joins supported?");
true_false = dbMetaData.supportsGroupBy();
System.out.println( true_false +
        ": Is some form of " +
          "\"GROUP BY\" clause supported?");
true_false =
    dbMetaData.supportsGroupByBeyondSelect();
System.out.println( true_false +
        ":  Can a \"GROUP BY\" clause " +
      " add columns not in the SELECT provided it " +
        " specifies all the columns in the SELECT?");
true_false = dbMetaData.supportsGroupByUnrelated();
System.out.println( true_false +
        ":  Can a \"GROUP BY\" clause " +
       " use columns not in the SELECT?");
```

```
true_false =
      dbMetaData.supportsIntegrityEnhancementFacility();
System.out.println( true_false +
            ": Is the SQL Integrity " +
             " Enhancement Facility supported?");
true_false = dbMetaData.supportsLikeEscapeClause();
System.out.println( true_false +
       ": Is the escape character in " +
        " \"LIKE\" clauses supported? " +
      "A JDBC Compliant driver always returns true.");
true_false = dbMetaData.supportsLimitedOuterJoins();
System.out.println( true_false +
        ": Is there limited support for " +
       " outer joins? (This will be true " +
        "if supportFullOuterJoins is true.");
true_false = dbMetaData.supportsMinimumSQLGrammar();
System.out.println( true_false +
          ": Is the ODBC Minimum SQL " +
           "grammar supported? All JDBC " +
           "Compliant drivers must return true.");
true_false = dbMetaData.supportsMixedCaseIdentifiers();
System.out.println( true_false +
 ": Does the database treat mixed " +
   " case unquoted SQL identifiers " +
 "as case sensitive and as a result store them " +
```

```
 "in mixed case? A JDBC " +
 "Compliant driver will always return false.");
true_false =
    dbMetaData.supportsMixedCaseQuotedIdentifiers();
System.out.println( true_false +
 ": Does the database treat mixed " +
 " case quoted SQL identifiers " +
 "as case sensitive and as a result store them " +
 "in mixed case? A JDBC " +
 "Compliant driver will always return true.");
true_false =
    dbMetaData.supportsMultipleResultSets();
System.out.println( true_false +
    ": Are multiple ResultSet from a " +
   " single execute supported?");
true_false = dbMetaData.supportsMultipleTransactions();
System.out.println( true_false +
    ": Can we have multiple " +
    "transactions open at once (on different " +
      " connections)?");
true_false = dbMetaData.supportsNonNullableColumns();
System.out.println( true_false +
      ": Can columns be defined as " +
         "non-nullable? A JDBC Compliant " +
         "driver always returns true.");
```

```
true_false =
    dbMetaData.supportsOpenCursorsAcrossCommit();
System.out.println( true_false +
    ": Can cursors remain open " +
    "across commits?");
true_false =
    dbMetaData.supportsOpenCursorsAcrossRollback();
System.out.println( true_false +
    ": Can cursors remain open " +
    "across rollbacks?");
true_false =
    dbMetaData.supportsOpenStatementsAcrossCommit();
System.out.println( true_false +
    ": Can statements remain open " +
    "across commits?");
true_false =
    dbMetaData.supportsOpenStatementsAcrossRollback();
System.out.println( true_false +
    ": Can statements remain open " +
    "across rollbacks?");
true_false = dbMetaData.supportsOrderByUnrelated();
System.out.println( true_false +
    ": Can an \"ORDER BY\" clause " +
    "use columns not in the SELECT statement?");
true_false = dbMetaData.supportsOuterJoins();
```

```
System.out.println( true_false +
            ": Is some form of outer " +
            "join supported?");
true_false = dbMetaData.supportsPositionedDelete();
System.out.println( true_false +
             ": Is positioned DELETE supported?");
true_false = dbMetaData.supportsPositionedUpdate();
 System.out.println( true_false +
           ": Is positioned UPDATE supported?");
 true_false =
       dbMetaData.supportsSchemasInDataManipulation();
System.out.println( true_false +
              ": Can a schema name be used in " +
             " a data manipulation statement?");
 true_false =
       dbMetaData.supportsSchemasInIndexDefinitions();
System.out.println( true_false +
              ": Can a schema name be used in " +
              " an index definition statement?");
 true_false =
       dbMetaData.supportsSchemasInPrivilegeDefinitions();
System.out.println( true_false +
     ": Can a schema name be used in " +
       "a privilege definition statement?");
true_false =
```

```
        dbMetaData.supportsSchemasInProcedureCalls();
    System.out.println( true_false +
            ": Can a schema name be used in " +
              " a procedure call statement?");
    true_false =
        dbMetaData.supportsSchemasInTableDefinitions();
    System.out.println( true_false +
    ": Can a schema name be used in " +
      "a table definition statement?");
    true_false = dbMetaData.supportsSelectForUpdate();
    System.out.println( true_false +
           ": Is SELECT for UPDATE " +
             " supported?");
    true_false = dbMetaData.supportsStoredProcedures();
    System.out.println( true_false +
           ": Are stored procedure calls " +
             "using the stored procedure escape " +
              "syntax supported?");
    true_false =
        dbMetaData.supportsSubqueriesInComparisons();
    System.out.println( true_false +
              ": Are subqueries in comparison " +
              " expressions supported? A JDBC " +
               "Compliant driver always returns true.");
    true_false = dbMetaData.supportsSubqueriesInExists();
```

TEAMFLY

Team-Fly®

```
System.out.println( true_false +
        ": Are subqueries in 'exists' " +
        " expressions supported? A JDBC " +
         "Compliant driver always returns true.");
true_false = dbMetaData.supportsSubqueriesInIns();
System.out.println( true_false +
       ": Are subqueries in 'in' " +
        "statements supported? A JDBC " +
        "Compliant driver always returns true.");
true_false =
  dbMetaData.supportsSubqueriesInQuantifieds();
System.out.println( true_false +
     ":  Are subqueries in quantified " +
     "  expressions supported? A JDBC " +
      "Compliant driver always returns true.");
true_false =
     dbMetaData.supportsTableCorrelationNames();
System.out.println( true_false +
     ":  Are table correlation names " +
   " supported? A JDBC Compliant " +
    "driver always returns true.");
true_false = dbMetaData.supportsTransactions();
System.out.println( true_false +
      ": Are transactions supported? " +
      "If not, invoking the method " +
```

```
            "commit is a noop and the isolation level is " +
              " TRANSACTION_NONE.");
        true_false = dbMetaData.supportsUnion();
        System.out.println( true_false +
            ": Is SQL UNION supported?");
        true_false = dbMetaData.supportsUnionAll();
        System.out.println( true_false +
            ": Is SQL UNION ALL supported?");
        true_false = dbMetaData.usesLocalFilePerTable();
        System.out.println( true_false +
              ": Does the database use a file " +
                          "for each table?");
        true_false = dbMetaData.usesLocalFiles();
        System.out.println( true_false +
                ": Does the database store " +
                "tables in a local file?");


//need cursor type
        true_false = dbMetaData.updatesAreDetected(cursor_type);
        System.out.println( true_false +
              ": Indicates whether or not a visible " +
             "row update can be detected by calling " +
              "the method ResultSet.rowUpdated.");
         true_false =
              dbMetaData.supportsTransactionIsolationLevel(
```

```
            cursor_type);
        System.out.println( true_false +
            ": Does this database support " +
            "the given transaction isolation level?");
        true_false = dbMetaData.supportsResultSetType(
            cursor_type);
        System.out.println( true_false +
            ": Does the database support the " +
            " given result set type?");
        true_false =
            dbMetaData.insertsAreDetected(cursor_type);
        System.out.println( true_false +
            ": Indicates whether or not a " +
            " visible row insert can be detected by " +
            "calling ResultSet.rowInserted().");
        true_false =
            dbMetaData.othersDeletesAreVisible(cursor_type);
        System.out.println( true_false +
            ": Indicates whether deletes " +
            "made by others are visible.");
        true_false =
            dbMetaData.othersInsertsAreVisible(cursor_type);
        System.out.println( true_false +
            ": Indicates whether inserts " +
            "made by others are visible. ");
```

```
        true_false =
            dbMetaData.othersUpdatesAreVisible(cursor_type);
        System.out.println( true_false +
            ": Indicates whether updates " +
            "made by others are visible.");
        true_false =
            dbMetaData.ownDeletesAreVisible(cursor_type);
        System.out.println( true_false +
            ": Indicates whether a result " +
            "set's own deletes are visible.");
        true_false =
            dbMetaData.ownInsertsAreVisible(cursor_type);
        System.out.println( true_false +
            ": Indicates whether a result " +
            "set's own inserts are visible.");
        true_false =
            dbMetaData.ownUpdatesAreVisible(cursor_type);
        System.out.println( true_false +
            ": Indicates whether a result " +
            "set's own updates are visible.");


//type codes from the class java.sql.Types
//take arguments in the form method(int fromType, int toType)
int INTEGER = 4;
int FLOAT = 6;
```

```
        true_false = dbMetaData.supportsConvert(INTEGER, FLOAT);
       System.out.println( true_false +
                  ": Is CONVERT between the given " +
                     "SQL types supported?");


//concurrency type defined in java.sql.ResultSet
//method(int fromType, int concurrency)
int CONCUR_READ_ONLY = 1007; //MS Access=false
int CONCUR_UPDATABLE = 1008; //MS Access=false
        true_false =
            dbMetaData.supportsResultSetConcurrency(INTEGER,
            CONCUR_UPDATABLE);
           System.out.println( true_false +
              ": Does the database support the " +
               " concurrency type in " +
                  "combination with the given result set type?");
      } catch (Exception e)
       {
            e.printStackTrace();
       }
    }
}
```

Using the DriverManager and Driver Classes

Listing 7.3 shows a program that demonstrates how to use the **DriverManager** class and the **Driver** class together. The **DriverManager** tries to decide what **Driver** class to use for a particular database. Users can add to the list of drivers without changing the program. This is good practice because you'd like to be able to change your database code quickly and easily with minimal disturbance to the rest of

your program. The program shown in Listing 7.3 is a good skeleton that you can copy and paste to start building your application's connection objects.

Listing 7.3: Using the DriverManager and Driver classes.

```
import java.sql.*;

import java.util.*;


public class DriverManagerDemonstration

{

   public static void main (String args[])

    {

       if (args.length != 1)

        {

            System.out.println("Please provide a data set name.");

          System.exit(1);

        }

        String dataSource = args[0];

        String username = " ";

        String password = " ";

        String url = "jdbc:odbc:" + dataSource.trim();


       Properties credentials = new Properties();

        credentials.put("user", username );

        credentials.put("password", password );


       DriverManagerDemonstration DMD =

              new DriverManagerDemonstration();
```

```
 try
 {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (Exception e)
 {
     System.out.println("JDBC/ODBC driver failure.");
     return;
 }

 try
{   // get the connection
     //OR getConnection(String url)
    //OR getConnection(url, user, password),
  // but internally does this:
     Connection conn = DriverManager.getConnection(
        url, credentials);

     DMD.drivermanager(url);
     System.exit(0);

     conn.close();
} catch (Exception e)
 {
     e.printStackTrace();
```

```
        }
    }
    void drivermanager(String url)
    {
        int loginTimeout, maxTimeout = 100; //seconds
         Driver driver;


          try
          {
//get timeout
              loginTimeout = DriverManager.getLoginTimeout();
//set timeout
               DriverManager.setLoginTimeout(maxTimeout);
              String timeoutMessage = loginTimeout > maxTimeout ?
               loginTimeout + " seconds is gracious":
                    ""+loginTimeout ;
               System.out.println(timeoutMessage);


//list all drivers
             Enumeration driverList = DriverManager.getDrivers();
              while (driverList.hasMoreElements())
              {
                  driver =(Driver)driverList.nextElement();
                  driverReport(driver, url);
              }
```

```
        }
        catch (Exception e)
        {
          System.out.println("Unexpected exception " + e);
        }
      }

    void driverReport(Driver driver, String url)
     {
         int version;
         boolean true_false;
        Properties p = new Properties();
         DriverPropertyInfo[] driverProperty;

          try
          {
            true_false = driver.jdbcCompliant(); //JDBC driver?
            System.out.println("jdbcCompliant()=" + true_false );
//can connect to this DB?
            true_false = driver.acceptsURL(url);
           System.out.println("jacceptsURL()=" + true_false );
//driver's major version number
            version = driver.getMajorVersion();
            System.out.println("getMajorVersion()=" + version );
```

```
// driver's minor version number
            version = driver.getMinorVersion();
            System.out.println("getMinorVersion()=" + version );

/* java.sql.SQLException:
    [Microsoft][ODBC Driver Manager] Driver
  does not support this function (but if it did or yours does):
   driverProperty = driver.getPropertyInfo(url, p);
   for(int index=0; index<driverProperty.length; index++)
    {
        System.out.println("name" + driverProperty[index].name);
        System.out.println("description" +
             driverProperty[index].description);
        System.out.println("required" +
               driverProperty[index].required);
        System.out.println("value" + driverProperty[index].value);
        String[] valueChoices = driverProperty[index].choices;

        for(int i=0; i<valueChoices.length; i++)
        {
            System.out.print(valueChoices[i] + ", ");
        }
  }
           */
```

TEAMFLY

Team-Fly®

```
        } catch(SQLException e) //108
         {
              System.out.println(e);
        }
    }
}
//returns:
//0
//jdbcCompliant()=true
//jacceptsURL()=true
//getMajorVersion()=2
//getMinorVersion()=1
```

The **DriverManager** object manages the JDBC drivers available for your application. This object tries to make life easier for you. For example, during initialization, the **DriverManager** object attempts to load the driver classes referenced in the **jdbc.drivers** system property (multiple drivers are separated by a colon). This allows a user to customize the JDBC drivers used by your applications at runtime (no recompile is necessary). Without direction, Java tries to find an appropriate driver when you call the **getConnection** method.

Note ODBC drivers are notorious for being finicky. Often, the drivers do not come from the database vendors themselves but come from third parties (such as Microsoft). If you suspect that your driver is misbehaving, try updating it from the driver vendor or see if the database vendor has a version that is newer than your driver vendor's version.

Working with Dates

Although Java has several date formats available, the database library uses a unique format. That means that databases don't understand dates formatted by **java.util.Date**, so you have to use **java.sql.Date** instead. The **java.sql.Date** class is simply a SQL date formatter (it extends **java.util.Date**) used so that JDBC can work with real SQL **DATE** values. The **java.sql.Time** class is a convenience, but Sun has deprecated most of its methods. The following methods of this class should be avoided: **getDate**, **getDay**, **getMonth**, **getYear**, **setDate**, **setMonth**, **setTime**, and **setYear**. Some databases support these functions, and others don't. You don't want to rely on methods that might not work at runtime.

Note Even if **getDate** works, it returns the date of the machine the database is on, not the date of the machine the application is on. This is also true of dates that might originate from a stored procedure. Be careful comparing dates when some dates might originate from the server and others might originate from the application.

The program shown in Listing 7.4 demonstrates how to use dates in **java.sql.Date**.

Listing 7.4: Using the java.sql.Date object.

```
import java.sql.*;

import java.util.*;


public class SQLDateDemonstration

{

   public static void main (String args[])

    {

       long startTime = System.currentTimeMillis();

        java.sql.Timestamp time =

              new java.sql.Timestamp(startTime);

       System.out.println( time.toString() );

        //yyyy-mm-dd hh:mm:ss.fffffffff format


       //compare to Calendar:

       AnotherDate anotherDate = new AnotherDate();

        anotherDate.setTimesDates();

        anotherDate.printCalendarDate();

        anotherDate.prettyPrintCalendarDate(startTime);
```

```
    }
}
class AnotherDate
{
    private int year;
    private int month;
    private int day;
    private int hour;
    private int minute;
    private int second;

/* should do it this way
 * must synch calendar with system
 * little wacky about offsets
 * private Calendar cal;
 * int year = cal.YEAR + 2000;
 * int month = cal.MONTH ;
 * int day = cal.DAY_OF_WEEK_IN_MONTH ;
 * int hour = cal.HOUR_OF_DAY;
 * int minute = cal.MINUTE;
 * int second = cal.SECOND;
 */

/* Doesn't work
 * void setTimesDates()
```

```
* {
*    java.sql.Time times =
*       new java.sql.Time(System.currentTimeMillis());
*    java.sql.Date dates =
*       new java.sql.Date(System.currentTimeMillis());
*     year = times.getYear();
*    month = times.getMonth();
*    day = times.getDay();
*    hour = dates.getHours();
*    minute = dates.getMinutes();
*    second = dates.getSeconds();
* }
*/

   void setTimesDates()
   {
      year = 2002;
      month = 10;
      day = 9;
      hour = 2;
      minute = 34;
      second = 22;
   }

   void printCalendarDate()
```

```
{
  String timestamp = year + "-" + month +
                     "-" + day + " " + hour + ":" +
                   minute + ":" + second;

  System.out.println( timestamp );
  //System.out.println( Calendar.toString());
}

void prettyPrintCalendarDate(long startTime)
{//idea swiped from java.sql.Time
    char digit[] =
        "2000-00-00 00:00:00.0       ".toCharArray();
    digit[0] = Character.forDigit(year/1000,10);
    digit[1] = Character.forDigit((year/100)%10,10);
    digit[2] = Character.forDigit((year/10)%10,10);
    digit[3] = Character.forDigit(year%10,10);
    digit[5] = Character.forDigit(month/10,10);
    digit[6] = Character.forDigit(month%10,10);
    digit[8] = Character.forDigit(day/10,10);
    digit[9] = Character.forDigit(day%10,10);
    digit[11] = Character.forDigit(hour/10,10);
    digit[12] = Character.forDigit(hour%10,10);
     digit[14] = Character.forDigit(minute/10,10);
    digit[15] = Character.forDigit(minute%10,10);
```

```
        digit[17] = Character.forDigit(second/10,10);
        digit[18] = Character.forDigit(second%10,10);
        int value = (int)startTime ;
        int divisor = 100000000;
       for (int pos = 20; value > 0 && pos < 29; pos++)
         {
             digit[pos] = Character.forDigit(value/divisor,10);
              value %= divisor;
             divisor /= 10;
        }

      System.out.println( new String(digit).trim() );
   }
}
//returns:
2001-09-17 12:20:05.033
2002-10-9 2:34:22
2002-10-09 02:34:22.027025065
```

It is strange, but the approach taken within **java.sql.Date** to get the time components involves calling deprecated methods in **java.util. Date** (such as **getHour**). The methods within **java.util.Date**, in turn, are calling the same methods in the **Calendar** class; this is the preferred technique. This practice ensures that older programs still run with Java 1.4 (or that this program works with older versions).

Creating and Executing a SQL Query

Once you establish a connection with the database, you'll usually want to execute queries and work with the results. The program in Listing 7.5 is simple, but it performs all the steps required by a typical database program: It connects to the database, executes a query, and processes the result set.

Listing 7.5: Executing a database query.

```
import java.sql.*;


public class ResultSetDemo
{
   public static void main (String args[])
    {
        String source = "jdbc:odbc:northwind",
               username="  ",password="  ";


        try
        {
           Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       } catch (Exception e)
        {
          System.out.println("JDBC/ODBC driver failure.");
           return;
        }


        try
        {
           Connection conn = DriverManager.getConnection(source,
               username, password);
           System.out.println("Connected to " + source);
```

```
            String sql = "select * from employees";
             query(sql, conn);
              conn.close();
        } catch (Exception e)
         {
              e.printStackTrace();
         }
    }

public static void query(String sql, Connection conn)
    throws Exception
 {

         try
         {
            Statement stmt = conn.createStatement();
           ResultSet results = stmt.executeQuery(sql);
            if (!results.next())
            {
                 throw new Exception("No results for: " + sql);
            }

           System.out.println("results from: " + sql);
           while ( results.next() )
            {
```

```
            String lastName = results.getString("LastName");
              System.out.println(lastName);
           }

           stmt.close();
        }
       catch (SQLException e)
       {
           throw new Exception("SQL:" + sql);
       }
   }
}
//returns:
//Connected to jdbc:odbc:northwind
//results from: select * from employees
//Fuller
//Leverling
//Peacock
//Buchanan
//Suyama
//King
//Callahan
//Dodsworth
```

The next section will further discuss result sets. For now, concentrate on the formation of the query. The previous program uses the **connection** object's **createStatement** method to create a **Statement** object. The **executeQuery** method actually accepts the SQL statements and processes them. Notice that the end of the program contains a call to **close** that releases the **Statement** object's resources. This is a good practice because you can be certain to minimize resource use. Sometimes you need to know information about statement execution. You also might want to set information about a statement. For example, you might limit the number of rows a statement can return or set a timeout value. You can learn and set this type of information about a query by using members of the **Statement** object (which physically represents your query and its attributes), as you can see in Listing 7.6.

Listing 7.6: Modifying the Statement object's properties.

```
import java.sql.*;


public class StatementDemo

{

   public static void main (String args[])

    {

        String source = "jdbc:odbc:northwind",

            username="  ",password="  ";


        try

        {

           Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

       } catch (Exception e)

       {

          System.out.println("JDBC/ODBC driver failure.");

           return;

        }
```

TEAMFLY

Team-Fly®

```
        try
        {
         Connection conn = DriverManager.getConnection(source,
                                          username, password);
         System.out.println("Connected to " + source);
        String sql = "select * from employees";
          Statement statement = conn.createStatement();
         int max = statement.getMaxRows();
         System.out.println("max rows: " + max);
         statement.setMaxRows(10); //limit ResultSet rows
          max = statement.getMaxRows();
         System.out.println("max rows: " + max);

        //driver escapes SQL before sending
//error on MS Access
           //statement.setEscapeProcessing(true);
//column max
         //max = statement.getMaxFieldSize();
        //System.out.println("max field size: " + max);
          //max.setMaxFieldSize(0);    //set column max
//driver limit in seconds
        //max = statement.getQueryTimeout();
       //System.out.println("query timeout: " + max);
         //statement.setQueryTimeout(10);          //set timeout
//driver limit in seconds
```

```
        //max = statement.getQueryTimeout();
        //System.out.println("query timeout: " + max);
        SQLWarning warning = statement.getWarnings();
        String warn = warning==null?
            "no warnings":warning.toString();
        System.out.println(warn);
        statement.clearWarnings();          // clear warnings
        statement.close();

        conn.close();
    } catch (Exception e)
    {
        e.printStackTrace();
    }
  }
}
//returns:
//Connected to jdbc:odbc:northwind
//max rows: 0
//max rows: 10
//no warnings
```

Because Microsoft Access doesn't implement some of the features that JDBC supports, we commented the offending statements out of Listing 7.6. If you are using another database, you can try running these lines and see if they work.

Listing 7.7 shows a more complete example. Here, the program uses the query's result set to populate a Java object.

Listing 7.7: Creating table objects in Java.

```
import java.sql.*;


public class QueryToObject
{
   public static void main (String args[])
    {
        String source = "jdbc:odbc:northwind",
            username=" ",password=" ";
       ResultSet result = null;
       Connection conn = null;
       Statement statement = null;
       int maxRows = 5;
       Object[] employeeList = new Object[maxRows];


         try
         {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       } catch (Exception e)
         {
            System.out.println("JDBC/ODBC driver failure.");
            return;
         }
```

```
        try
        {
            conn = DriverManager.getConnection(source,
                username, password);
           System.out.println("Connected to " + source);
           String sql = "select * from employees";
            statement = conn.createStatement();
            result = statement.executeQuery(sql);

           //Option: add a constructor to Employee that
           //populates the attributes after passing ResultSet.
            for(int row = 0; result.next() && row < maxRows; row++)
             {
                Employee employee = new Employee();
//optional
                 //System.out.print( result.getInt("EmployeeID") );
                 employee.setEmployeeID(result.getInt("EmployeeID"));
                 employee.setLastName(result.getString("LastName"));
                 employee.setHireDate(result.getDate("HireDate"));
               //additional data types Java ResultSet can handle
                //Array = getArray(String colName)
                 //InputStream = getAsciiStream(String columnName)
                  //BigDecimal = getBigDecimal(String columnName)
                //InputStream = getBinaryStream(int columnIndex)
```

```
                //Blob = getBlob(String colName)
                //boolean = getBoolean(String columnName)
                 //byte = getByte(String columnName)
                //Reader = getCharacterStream(String columnName)
                //Clob = getClob(String colName)
                 //double = getDouble(String columnName)
                 //float = getFloat(String columnName)
                 //long = getLong(String columnName)
                //Object = getObject(String columnName)
                //short = getShort(String columnName)
                //Time = getTime(String columnName)
                //Timestamp = getTimestamp(String columnName)
                 //URL = getURL(String columnName)

                 employee.setDelimiter(", ");
//optional
                 //System.out.println( employee.toString() );

                 employeeList[row] = employee;
              }

            DisplayEmployee showEmployee =
               new DisplayEmployee(employeeList);
             showEmployee.print();
        } catch (Exception e)
```

```
        {
            e.printStackTrace();
        }
        finally
        {
            try
            {
                if(result != null) { result.close(); }
                if(statement != null) { statement.close(); }
                if(conn != null) { conn.close(); }
            } catch(SQLException sqlE)
            {
            }
        }
    }
}
class DisplayEmployee
{
    private Object[] employeeList;

    public DisplayEmployee(Object[] employeeList)
    {
        this.employeeList = employeeList;
    }
```

```
    void print()
    {
        int length = this.employeeList.length;

       for(int index = 0; index < length ; index++)
        {
            String output = this.employeeList[index].toString();
             System.out.print( output );
        }
    }
}
class Employee
{
   private int EmployeeID = 0;
    private String LastName;
    private String FirstName;
    private String Title;
    private String TitleOfCourtesy;
   private Date BirthDate;
   private Date HireDate;
    private String Address;
    private String City;
    private String Region;
    private String PostalCode;
    private String Country;
```

```
private String HomePhone;
private String Extension;
private String Photo;
private String Notes;
private String ReportsTo;
private String Delimiter;

public String getDelimiter()
{
    return this.Delimiter;
}
public void setDelimiter(String delimiter)
{
    this.Delimiter=delimiter;
}
public int getEmployeeID()
{
    return this.EmployeeID;
}
public void setEmployeeID(int EmployeeID)
{
    this.EmployeeID=EmployeeID;
}
public String getLastName()
{
```

```
        return this.LastName;
    }
    public void setLastName(String LastName)
    {
        this.LastName=LastName;
    }
    public String getFirstName()
    {
        return this.FirstName;
    }
    public void setFirstName(String FirstName)
    {
        this.FirstName=FirstName;
    }
    public Date getHireDate()
    {
        return this.HireDate;
    }
    public void setHireDate(Date HireDate)
    {
        this.HireDate=HireDate;
    }
// Add get & set for other attributes

    public String toString()
```

```
{
    String output = " ", del = this.Delimiter;

  if (this.EmployeeID != 0)
     { output += this.EmployeeID + del;}
  if (this.LastName != null)
     { output += this.LastName + del; }
  if (this.FirstName != null)
     { output += this.FirstName + del; }
  if (this.Title != null)
     { output += this.Title + del; }
  if (this.TitleOfCourtesy != null)
     { output += this.TitleOfCourtesy + del; }
  if (this.BirthDate != null)
     { output += this.BirthDate + del; }
  if (this.HireDate != null)
     { output += this.HireDate + del; }
  if (this.Address != null)
     { output += this.Address + del; }
  if (this.City != null)
     { output += this.City + del; }
  if (this.Region != null)
     { output += this.Region + del; }
  if (this.PostalCode != null)
     { output += this.PostalCode + del; }
```

TEAMFLY

Team-Fly®

```
        if (this.Country != null)
            { output += this.Country + del; }
        if (this.HomePhone != null)
            { output += this.HomePhone + del; }
        if (this.Extension != null)
            { output += this.Extension + del; }
        if (this.P hoto != null)
            { output += this.Photo + del; }
        if (this.Notes != null)
            { output += this.Notes + del; }
        if (this.ReportsTo != null)
            { output += this.ReportsTo + del; }
        if (output != " ")
            { output += "\n"; }

        return output;
    }
}
//returns:
//Connected to jdbc:odbc:northwind
//1, Davolio, 1992-05-01,
//2, Fuller, 1992-08-14,
//3, Leverling, 1992-04-01,
//4, Peacock, 1993-05-03,
//5, Buchanan, 1993-10-17,
```

The previous program raises a question about result persistence. When you want to create objects in Java to manipulate the data within an application, sometimes Java affords you many more options than the database will. You have more control over the data this way. Also, it is usually faster to fetch the data from the object fields (as is done in the **Employee.toString** method) than to work directly with the **ResultSet**.

Warning If you create your own objects that are equivalent to a database table, be very careful about changing the data. What you do to your object will *not* be reflected in the database (unless you write code to modify the database appropriately). The reverse is also true: If you update the database, your objects will not reflect these changes automatically.

In Listing 7.7, notice the highlighted **finally** block. This will always execute after the associated **try** block. The **finally** block will execute if there is no exception thrown or if an exception is processed. This block is an ideal place for closing database objects because it will always execute, even under unusual conditions.

Notice that there are many **get** methods, one for each data type. The method you call must match the data type in the database. So, for example, **getString** retrieves a **String** object, and **getInt** returns an integer. Unfortunately, the mapping of database types to Java types is driver dependent.

Related solution:	Found on page:
Defining Blocks	66

Interrogating a ResultSet

In addition to the data in a **ResultSet**, you frequently need to know details about the results (for example, the number of rows in the set or the names of each column). Listing 7.8 shows you how to use the **ResultSetMetaData** object to learn about a result set.

Listing 7.8: Interrogating a result set.

```
import java.sql.*;


public class ResultSetMetadata

{

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

```
{
    String source = "jdbc:odbc:northwind",
        username=" ",password=" ";
   ResultSet result = null;
   Connection conn = null;
   Statement statement = null;

    try
    {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   } catch (Exception e)
    {
     System.out.println("JDBC/ODBC driver failure.");
      return;
    }

    try
    {
       conn = DriverManager.getConnection(source,
           username, password);
      System.out.println("Connected to " + source);
     String sql = "select * from employees";
      statement = conn.createStatement();
      result = statement.executeQuery(sql);
```

```
            ResultsInfo resultsinfo = new ResultsInfo(result);
            resultsinfo.print();
        } catch (Exception e)
        {
            e.printStackTrace();
        }
        finally
        {
            try
            {
              if(result != null) { result.close(); }
                if(statement != null) { statement.close(); }
              if(conn != null) { conn.close(); }
            } catch(SQLException sqlE)
            {
            }
        }
    }
}

class ResultsInfo
{
    private ResultSet resultset;

    public ResultsInfo(ResultSet resultset)
```

```
    {
        this.resultset = resultset;
    }

    void print()
    {
        ResultSetMetaData columnInfo;

        try {
            boolean true_false = this.resultset.next();
            columnInfo = this.resultset.getMetaData();

        for (int col = 1; col <= columnInfo.getColumnCount();
                            col++) {
            String tableName = columnInfo.getTableName(col);
            System.out.println("getTableName()=" + tableName);
            String catalogName = columnInfo.getCatalogName(col);
             System.out.println("getCatalogName()=" + catalogName);
            String schemaName = columnInfo.getSchemaName(col);
            System.out.println("getSchemaName()=" + schemaName);
            String columnName = columnInfo.getColumnName(col);
             System.out.println("getColumnName()=" + columnName);
            String columnLabel = columnInfo.getColumnLabel(col);
            System.out.println("getColumnLabel()=" + columnLabel);
            int columnType = columnInfo.getColumnType(col);
```

```
            System.out.println("getColum nType()=" + columnType);


            /* Optional:
             *
             *   switch (columnType)
             *   {
             *       case Types.DOUBLE:
             *           do? columnInfo.getDouble(i);
             *           break;
             *       case Types.FLOAT:
             *           do? columnInfo.getDouble(i));
             *           break;
               *       case Types.INTEGER:
               *           do? columnInfo.getInt(i));
               *           break;
               *       //ARRAY,BIGINT,BINARY,BIT,BLOB,BOOLEAN
               *       //CHAR,CLOB,DATALINK,DATE,DECIMAL,DISTINCT
               *       //DOUBLE,FLOAT,INTEGER,JAVA_OBJECT
               *       //LONGVARBINARY,LONGVARCHAR,NULL
               *       //NULL,NUMERIC,OTHER,REAL,REF
               *       //SMALLINT,STRUCT,TIME,TIMESTAMP
               *       //TINYINT,VARBINARY,VARCHAR
             *   }
             *
             */
```

```
            String columnTypeName =
                        columnInfo.getColumnTypeName(col);
            System.out.println("getColumnTypeName()=" +
                        columnTypeName);
            int columnDisplaySize =
                        columnInfo.getColumnDisplaySize(col);
            System.out.println("getColumnDisplaySize()=" +
                        columnDisplaySize);
            int precision = columnInfo.getPrecision(col);
            System.out.println("getPrecision()=" + precision);
            int scale = columnInfo.getScale(col);
            System.out.println("getScale()=" + scale);
            boolean autoIncrement =
                        columnInfo.isAutoIncrement(col);
            System.out.println("isAutoIncrement()=" +
                        autoIncrement);
            boolean caseSensitive =
                        columnInfo.isCaseSensitive(col);
            System.out.println("isCaseSensitive()=" +
                        caseSensitive);
            boolean isMoney = columnInfo.isCurrency(col);
            System.out.println("isCurrency()=" + isMoney);
            boolean surelyWritable =
                        columnInfo.isDefinitelyWritable(col);
```

```
                System.out.println("isDefinitelyWritable()=" +
                        surelyWritable);
            int nullType = columnInfo.isNullable(col);
            final int columnNoNulls = 0;
            final int columnNullable = 1;
            final int columnNullableUnknown = 2;
            switch(nullType) {
                case columnNoNulls:
                    System.out.println("isNullable()=NoNulls");
                    break;
                case columnNullable:
                    System.out.println("isNullable()=Nullable");
                    break;
                case columnNullableUnknown:
                    System.out.println("isNullable()=
                            NullableUnknown");
                    break;
            }
            boolean readOnly = columnInfo.isReadOnly(col);
            System.out.println("isReadOnly()=" + readOnly);
            boolean searchable = columnInfo.isSearchable(col);
            System.out.println("isSearchable()=" + searchable);
            boolean signed = columnInfo.isSigned(col);
            System.out.println("isSigned()=" + signed);
            boolean writable = columnInfo.isWritable(col);
```

```
System.out.println("isWritable()=" + writable);
            }
            this.resultset.close();

        } catch(SQLException e) {
            System.out.println(e);
        }
    }
}
//returns:
//getTableName()=employees
//getCatalogName()=
//getSchemaName()=
//getColumnName()=ReportsTo
//getColumnLabel()=ReportsTo
//getColumnType()=4
//getColumnTypeName()=INTEGER
//getColumnDisplaySize()=11
//getPrecision()=10
//getScale()=0
//isAutoIncrement()=false
//isCaseSensitive()=false
//isCurrency()=false
//isDefinitelyWritable()=true
//isNullable()=Nullable
```

```
//isReadOnly()=false

//isSearchable()=true

//isSigned()=true

//isWritable()=false

//this is repeated for every column
```

Updating Database Data

Java provides many ways to update data on a table. The **ResultSet** object is the recommended way to change data. The easiest way to do this is with a SQL **UPDATE** statement, like this:

```
update employees
   set employeeID = EmployeeID + 3088
   where EmployeeID = 4
```

This SQL code will change records that have an **EmployeeID** of 4 so that they have an ID of 3092. In Java, you can pass this SQL to the **executeUpdate** method of the **Statement** object. Here's how you can execute this update:

```
String sql = "update employees "
sql += "set employeeID = EmployeeID + 3088 "
sql += "where EmployeeID = 4 "
int rowsAffected = statement.executeUpdate(sql);
```

In addition to **SELECT** statements, you can also execute **INSERT**, **UPDATE**, or **DELETE** statements through the **executeUpdate()** method. Here's another way to do an update:

```
resultSet.absolute(298);// Row to change
resultSet.updateInt("EmployeeID", 392);
resultSet.updateRow();//commit it
```

The previous code moves the **ResultSet** to record 298 (which happens to be the one we want to change). The program updates the **EmployeeID** to 392. You can also insert rows with new data by using a similar technique. An updatable **ResultSet** object has a set of methods that, together, allow you to insert a row into the database. The code looks like the following:

```
resultSet.moveToInsertRow(); // creates a temporary row
resultSet.updateString("LastName", "Cox");
resultSet.updateString("FirstName", "Perry");
resultSet.updateInt("EmployeeID",3093);
resultSet.insertRow();//commits new row
```


Team-Fly®

Using Prepared Statements

The **PreparedStatement** object stores statements before you execute them. This can improve efficiency if you will execute the same sequence of statements more than once. The **PreparedStatement** object is Java's equivalent to a database stored procedure. Both are precompiled, which means they don't have to be parsed after their first execution. Like a stored procedure, a **PreparedStatement** acts like a method that needs only a few parameters (the only part that changes between executions). You use question-mark placeholders as the parameter list (which can include output parameters). You identify each parameter by its count. Therefore, the first question mark is 1, the second is 2, and so on. Once you give the **PreparedStatement** object the query, you provide the values to replace the question marks, like this:

```
String sql = "update employees "
sql += "set employeID = ?, "//parameter 1
sql += "FirstName = ?, "//parameter 2
sql += "LastName = ? "//parameter 3
sql += "where employeeID = ? "//parameter 4

PreparedStatement updateEmployee = conn.prepareStatement(sql);
updateEmployee.setInt(1, ID + 34);//1 NOT 0 based
updateEmployee.setString(2, newFirstName);
updateEmployee.setString(3, newLastName);
int rowsUpdated = updateEmployee.executeUpdate()://commit update
```

Tip Use the **PreparedStatement** object to manage statements that will be used repeatedly.

Note If you turn off the autocommit feature (**connection.setAutoCommit(false)**), then you must explicitly commit the transaction (**connection.commit()**).

Using Stored Procedures

A stored procedure is a precompiled collection of SQL statements that run as a unit. Stored procedures reside in the database server and are the counterpart to Java prepared statements. Stored procedures execute very quickly on the database, especially in client/server environments. The following code shows you how to look up an employee from a stored procedure rather than by simply executing a SQL statement:

```
//build create procedure string
String procedure = "Create PROCEDURE sp_getEmployee " +
"(" +
"    @employee_id int " +
")" +
```

```
"AS " +
"BEGIN" +
"   SELECT * from employee where EmployeeID = @employee_id " + "END" +

//execu te create procedure statement on DB
Statement statement = conn.createStatement();
statement.executeUpdate(procedure);

//execute stored procedure with one parameter
CallableStatement executeSP =
      conn.prepareCall("{Call  sp_getEmployee(?)}");
executeSP.setInt(1, 83);
ResultSet results = executeSP.executeQuery();

//loop through result set
while (results.next())
{
  System.out.println(rs.getInt(1));// EmployeeID
  System.out.println(rs.getString(2));// FirstName
  System.out.println(rs.getString(3));// LastName
}
```

The **CallableStatement** class is actually an interface that refers to some other object (which is driver dependent). The object represents a stored procedure, as you could probably guess. This code creates the stored procedure using **executeUpdate**. Of course, you can omit this initial part after you create the procedure once or if the procedure already exists.

Some stored procedures have output parameters. Consider this stored procedure:

```
create procedure newEmployeeID @oldID int, @newID int output
as
begin
  if @oldID = 0
       begin
          return  0
       end
   else
```

```
        begin
          select EmployeeID+3000 from Employees
          where EmployeeID = @OldID
           return 1
        end
end
```

You can use the **registerOutParameter** method to mark certain parameters as output parameters and then retrieve them by using the usual **get** methods. Here's an example:

```
CallableStatement executeSP =
     conn.prepareCall("{? = Call newEmployeeID(?, ?)}" );
executeSP.registerOutParameter(1, Types.INTEGER);
executeSP.setInt(2, 5);
executeSP.registerOutParameter(3, Types.INTEGER);
executeSP.execute();
System.out.println("Return Successful? ="+ executeSP.getInt(1));
System.out.println("New EmployeeID ="+ executeSP.getInt(3));
```

Chapter 8: The Internet and Networking

In Brief

Java is one of the first widely used programming languages built to be Internet-aware. Java makes it simple—almost too simple—to write network programs. However, there is one catch: Java recognizes certain network protocols. If you want to work with one of thes e protocols, everything is simple. If you want to work with something more exotic, however, you'll have to do a great deal of work. You might even have to resort to nonportable code. Fortunately, Java does handle the most common protocols, and that makes most network programming very simple.

Java does a great job of hiding low-level networking details from you, but sometimes you really need to know what is going on at the low level of a network transaction. Most of these details appear in various Requests for Comments (RFCs) maintained by the groups that are responsible for the Internet's operation. Many of these RFCs are actually standards that Internet hosts must follow. You can find a list of the commonly used RFCs in Appendix D.

Essential Socket Programming

Traditional socket programs use C. However, Java offers many high-level ways to handle sockets, and these ways make writing network programs much easier. The downside is that it is very difficult to circumvent this built-in support. For example, Java sockets support User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) connections. If you want something else—for example, Internet Control Message Protocol (ICMP) for a ping program—you'll have to resort to native method calls (probably written in C).

Java's network support—not surprisingly—is in the **java.net** package. Many of the classes in this package aren't meant for ordinary use. You'll likely use the following classes:

- **DatagramPacket**—A packet of data sent (or received) via a UDP socket (implemented by **DatagramSocket**).
- **DatagramSocket**—A socket that communicates via UDP.
- **HttpURLConnection**—A class used to communicate specifically with HTTP servers.
- **InetAddress**—A representation of an IP address by name or number.
- **JarURLConnection**—A class used to work with JAR files from a local file, a Web server, or an FTP server.
- **MulticastSocket**—A socket designed for multicasting (i.e., sending data to and receiving data from more than one remote socket).
- **ServerSocket**—A socket that listens for connections from clients.
- **Socket**—A general-purpose socket.
- **URL**—A class that represents a URL address.

- **URLDecoder**—A class that decodes data formatted as a URL.
- **URLEncoder**—A class that encodes URL data.

We'll address some of these classes in later chapters. The primary classes of interest in this chapter deal directly with sockets.

The basic idea behind socket communications is simple. A client establishes a connection with a server. Once the connection is made, the client can write to the socket to send data to the server. Conversely, the server sends data to the socket that the client will read. The details can be complex, but the idea is just that simple.

Java provides three main types of socket classes. **DatagramSocket** is the class that implements the UDP protocol. UDP sockets don't use connections, don't ensure data delivery, and don't preserve the data's sequence. Data in and out of the socket resides in a **DatagramPacket** object.

The other two socket classes are **Socket** and **ServerSocket**, and they both support TCP connections. If you are connecting to a server, you'll use **Socket**. If you are writing a server, you'll use **ServerSocket**. Why the difference? A client socket doesn't really care what port it uses locally. The client socket does need to connect to a specific port on another computer. On the other hand, a server is very concerned with its local port assignment (that's how clients find it). Servers also have to listen for incoming connections.

Listening for an incoming connection isn't an intuitive process. Suppose you are a Web server listening on port 80. When a client connects to you, it makes sense to think that you'd be using port 80 to talk to the client, right? That's not how it works, however. If it did work this way, only one client would be able to connect at a time. Internally, the networking software arranges it so that when a client connects on a port, the request goes to another socket that has a randomly assigned port. The client doesn't really care as long as it connects to the server, and the server's main socket is free to continue listening for incoming connections.

Addressing

No matter what kind of socket you plan to use, you'll need a way to specify the address of the socket. You might think you could just pass a hostname or an IP address to the socket's constructor, but that's not quite the case. Instead, you'll use **InetAddress** to represent the remote computer's address. **InetAddress** doesn't have any public constructors. So how do you get an instance of the object? You can use one of three static methods to create a new instance for you:

- The **getLocalHost** method returns an **InetAddress** object that refers to your local computer.
- The **getByName** method returns an object for the specified host. The name can be a string that represents the IP address, or it can be the actual hostname.
- The **getAllByName** method finds all addresses that match a specified name. The name might be a case-insensitive machine name (http://www.coriolis.com) or a string that contains an IP address (if you specify an IP address, then the method

will return that IP address only after checking to see that it is a valid address). This method returns an array.

Making any of these calls will either return an **InetAddress** object (or objects, in the case of **getAllByName**) or throw an **UnknownHostException** if the name is not resolvable. Usually, you'll just pass the **InetAddress** object to a socket constructor. However, you can also use the object as a way to resolve hostnames to IP addresses (sort of an interface to DNS). You can call the instance methods **getHostName** and **getHostAddress** to return the hostname and IP address. You can also use **getAddress** to return the IP address as a byte array, not as a string.
Listing 8.1 shows a simple console program that can resolve a name or IP address. When you pass an IP address on the command line, the program will work, but it might or might not look up the corresponding name. If Java can't resolve the hostname, the socket's name is simply a string that represents its IP address.

Listing 8.1: A simple console program that can resolve a name or IP address.

```
import java.net.*;



public class GetIP
  {
 public static void main(String [] args) {
    InetAddress address=null;
   if (args.length==0) {
        System.out.println("usage: GetIP host");
         System.exit(1);
         }
    try {
       address=InetAddress.getByName(args[0]);
       }
   catch (UnknownHostException e) {
      System.out.println("I can't find " + args[0]);
       System.exit(2);
```

```
        }
      System.out.println(address.getHostName() + "="
        + address.getHostAddress());
        System.exit(0);
        }
    }
```

Listing 8.2 shows how to use the **getAllByName** method. This method returns an array of all the **InetAddress** objects that apply to the host. Armed with that array, you can iterate through the array and determine the characteristics of all the **InetAddress** objects. For example, you can determine if the address refers to the local computer, or if it is a multicast address.

Listing 8.2: How to use the getAllByName method.

```
import java.net.*;


public class GetAllIP {
  public static void main(String [] args) throws Exception {
    InetAddress[] addr = InetAddress.getAllByName(args[0]);
    for (int i=0;i<addr.length;i++)
        System.out.println(addr[i]);
    }
  }
```

The **InetAddress** class is not very complex, but you can use it when you connect to another machine by using a socket. The constructors also accept hostnames, so you rarely have to use this class; it is useful when you want to resolve addresses yourself, however.

A TCP Client

When you want to connect to a server, you use the **Socket** class. The simplest way to create a **Socket** is to provide a hostname (or **InetAddress** object) and a port number to the constructor. Listing 8.3 shows a simple program that connects to a Web server. You supply the hostname or IP

address on the command line. The program doesn't transfer any data, but it does check to see if some server, presumably a Web server, is listening on port 80, and then the program connects to it.

Listing 8.3: A simple program that connects to a Web server.

```
import java.net.*;

import java.io.*;


public class WebPing {

  public static void main(String[] args) {

    try {

      InetAddress addr;

     Socket sock=new Socket(args[0],80);

      addr=sock.getInetAddress();

      System.out.println("Connected to " +

          addr);

     sock.close();

     }

   catch (java.io.IOException e) {

    System.out.println("Can't connect to " + args[0]);

     System.out.println(e);

     }

  }

}
```

If you are running a Web server on your local machine, you can test this program against the **localhost** computer. The output will look something like this:

```
Connected to localhost/127.0.0.1
```

Notice that the implicit call to **InetAddress.toString** (made by **println**) prints the hostname and the IP address automatically. You can, of course, obtain the hostname and the IP address and format them yourself.

A TCP Server

The most common type of socket you'll use is a TCP socket. When TCP is used, one computer acts as a server, and the other computer acts as a client. You'll use **ServerSocket** to write a server. You construct a **ServerSocket** object by calling the constructor with a port number. If you are writing a standard server, you'll use the well-known port number associated with that server type. For example, a Web server would use p ort 80. If you aren't writing a standard server, you can select a port number that isn't in use on your system (typically higher than 1023).

Try issuing the following command:

telnet localhost 8123

It is very likely that the program will report that it can't connect to that port. If the port is in use, just select another number. Now look at the program in Listing 8.4 . This program provides a server on port 8123. The server doesn't do anything; however, if you run this program, the Telnet program will be able to connect to port 8123.

Listing 8.4: A program that provides a server on port 8123.

```
import java.net.*;

import java.io.*;



public class Techo {

  public static void main(String[] args) {

     try {

       ServerSocket server=new ServerSocket(8123);

       while (true) {

        System.out.println("Listening");

      Socket sock = server.accept();

       InetAddress addr=sock.getInetAddress();

      System.out.println("Connection made to "
```

```
            + addr.getHostName() + " ("
            + addr.getHostAddress() + ")");
          pause(5000);
           sock.close();
          }
        }
    catch (IOException e) {
       System.out.println("Exception detected: " + e);
       }
    }
private static void pause(int ms) {
    try {
         Thread.sleep(ms);
        }
   catch (InterruptedException e) {}
   }
}
```

TEAMFLY

The constructor for **ServerSocket** accepts the port number. You could easily modify the code to accept a port number from a property file or from the command line, if you thought the port number might change. Once you have the server socket, you can call **accept** to listen for incoming connections. This call will block the program's execution, so the program will halt until a client connects. If this is unacceptable, you'll have to make the call from within a thread, a topic covered later in this chapter (see "Creating a Multithreading Server," in the Immediate Solutions section). Because this server doesn't do anything, it just pauses for five seconds when someone connects. If you try to run two copies of the server, the second copy will throw an exception. Only one program can listen to a port at once. If you connect to the host while it is busy, the system will complete the connection when the server calls **accept** again. The limit on how many clients can be waiting for the server varies by system. You can ask for a certain queue size by using a different **ServerSocket** constructor, but the underlying system is not obligated to fulfill your request.

Team-Fly®

If the port number you want is already in use, the constructor will throw an **IOException**. You can use this to discover the ports that are already in use on your machine (see Listing 8.5).

Listing 8.5: A program that can be used to scan your computer for ports in use.

```
import java.net.*;


public class LocalScan {
  public static void main(String [] args) {
    for (int i=1;i<1023;i++) {
        testPort(i);
        }
    System.out.println("Completed");
    }
  private static void testPort(int i) {
    try {
       ServerSocket sock=new ServerSocket(i);
        }
   catch (java.io.IOException e) {
        System.out.println("Port " + i + " in use.");
        }
    }
}
```

Note If you are running under Unix, you'll probably need to be running as root to start servers on reserved port numbers (those lower than 1024).

The **Socket** classes allow Java networking at a relatively low level. Even then, the techniques do not require you to know about arcane details such as Internet Protocol (IP) headers and source routes. However, Java provides an even higher-level interface to networking that you can often—but not always—use to get quick results.

This higher-level interface works great for Web pages. For other protocols, however, it isn't very programmer-friendly, so you might want to continue using the **Socket** object as you have in the earlier examples.

Inside the URL Class

The **URL** class represents a resource in Universal Resource Locator format, such as http://www.coriolis.com. Usually, you'll construct the object with the URL, although a variety of constructors will allow you to specify the URL in pieces instead of in one string, if you prefer. Once you have a **URL** object, you can retrieve the contents of the URL in several ways. For example, you can call **openStream** to fetch an **InputStream** that corresponds to the document. You can also call **getContent**, which returns an object. The object's type is dependent on the document's Multipurpose Internet Mail Extension (MIME) type.

That means you can retrieve a Web page with just a few lines of code (see Listing 8.6). However, you don't get access to the headers, nor can you send data to the server. If you need to access the headers or send data, you need a **URLConnection** object (discussed shortly).

Listing 8.6: A simple way to read a Web page.

```
import java.net.*;

import java.io.*;


public class EZUrl {

   public static void main(String[] args) throws Exception {

       URL url = new URL(args[0]);

      InputStream html = url.openStream();

       int c;

       do {

          c=html.read();

          if (c!=-1) System.out.print((char)c);

         } while (c!=-1);

    }

}
```

Here's another problem with using **URL** (and, more specifically, **getContent**): The classes that represent data are Sun-specific dasses unless you provide your own. These Sun-specific classes are not documented, and you can't be sure that Sun will continue to provide them in future versions. If Sun does make major changes to the classes (it might, because they are not documented), you could find yourself making major program changes to adapt to a new Java version.

Inside the URLConnection Class

Reading a URL with the **URL** object is easy. However, what do you do if you want more control over the Hypertext Transport Protocol (HTTP) transaction? Perhaps you want to pass data to a server-side script or read headers. Then you probably want to use the **URL** object's **openConnection** member. This function returns a **URLConnection** object. If the URL actually uses the HTTP protocol (in the case of a Web page, as opposed to, say, an FTP download), the object returns a subclass of **URLConnection** known as an **HttpURLConnection**. This connection object lets you set headers and requests before submitting the URL request.

For example, suppose you want to submit a request to a form on a Web server. You might use code like that shown in Listing 8.7. This code creates the **URL** object as usual (in this case, it opens InterNIC's WHOIS form). The program casts the **URLConnection** object into a specialized subclass (the **HttpURLConnection** object, covered shortly).

Listing 8.7: Submitting data with URLConnection.

```
//Sample Code to Submit a Form

import java.net.*;

import java.io.*;




class insearch
  {
 static public void main(String [] argv) throws Exception
    {
   URL url=new URL("http://www.internic.net/cgi-bin/whois");
    HttpURLConnection conn=
        (HttpURLConnection)url.openConnection();
```

```
int c;
conn.setDoInput(true);
conn.setDoOutput(true);
conn.setRequestMethod("POST");
conn.setRequestProperty("Content-type",
      "application/x-www-form-urlencoded");
conn.connect();
PrintWriter pout = new
      PrintWriter( new
      OutputStreamWriter(conn.getOutputStream(),
      "8859_1"), true );
pout.print("whois_nic=" + URLEncoder.encode(argv[0]) +
  "&submit=Search&type=domain");
pout.flush();
  // read results
System.out.println(conn.getResponseMessage());
InputStream is=conn.getInputStream();
do
   {
   char x;
    c=is.read();
    x=(char)c;
   if (c!=-1) System.out.print(x);
   } while (c!=-1);
}
```

```
}
```

Before the program establishes a connection, it calls **setDoInput** and **setDoOutput** to signify that it wants to read from and write to the URL. The **setRequestMethod** tells the URL to use the **HTTP Post** method to submit form data. Finally, the **setRequestProperty** sets the request headers to indicate that the server can expect form data.

After the program attends to these details, it connects, using the **connect** method. At this point, the program uses **getOutputStream** to ask the URL for an **OutputStream**. Writing to this stream causes data to flow from the program to the Web server. D ata going to the server should be encoded to prevent special characters from confusing the server, and that's the purpose of the **URLEncoder** object's static **encode** method. It replaces blanks with plus signs and replaces special characters with hexadecimal escape sequences —just the way the Web server wants it.

The **URLConnection** class has many methods, some of which are not very useful. Here are the most important ones:

- **getDefaultAllowUserInteraction**—This static method returns **true** or **false** depending on the system's default value for the internal flag that determines whether **URLConnection** objects can interact with the user. For example, if the connection requires a password, this flag determines if the object can prompt the user for it.
- **setDefaultAllowUserInteraction**—This static method sets the interaction flag for all new instances. The value applies only to subsequently created instances.
- **getAllowUserInteraction**—This method returns the value of the interaction flag for this object. This flag applies only tothis instance, and it originally set the value of the default value previously mentioned.
- **setAllowUserInteraction**—This method sets the interaction flag for this instance of the object. Again, this flag only applies to the instance and not the entire class.
- **getFileNameMap**—You can use this method to retrieve a **FileNameMap** reference. You can use this reference to guess a MIME type from a file name (for example, a file with a name ending in .txt will be text/plain).
- **setFileNameMap**—If you want to set your own **FileNameMap** object, you can use this method.
- **guessContentTypeFromName**—This method guesses a MIME type from a file name. The method calls the default **FileNameMap** object or the one set by **setFileNameMap** to do the work.
- **guessContentTypeFromInputStream**—Instead of using the MIME type, you can ask this method to examine a few bytes of the input stream to try to determine the type of the data.
- **getContent**—This method has two variations. The first one returns an object that varies depending on the document's MIME type. For example, the function returns images as **sun.awt.image.URLImageSource** objects and returns text as a type of **InputStream** . This variation is similar to the **URL.getContent** method mentioned

earlier in the chapter. The second version requires an array of **Class** objects. The method will then attempt to return one of those types, if possible.

- **getContentLength**, **getContentEncoding**, **getContentType**, **getDate**, **getExpiration**, **getLastModified**—These methods all return values from the corresponding headers of the document, if applicable.
- **setIfModifiedSince**—This method sets the **IfModifiedSince** header.
- **getHeaderField, getHeaderFieldDate**, **getHeaderFieldInt**, **getHeaderFieldKey**—You can use these methods to retrieve arbitrary headers, either as a **String** or as a specific data type (for example, **Date** or **int**).
- **setRequestProperty**—You can use this call to set an arbitrary header. You might set a header to authenticate with the server or simulate a form submission, for example.
- **setDoInput**, **setDoOutput**, **getDoInput**, **getDoOutput**—By default, the object will handle only incoming data from the server. However, you can control the data directions by using these methods.
- **connect**—This method contacts the server. Before calling **connect**, you must have any options and headers set.
- **getInputStream**, **getOutputStream**—These methods return streams you can use to communicate with the server. Keep in mind that these streams are subject to the state of the **setDoInput** and **setDoOutput** methods.

As you can see, although **URLConnection** is supposed to be a general-purpose class, it is actually heavily slanted toward HTTP transactions. Special classes known as *content handlers* do the actual work, and you can even create your own, if you like.

URLConnection Subclasses

You can use a subclass of **URLConnection**—**HttpURLConnection**—which is abstract. This is the class used earlier in Listing 8.7. The only way to get this subclass is to create a **URL** object with an **http://www.** URL and then call **openConnection** on the object. You can then cast the returned **URLConnection** to an **HttpURLConnection**.

This special subclass allows you to set the request type (for example, **GET** or **POST**) by using **setRequestMethod**, get the response code by using **getResponseCode**, and perform other HTTP-specific tasks.

In addition to the HTTP-specific subclass, you can use a special subclass for Java Archive (JAR) files, which are the same JAR files you use to package Java classes. You can retrieve JAR files from a local file or via HTTP. To create a JAR connection for a file named inside.htm in the nettest.jar file, you again create a **URL** object. However, in this case, you provide a pseudo-URL, like these three:

- **jar:file://c%3A/lib/nettest.jar!/inside.htm**
- **jar:http://www.coriolis.com/jars/nettest.jar!/inside.htm**
- **jar:ftp://ftp.coriolis.com/jars/nettest.jar!/inside.htm**

In the first example, the JAR file is a local file named nettest.jar in the c:\lib directory. Note that the colon requires encoding, but the slashes do not. In the second and third examples, the JAR file resides on a server at the indicated URL. In all cases, you want the inside.htm file from within the JAR file. The exclamation mark simply ends the pseudo-URL and is not part of the JAR file name. The portion after the exclamation mark is the file name within the JAR file that you want to extract. There is no need to encode the exclamation point because these have no special meaning within a URL. Once you have the **URL** constructed with the special URL, you can read data from the file just as you would any other **URL**. That means you can call **openStream** or extract a **URLConnection** object and cast it to a **JarURLConnection**. Like any other input/output (I/O)—related call, if the file doesn't exist (or if an error occurs) you'll get an **IOException**.

Protocol and Content Handlers

The **URL** and **URLConnection** objects rely on **URLStreamHandler** and **URLConnection** classes to perform protocol-specific processing. In addition, a **ContentHandler** class understands how to convert incoming data into a Java type. However, the official Java library doesn't have implementations for any of these classes. The classes you use when you use common protocols, such as HTTP or FTP, are actually classes in the **sun** package and are not part of the Java baseline.

Custom Protocols

Under the hood, the **URL** object examines the protocol portion of its URL and calls an object that implements **URLStreamHandler-Factory** (you set this object with the static **URL.setURLStream-HandlerFactory** method). This object is responsible for creating a **URLStreamHandler** subclass that corresponds to the specified protocol. However, you can install only one **URLStreamHandler-Factory**—once it is set, you can't change it.

The object that subclasses **URLStreamHandler** creates a corresponding **URLConnection** object (or subclass of **URLConnection**). It also parses the URL, so you can define custom URL formats (such as the **jar:** protocol that **JarURLConnection** object uses). The **URL-Connection** object communicates with the server.

Ordinarily, you don't care about any of this because it just works transparently. If you want to add to a custom handler, however, you can do so by creating your own objects. Why would you want to create a handler? You might want to extend Java's **URL** object so that it understands, for example, the **Finger** protocol. You could define custom URLs that point to database tables or other custom resources. You'll have to do a bit of work at first, but after you have the classes available it will be easy for you (or anyone else) to access your custom content. You can even load new handlers at runtime. Listing 8.8 shows a subclass of **URLConnection** that opens a time server on port 13. The **connect** method is where most of the work occurs. This method opens a socket on the correct port and sets the connected flag (part of **URLConnection**). Because this connection directly wraps a socket, the **getInputStream** method is trivial.

Listing 8.8: A subclass of URLConnection that opens a time server.

```
import java.net.*;

import java.io.*;


public class TimeURLConnection extends URLConnection {

      private Socket conn;

      public final static int DEFPORT=13;


     public TimeURLConnection(URL url) { super(url); }


     public synchronized void connect() throws IOException {

        if (!connected) {

             int port=url.getPort();

            if (port<=0) port=DEFPORT;

             conn=new Socket(url.getHost(),port);

              connected=true;

             }

       }


     public String getContentType() {

        return "text/plain";

        }


     public synchronized InputStream getInputStream()

               throws IOException {

         connect();
```

```
        return conn.getInputStream();
        }
}
```

How does the **URL** object know to use this particular connection object? The first part required is a **URLStreamHandler** subclass. This object creates the correct **URLConnection** object and is quite simple (see Listing 8.9).

Listing 8.9: A simple class that creates the correct URLConnection subclass on behalf of the URL class.

```
import java.net.*;
import java.io.*;

public class TimeHandler extends URLStreamHandler {
   public int getDefaultPort() {
       return TimeURLConnection.DEFPORT;



   protected URLConnection openConnection(URL url)
         throws IOException {
      return new TimeURLConnection(url);
    }
}
```

You still have to instruct the system to use this **URLStreamHandler** object. When you create a **URL** object, Java first looks to see if you've installed a **URLStreamHandlerFactory** class. If you have, Java calls the **createURLStreamHandler** method of this class. If you didn't provide this class (or if the method returns **null**), the system then looks in the system property named **java.protocol.handler.pkgs**. This property might contain a list of package names, separated by the vertical-bar character (|)—sometimes called a *pipe character*.

If this property has a value, Java looks for a class that matches the protocol you are trying to use. For example, suppose the **java. protocol.handler.pkgs** property equals "com.al_williams.

proto|com.coriolis.java.phandlers" and you are trying to load a **mailto** URL. Java will look for these classes along the **CLASSPATH** :

com.al_williams.proto.mailto.Handler

com.coriolis.java.phandlers.mailto.Handler

This class, if it exists, is a subclass of **URLStreamHandler** (as shown in Listing 8.9). If you don't have a class that fits this description, Java finally looks for a class named **sun.net.www.protocol.mailto.Handler** (for the **mailto** protocol).

Of course, the custom **URLStreamHandlerFactory** offers the most control over the process, and it is simple to write (see Listing 8.10). Just remember that you can install only one **factory** class. Subsequent attempts to set the **factory** class will throw an exception.

Listing 8.10: The URLStreamHandlerFactory class selects a custom handler for a protocol.

```
import java.net.*;


public class MyStreamHandlerFactory
            implements URLStreamHandlerFactory {
    public URLStreamHandler
              createURLStreamHandler(String protocol){
        if (protocol.equalsIgnoreCase("time"))
       return new TimeHandler();
      return null; // huh?
       }
}
```

If you are using a custom factory class, you should set it by using the static **URL.setURLStreamHandlerFactory** method, as the program in Listing 8.11 does.

Listing 8.11: This simple program uses the custom protocol handler

```
import java.net.*;

import java.io.*;
```


Team-Fly®

```
public class TimeURLTest {

  public static void main(String [] args) throws Exception {

    URL.setURLStreamHandlerFactory(new MyStreamHandlerFactory());

    URL url=new URL("time://tock.usno.navy.mil");

     InputStream is=url.openStream();

    int c;

    do {

          c=is.read();

         if (c!=-1) System.out.print((char)c);

    } while (c!=-1);

     is.close();

   }

}
```

Custom Content Handlers

The protocol handler and connection objects are purely concerned with the transmission of data. However, the **URL** object also converts data into a Java object when you call the **getContent** method. How can Java know what type of object to use? That is the province of a content handler derived from **ContentHandler**.

The class itself implements the **getContent** method, which returns the appropriate type. Alternatively, the calling program can specify a list of classes, and **getContent** will return a type from this list, if possible.

The process that Java uses to find appropriate content handlers is similar to the one it uses for protocol handlers. First, the **URL** object calls the **URLConnection** object's **getContent** call. If you are writing a custom **URLConnection** object, you can handle everything at this point.

The default processing for **getContent**, however, checks whether you have installed a **ContentHandlerFactory** class by using the static **setContentHandlerFactory** method of **URLConnection**. If you have, Java calls the **createContentHandler** method to find a content handler. If this class has not been installed or the method returns **null**, Java examines package names in the system property **java. content.handler.pkgs** .

The **java.content.handler.pkgs** property contains a list of package names (separated by the vertical-bar character). Java uses the MIME type as a class name and searches the packages listed for that class. For example, to locate a handler for the type application/x-video, Java would examine the packages for a class named **application.x_video** (you convert the dashes to underscores because dashes are not legal characters in class names).

If all else fails, Java looks for the default handler. For the example MIME type, the default handler would be at **sun.net.www.content. application.x_video**. You'll find an example content handler in the Immediate Solutions section of this chapter.

Immediate Solutions

Resolving a Hostname

If you want to transform a hostname to an IP address, you can use the **InetAddress** class. In many cases, this same class can resolve an IP address to a hostname, although this is not always possible. You can't construct an **InetAddress** object directly. Instead, you call one of three static member functions: **getLocalHost**, **getByName**, or **getAllByName**.

Once you have an **InetAddress** object, you can call **getHostName** or **getHostAddress** to find the corresponding name or IP address. You can also call **getAddress** to get the IP address as an array of bytes. Listing 8.12 shows an excerpt of the code required to use **InetAddress**.

Listing 8.12: Finding a hostname or IP address by using InetAddress.

```
try {
   address=InetAddress.getByName("www.coriolis.com");
   }
catch (UnknownHostException e) {
  System.out.println("I can't find host");
   System.exit(1);
   }
System.out.println(address.getHostName() + "="
   + address.getHostAddress());
```

Opening a TCP Socket to a Server

When you create a **Socket** object, you specify the destination address and port in the constructor. You can specify the destination address as a string or as an **InetAddress** object. If the computer has multiple network interfaces, you can also specify the local port and the outgoing interface to use, either as a string or as **InetAddress**.

When you construct a socket, you might trigger an exception if the hostname is unknown or if any other error occurs. Listing 8.13 shows an example of opening the Web server at http://www.coriolis.com.

Listing 8.13: Opening a socket requires exception-handling code.

```
try {
  Socket sock = new Socket("www.coriolis.com",80);
  }
catch (UnknownHostException e) {
  // not found
  }
catch (java.io.IOException ioe) {
  // other error
  }
```

Opening a Server Socket

When you want to start a server, you can instantiate a **ServerSocket** object. The minimal constructor requires a port number that the socket will use to listen for incoming requests. If you wish, you can also suggest a queue size for incoming requests. Also, if there are multiple network interfaces, you can specify an **InetAddress** object to select a specific address.

Once a **ServerSocket** is instantiated, you can wait for a connection with **accept**. This will return a **Socket** object that you can use to communicate with the client. Client connections are possible only when the server calls **accept**; then a single client will connect. If no clients are waiting to connect, the call will block (subject to any timeout set by **soSetTimeout**).

Industrial-strength servers usually create separate threads to handle incoming requests so that other clients will not have to wait to connect. You'll find an example in Listing 8.14.

Listing 8.14: The ServerSocket class makes it easy to accept client connections.

```
try {

ServerSocket ssock = new ServerSocket(2222); // port 2222

while (true) {

  Socket sock=ssock.accept();

  HandleClient(sock);

  }

 }

 catch (java.io.IOException e) {

}
```

Creating a UDP S ocket

For some protocols, or for maximum efficiency, you might want to use UDP sockets instead of TCP. UDP sockets do not ensure data delivery. There is also no guarantee that data will arrive in the order in which it was sent. If you use these sockets, you'll have to provide an alternate way to deal with transmission loss and sequencing.

When you send and receive data via UDP, you use a **DatagramPacket** object. This object allows you to set an IP address (which can be a broadcast address) and a port number. The object also contains a byte array that contains the data to be sent or received. You pass the packet object to the **DatagramSocket** object's **send** or **receive** methods.

If you are doing multiple transactions with one server, consider calling **DatagramSocket.connect** to set a connection to the remote computer. Doing this reduces the overhead involved with sending each packet to the remote computer. However, then you can't send to other computers until you call **disconnect**.

UDP sockets don't use streams. You simply send and receive arrays of bytes. You'll find an example in Listing 8.15.

Listing 8.15: UDP sockets use DatagramPacket objects to send and receive data.

```
byte [] ary= new byte[128];

DatagramPacket pack=new DatagramPacket(ary,128);
```

```
if (reading) {
// read
  DatagramSocket sock=new DatagramSocket(portnum);
   sock.receive(pack);
  String word=new String(pack.getData());
   System.out.println("From: " + pack.getAddress()
    + " Port: " + pack.getPort());
   System.out.println(word);
   sock.close();
} else { // write
  DatagramSocket sock=new DatagramSocket();
   pack.setAddress(InetAddress.getByName(hostname);
   pack.setData(dataString.getBytes());
   pack.setPort(portnum);
   sock.send(pack);
   sock.close();
}
```

Sending Data to a TCP Socket

Once you have a **Socket** object (either from constructing one or as a result of a **ServerSocket.accept** call), you can fetch a stream by calling **getOutputStream**. You can then write data to this stream. The **getOutputStream** method, which is part of the **java.io** package, allows you to write bytes (or a byte array) to the socket for reading by the other computer.

You normally won't use the stream directly. Instead, you'll add one or more stream filters to make things simpler. For example, you might use **BufferedOutputStream** to buffer bytes to improve performance. A **DataOutputStream** object allows you to write primitive data types, or an **ObjectOutputStream** object can write entire objects as long as they are serializable. Listing 8.16 shows a simple example that opens a socket on a remote computer and uses the **DataOutputStream** filter to send an integer.

Listing 8.16: A TCP socket uses streams to send data.

```
void sendToSocket(String host, int port) throws Exception {

  Socket sock = new Socket(host,port);

  DataOutputStream os =

        new DataOutputStream(sock.getOutputStream());

   os.writeChar('X');

   os.writeChar('Y');

   os.close();

   sock.close();

   }
```

Receiving Data from a TCP Socket

Receiving data requires you to call **getInputStream**. Once you have this input stream object, you can use it to read raw bytes from the socket. If you issue a **read** and no data is available, the **read** call will block unless you've set a timeout.

You might want to use filters to provide extra features, such as buffering (**BufferedInputStream**) or data reception (**DataInputStream**). Listing 8.17 shows a simple function that receives data from a TCP socket.

Listing 8.17: TCP sockets deliver data via streams.

```
void sockRcv(Socket sock) throws Exception {

   char c1,c2;

  DataInputStream is = new DataInputStream(sock.getInputStream());

   c1=is.readChar();

   c2=is.readChar()();

   is.close();

   }
```

Compressing Socket Data

Because you can add filters to input and output streams, you can perform sophisticated processing, such as compression and decompression, with relative ease. The **java.util.zip** package has several stream classes you can use to compress and decompress data on the fly.

Listing 8.18 shows a program that receives data from a socket and decompresses it by using **GZIPInputStream** . This server accepts only one connection and terminates when the transaction is completed. Start the server with a port number on the command line.

Listing 8.18: Using Java's compressions classes to reduce the amount of data sent over a socket.

```
import java.net.*;

import java.io.*;

import java.util.zip.*;


public class CompRcv {

  public static void main(String[] args) throws Exception {

    ServerSocket ssock = new ServerSocket(

          Integer.parseInt(args[0]));

     System.out.println("Listening");

     Socket sock=ssock.accept();

    GZIPInputStream zip = new GZIPInputStream(

        sock.getInputStream());

    while (true) {

       int c;

       c = zip.read();

       if (c==-1) break;

        System.out.print((char)c);

        }

     }
```

```
}
```

Listing 8.19 contains a companion client that sends a file to the server in Listing 8.18. Start the client with the hostname, the port number, and a file to send on the command line. This client and server work just the same as if they did not compress the data stream. The only difference is that they filter their streams with the special classes that compress data.

Listing 8.19: Sending a compressed file to the server in Listing 8.18.

```
import java.net.*;

import java.io.*;

import java.util.zip.*;



public class CompSend {

  public static void main(String[] args) throws Exception {

     Socket sock=new Socket(args[0],Integer.parseInt(args[1]));

    GZIPOutputStream zip = new GZIPOutputStream(

        sock.getOutputStream());

      String line;

    BufferedReader bis = new BufferedReader(

        new FileReader(args[2]));

     while (true) {

        try {

           line=bis.readLine();

           if (line==null) break;

           line=line+"\n";

           zip.write(line.getBytes(),0,line.length());

           }

       catch (Exception e) { break; }
```

```
        }
    zip.finish();
    zip.close();
    sock.close();
   }
}
```

Setting a Socket Read Timeout

When you make any call to read from a socket, the call normally blocks until the read is completed. If you want a different behavior, you can call **Socket.setSoTimeout**. The timeout is expressed in milliseconds and defaults to zero, which indicates that there's no timeout at all. If there is a timeout value and the read is not completed in the specified interval, the read will throw an **InterruptedIOException**.

As an example, suppose you made this call to set a 1-second (1,000-millisecond) timeout:

```
Sock.setSoTimeout(1000); // may throw SocketException
```

Then, assuming you have an input stream named **is**, you might write:

```
try {
    c=is.read();
     }
catch (InterruptedIOExecption e) { /* Time out! */ }
```

If you find yourself using **setSoTimeout**, you should think about using threads instead. Usually, doing reads in one thread and processing in another leads to a more robust solution. You can also use **getSoTimeout** to read the value back.

Setting a Server Accept Timeout

When you call **ServerSocket.accept**, your program will stop until a client connects. If you want to set a timeout, you can call **Server-Socket.setSoTimeout** to set a timeout in milliseconds. If you find yourself doing this, however, you should rethink your design and use threads. By using threads, you can wait for client connections without stopping your processing. However, if you want to use the timeout, you can call **getSoTimeout** to read the value. Here's an example:

```
ssock.setSoTimeout(1000); // may throw SocketException
```

Then a call to **accept** will throw an **IOException** if a connection is not made within one second.

Setting SoLinger

Sometimes you might close a socket before all the data you've sent has been transmitted to the other computer. In this case, you can call **setSoLinger** on the **Socket** object to set a time (in seconds) to wait for the data to clear before closing the socket. You can call **getSoLinger** to retrieve the value. The following code sets the linger time to five seconds, and it may throw a **SocketException**:

```
sock.setSoLinger(true,5);
```

If you set the first parameter to **false**, you'll turn the linger feature off; this is the default state. Obviously, this call applies only to TCP sockets because UDP sockets don't have a connection to close.

Setting Socket Delay Behavior

To make data transmission more efficient, the underlying operating system usually buffers data and does not send a packet until the other computer acknowledges the previous transmissions. For some applications—notably, those that are interactive—this might not be appropriate. If you want to change this behavior, call **socket. setTcpNoDelay(true)**. Call **getTcpNoDelay** if you want to determine the current value of this flag.

Tip RFC896 defines Nagle's algorithm, which is the algorithm used to determine when to send packets. You'll find a list of commonly used RFCs in Appendix D.

Setting Keep-Alive Options

Suppose that a client and server connect and then infrequently exchange data. The server could crash, and the client wouldn't notice until the next time the client sends data. If you call **Socket.setKeep-Alive** and pass **true** as an argument, the sockets will periodically send meaningless data just to see if the connection is still alive. Here's an example:

```
sock.setKeepAlive(true); // may throw SocketException
```

Your program won't see this data; if the sockets stop communicating, however, your program will receive an exception, even if you're not actively transmitting data. You can call **getKeepAlive** to find the current setting.

Setting Buffer Sizes

TCP sockets can buffer data, although the exact details are platform-dependent. You can make a suggestion to the operating system by calling **Socket.setReceiveBufferSize** and **Socket.setSendBuffer-Size**. These values are only suggestions. You can also call **getReceive-BufferSize** and **getSendBufferSize** to read the buffer sizes. Here's an example:

```
sock.setSendBufferSize(sock.getSendBufferSize+1024);
```

Of course, these calls might throw a **SocketException**.

Team-Fly®

Handling Socket Exceptions

Many socket operations can throw exceptions. Four common exceptions you might want to handle are the following:

- **java.io.IOException**—Occurs when there is a general I/O error.
- **java.net.BindException**—Occurs when the requested port is in use.
- **java.net.ConnectException**—Occurs when a client is unable to connect to the server.
- **java.NoRouteToHostException**—Occurs when a network problem prevents the program from finding the host.

You can catch the last three exceptions by handling **Socket-Exception**, which is the base class for all three.

Creating a Multithreading Server

The basic procedure for creating a simple server is the following:

1. Create a **ServerSocket** class.
2. Call **accept**.
3. Perform necessary processing.
4. Go back and call **accept** again.

If any significant processing is required, however, this approach isn't efficient. You'd like to let the server handle each client in a separatethread. The easiest way to do that is to make your class extend **Thread**. Then you can write your processing in the **run** method. In the main routine (usually **static**), you can call **accept**, create a new instance of your class (probably passing it the socket returned by **accept**), and then call **start** to run the thread. You can find an example of this in Listing 8.20.

Listing 8.20: A basic multithreaded server.

```
import java.net.*;

import java.io.*;



public class AMTServer extends Thread {

    Socket csocket;

   AMTServer(Socket csocket) { this.csocket = csocket; }

    public static void main(String args[]) throws Exception {
```

```
        ServerSocket ssock=new ServerSocket(1234);
        while (true) {
           new AMTServer(ssock.accept()).start();
         }

      }
   public void run() {
// client processing code here
      }
   }
```

Automating the Multithreaded Server

Because the logic behind a multithreaded server is so predictable, you might be tempted to write a common base class for all multithreaded servers. That's a great idea, but there is a problem. You'd like to create a single base class that can instantiate your classes that perform client processing. However, that's hard to do without some special tricks.

You'll see that the code in Listing 8.21 uses an unusual technique to allow one static member to create an instance of the class—even if that class is a subclass of the **MTServerBase** class. The idea is to pass a **Class** object into **startServer**. The method then uses the **newInstance** method to create the object. This approach assumes that your subclass has a default constructor, although you can use the reflection functions in **java.lang.reflection.Constructor** to call a nondefault constructor if you want to modify the base-class code. This technique allows you to extend **MTServerBase** without providing a custom version of **startServer**. You can find a class object for a particular class type by appending **.class** to the ordinary name. If you have an instance of an object, you can use **getClass** to do the same thing. Because you pass the **startServer** method your class object, **startServer** can create an instance of your object by using the **newInstance** method. Therefore, you don't have to replace the **startServer** method in your subclass.

Listing 8.21: Using a base class to build multithreaded servers easily.

```
import java.net.*;
import java.io.*;
```

```
public class MTServerBase extends Thread {

    // client

    protected Socket socket;


   // Here is the thread that does the work

   // Presumably you'll override this in the subclass

    public void run() {

        try {

          String s = "I'm a test server. Goodbye";

            socket.getOutputStream().write(s.getBytes());

            socket.close();

            }

        catch (Exception e) {

            System.out.println(e);

            }

       }


    static public void startServer(int port,Class clobj) {

    ServerSocket ssock;

    Socket sock;

            try {

          ssock=new ServerSocket(port);

          while (true) {

          Socket esock=null;

         try {
```

```
                esock=ssock.accept();
// create new MTServerBase or subclass
                MTServerBase  t=(MTServerBase)clobj.newInstance();
                t.socket=esock;
                t.start();

            } catch (Exception e) {
                    try { esock.close(); } catch (Exception ec) {}
              }
            }
        } catch (IOException e) {
        }
        // if we return something is wrong!
      }


    // Very simple test main
        static public void main(String args[]) {
          System.out.println("Starting server on port 808");
            MTServerBase.startServer(808,MTServerBase.class);
          }
        }
```

Listing 8.22 provides an example of how you can use this server base class to form another server. This particular server, which accepts the port number on the command line, converts lowercase input (from a Telnet program, for example) to uppercase. You can end the server by pressing Ctrl+C or Ctrl+D.

Notice that the only code in the subclass is the **run** method and a new **main** method to start the server properly. You can make any number of servers that derive from **MTServerBase** and focus on just the client interactions.

Listing 8.22: An example server using the MTServerBase object.

```
import java.net.*;

import java.io.*;


public class UCServer extends MTServerBase {


  public void run() {

       try {

          InputStream is=socket.getInputStream();

          OutputStream os=socket.getOutputStream();

           while (true) {

               char c=(char)is.read();

// end on Control+C or Control+D

              if (c=='\003' || c==' \004') break;

              os.write(Character.toUpperCase(c));

              }

       socket.close();

       }

        catch (Exception e) {

            System.out.println(e);

           }

         }
```

```
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
   System.out.println("Starting server on port " + n);
     startServer(n,UCServer.class);

    }
  }
```

Using the URL Class to Get Accurate Time

One trick you can perform with the **URL** class is to request the time from the National Institute of Standards and Technology (NIST) by pointing your **URL** object to **http://www.132.163.4.101:14/**. You get a time back with several parts. Just parse what you need. You can make a clock with the code in Listing 8.23.

Listing 8.23: Building a NIST standard time checker.

```
import java.io.*;
import java.net.*;

public class WhatTimeIsIT
{
  public static void main(String[] args) throws Exception
   {
      String date = "", time = "", html = "";
     int dateStart = 6, dateEnd = 15;
     int timeStart = 15, timeEnd = 23;
      String url_NIST_clock = "http://www.132.163.4.101:14/";

     URL nistClock = new URL(url_NIST_clock);
```

```
        BufferedReader page = new BufferedReader(
                new InputStreamReader( nistClock.openStream() ) );


        StringBuffer pageBuffer = new StringBuffer();
        while ((html = page.readLine()) != null)
         {
             pageBuffer.append(html);
         }


         page.close();


         html = pageBuffer.toString();
        date = html.substring(dateStart, dateEnd);
        time = html.substring(timeStart, timeEnd);


         System.out.println(html);
         System.out.println(date);
         System.out.println(time);
    }
}
//returns:
//52172 01-09-20 06:05:22 50 0 0 979.7 UTC(NIST) *
//01-09-20
//06:05:22
```

The previous program is simple. The explanation of the signal is given at http://www.boulder.nist.gov/timefreq/service/its.htm .

Using Web Services

Just as the last program retrieved the time, you can just as easily fetch stock quotes, weather reports, or any other Web-enabled data source. For example, the program in Listing 8.24 shows how you might get a stock quote from Yahoo.

Listing 8.24: Building a Yahoo stock-quote fetcher.

```
import java.io.*;

import java.net.*;



public class GetYahooQuote

{

   public static void main(String[] args) throws Exception

    {

        String ticker = "SUNW", html = "", quote = "";

       if (args.length == 1)

        {

            ticker = args[0].toUpperCase();

        }



       String yahooQuote = "http://www.finance.yahoo.com/q";



       URL quoteURL = new URL(yahooQuote);

       URLConnection connection = quoteURL.openConnection();

        connection.setDoOutput(true);



       PrintWriter out = new PrintWriter(

                                connection.getOutputStream());
```

```
        out.println("s=" + ticker + "&d=v1");
        out.close();
        BufferedReader yahooPage = new BufferedReader(
            new InputStreamReader(
            connection.getInputStream()));

        StringBuffer pageBuffer = new StringBuffer();
        while ((html = yahooPage.readLine()) != null)
        {
            pageBuffer.append(html);
        }
        yahooPage.close();
        html = pageBuffer.toString();

        int start = html.indexOf("<b>Trade</b>");
        start = html.indexOf("s=" + ticker, start);
        start = html.indexOf("<td", start);
        start = html.indexOf("<td", start);
        start = html.indexOf("<b>", start);
        int end = html.indexOf("</b>", start);
        quote = html.substring(start + 3, end);

        System.out.println(quote);
    }
}
```

```
//java GetYahooQuote ibm

//returns:

//96.00
```

This program is only a trivial variation of the time program, but there is one major difference. Notice the highlighted lines. These lines are actually writing *to* the Web server. This is where we are sending data that mimics Yahoo's stock-quote form. You can send any character data this way. If you are reverse-engineering a Web service like this, it is useful to use the View Source menu on your browser to look at the form's HTML directly.

Tip Use the **PrintWriter** to send text data, but the **PrintStream** to send binary. Sun's documentation says you should use the **PrintStream** to send character data, as bytes, but not use it for non-character data. Also, the **PrintWriter** methods never throw I/O exceptions, so you use **checkError()**.

TEAMFLY

Team-Fly®

Chapter 9: XML and Java

In Brief

Before Web browsers, the Internet was just a widespread computer network. There were lots of things you could download—if you knew where and how to find the information you were looking for. What Web browsers—and, in particular, HTML—provided was an easy way for people to view information; they also provided links to other information on the Web.

HTML focuses on displaying data and managing hyperlinks. It isn't good at actually describing data (except for the data that relates to formatting a Web page). That's where XML comes in; it is used for managing data instead of display.

Introduction to XML

Superficially, XML (Extensible Markup Language) files look like HTML (Hypertext Markup Language) files. These markup languages function slightly differently, but the biggest difference is their intended purpose. HTML allows you to describe a Web page's appearance. Browsers on multiple platforms render this HTML in an appropriate way. For example, a browser running on a workstation (with a resolution of 1,280× 1,024) shows a page much differently than does a Windows CE handheld computer and its credit card—size screen.

In contrast, XML does not describe data presentation at all. Rather, XML stores information. Of course, plenty of database products exist for storing data. The difference between an ordinary database and data in XML format is simple: XML is very portable. Not only can XML documents traverse platforms, but programs can process XML without prior knowledge of an XML file's contents. The XML specification (the official document is at http://www.w3.org/TR/2000/REC-xml-20001006) uses tags that are similar to HTML tags. Unlike HTML tags, XML tags are case-sensitive. Also, XML defines few standard tags. Instead, you are free to make up any tags you like (with a few restrictions). There are a few other minor differences. For example, HTML has many tags that don't have corresponding closing tags (for example, **
**), whereas XML always requires closing tags (although there is a way to write a tag that opens and closes all at

once). In XML, for another example, attributes must have quotes, even if they don't contain spaces. In HTML, quotes are optional if the attribute value doesn't contain spaces. As an example, the following is legal HTML:

```
<A HREF=t.jsp>
```

In XML, however, anything after the equal sign (the attribute value) must be in quotes. So an XML document might contain:

```
<XLINK URI="t.jsp">
```

Because XML doesn't prescribe tags, it is up to the data producer and the data consumer to agree on the various tags. Because the tags have a rigorous format, however, tools that don't know specifics

about the tags can still identify them. A tool might not understand what the tag means, but the tool can still identify the parts of the document.

Here is a simple XML document:

```
<?xml version='1.0' encoding='UTF-8'?>
<item>
    <ball id="97">
        <color>blue</color>
        <size  units="mm">50</size>
        <shippable/>
    </ball>
</item>
```

The first element is one of the few predefined tags. This tag identifies this document as following XML version 1 and as using the 8-bit Unicode Transformational Format (UTF-8). Apparently, the creator of this file wants to describe an item. In particular, this item is a ball that has an ID of 97, a color of blue, and a size of 50 millimeters. Each XML file has a single root tag (in this case, **item**), which contains all the other tags.

The **shippable** tag is a special type of tag. XML requires that all tags have a corresponding closing tag. In the previous example, the way the **shippable** tag appears indicates that it is both an opening and a closing tag, all in one. This is equivalent to:

```
<shippable></shippable>
```

If that was all there was to XML, it would be simple, right? Actually, XML itself is just this simple. The complexity arises in processing the XML, however.

I mentioned earlier that programs that don't know anything about our **item** document can still correctly identify the pieces that make up the document. The downside is that the program can't tell if the document meets any rules other than XML's basic rules. Suppose that you and I want to exchange these item documents, and we agree that **item** is the root tag and the only other allowed tags are **ball**, **cube**, and **rod**. Furthermore, each of these tags can contain only **color**, **size**, **shippable**, and **price** tags. The **shippable** and **price** tags are optional, but the other tags are not. Each **item** has only one sub-tag, and each sub-tag can have only one **color**, one **size**, one **shippable**, and one **price** tag (balls don't have two prices). Of course, to be precise, we'd also need to agree on the format of an ID, the acceptable units for the **size** tag, and the valid colors.

A program that processes generic XML can't test these rules because they constitute a private agreement between us. However, there are two ways you can inform programs (and people) about the format that an XML document should have. The original method is to use a Document Type Definition (DTD). This is another file format that describes the rules for an XML document. A newer

method, XSchema (sometimes simply called a *schema*), describes the document's format with another XML document (which has a specific, published format defined by a DTD). By examining a document and its related DTD or schema, any program can verify that an XML document is correct, even if the program doesn't understand what the document contains.

XML Processing

To simplify working with XML, you can use programs known as *parsers*, which can interpret an XML document. Of course, you usually want to do more than just break the XML into its constituent parts. You probably want to act on the data in the XML file. All parsers take the following general steps:

1. Remove any comments.
2. Start tokenizing the document into recognizable elements.
3. Verify the **?xml** tag. (Is it XML? What version? What encoding?)
4. Validate the document's structure against a schema or a DTD.
5. Provide results of the parse to the caller.

Naturally, the last step is the tricky part. The parser knows only how to identify pieces of the XML document. Somehow it has to communicate those pieces to your custom code.

Parsers can communicate an XML document to your program in one of two ways. A SAX (Simple API for XML) parser calls your program immediately each time it detects a unique XML element. Using a SAX parser is fast and easy, but it adds extra overhead to your program if you want to work with relationships between elements. After all, each call occurs in isolation from the others. If your program wants to study multiple nodes, it will have to store information between calls from the parser.

More complex parsers follow the DOM (Document Object Model). They return the entire document in one piece structured as a tree. The parser will return an object that represents the root tag of the document. Using methods in that object, you can enumerate the immediately subordinate tags (also as objects). Of course, those objects might also have subordinate tags. A DOM parser allows you to process the entire document easily, but it also requires that the parser completely process the document before calling your program. As you might expect, this takes more memory and requires more time before you can start processing.

Immediate Solutions

Building an XML Parser to Serve as an RPC Engine

For some purposes, you might just want to write your own simple XML parser. Suppose that you want to call a method that is on another machine. This process is commonly called a *Remote Procedure Call (RPC)*, and there are many ways to handle these calls. For now, suppose that you want to make a simple RPC system that uses XML to direct the remote machine's actions and receive a response from the remote machine.

Tip One common model of RPC is the Open Software Foundation's Distributed

Computing Environment (DCE). The Institute of Electrical and Electronics Engineers has defined RPC in its ISO Remote Procedure Call Specification, ISO/IEC CD 11578 N6561, ISO/IEC, November 1991.

We will keep our RPC schema and its associated markup language modest, however; employing RPC makes the XML application more compelling. The following is our sample XML document:

```
<!--method calling XML-->
<?xml version="1.0" encoding="UTF-8"?>
<rpc>
    <method>addition</method>
    <argument>5</argument>
    <argument>97</argument>
    <argument>30</argument>
    <argument>221</argument>
    <argument>97</argument>
    <argument>4813</argument>
</rpc>
```

The goal for our XML-based RPC project is to compile a list of numbers, send it to another machine where the list is added, and have the result shipped back. The previous XML document has only five parts:

- Comment
- Prolog (the **?xml** tag)
- Root (the **rpc** tag)
- Method name (**addition** in the above example)
- Argument list

For this example, two classes will simulate the two disparate machines and dispense with actual network code. The "server" will have four tasks:

1. Validate the request; that is, verify that the XML is properly formed and has the necessary content.
2. Choose the action category; that is, extract the method name.
3. Read the arguments.
4. Perform the requested action.

The code will verify the XML document in the same way that commercial parsers do it (minus niceties like ignoring white space), like this:

```
int isXML = xml.startsWith("<?xml");
```

Now we need to know which version of XML is being used. You need this information because later versions might require a change in your processing rules. The program performs an easy string scan:

int isVersion_1 = xml.indexOf("version=\"1.0\"");

Now we need to find out if the document is encoded properly, like this:

int isEncoded = xml.indexOf("encoding=\"UTF-8\"");

If the document passes these three tests, then we process it. For our custom parser, we need to know what the operation is (addition, in the example XML) and which arguments are provided. Listing 9.1 shows the parser class, a test **main**, and some helper classes.

Listing 9.1: Listing 9.1 Building a basic XML parser.

```
import java.util.*;


public class xmlProcessor

{

   public static void main(String[] args) throws Exception

    {

// Build an example XML document

       String xml = " <!--method calling XML--> " +

        " <?xml version=\"1.0\" encoding=\"UTF-8\"?>" +

        "<rpc>" +

        " <method>ADD</method>" +

        " <argument>5</argument>" +

        " <argument>97</argument>" +

        " <argument>30</argument>" +

        " <argument>221</argument>" +

        " <argument>97</argument>" +

        " <argument>4813</argument>" +

         "</rpc>";


       xmlHelper xmlhelp = new xmlHelper(xml);
```

```
        xmlhelp.stripComments(); // remove comments
// make sure document is OK
        boolean isXML = xmlhelp.validate();


        if ( isXML )
         {
// read operation (i.e. ADD)
            String operation = xmlhelp.getOperation();
// get arguments
            List parameter = new
                    ArrayList( xmlhelp.getList("argument") );
// Create class that does the math work
           mathHelper mathhelp = new mathHelper();
           String result = mathhelp.math(operation, parameter);
// show result
              System.out.println(result);
        }  else
        {
             System.out.println("Error: couldn't process XML");
        }
    }
}


// This class takes the parsed command and argument list
// and performs the specified operation
```

```
class mathHelper
{
    final int ADD = 64641;
    final int MULTIPLY = 1436456484;
    final int SUBTRACT = -1277621484;
    final int DIVIDE = 2016833657;

  public String math(String operation, List parameter)
   {
      int choice = operation.toUpperCase().hashCode();
      double total = 0;
      int items = parameter.size();
     for(int index = 0; index < items; index++)
       {
         String number = (String)parameter.get(index);
         double d = Double.parseDouble(number);
          switch (choice)
          {
              case ADD:
                 total = total + d;
                   break;
              case MULTIPLY:
                 total = total * d;
                   break;
              case SUBTRACT:
```

```
                    total = total - d;
                    break;
                case DIVIDE:
                    total = total / d;
                    break;
            }
        }

        return Double.toString(total);
    }
}

// The simple-minded XML parser
class xmlHelper
{
    private String xml;
    private String xmlLine;
    public int start, end, position;
    public StringBuffer sb = new StringBuffer();

    xmlHelper(String xml)
    {
        this.xml = xml.trim(); // eat spaces at ends
    }
    public String getOperation()
```

```
    {
// get operation
        String line = this.getText("<method>",
              "</method>",0,false,false);
        return line.trim();
    }
    public String getXML()
    {
        return this.xml;
    }
// get arguments from specified tags into a List
    public List getList(String tag)
    {
        boolean moreList = true;
        List aList = new ArrayList();
        int oldStart = 0, count = 0;
        while (moreList && (count < 100) )
          {
             count++;
            moreList = false;
            start = xml.indexOf("<"+tag+">", start);
            if (start > 0)
             {
                String line = this.getText("<"+tag+">","</"+tag+">",
                                                    start,false,false);
```

```
                aList.add(line);
               if(line.length() > 0)
                {
                    moreList = true;
                   start += line.length();
                }
            }
        }

        return aList;
    }
// Remove comments that don't count (<!-- . . . -->)
      public boolean stripComments()
       {
          boolean isComment = true;
         int oldStart = 0, count = 0;
          start = 0;
          end = 0;
           sb.setLength(0);
         while (isComment && (count < 100) )
           {
               count++;
               isComment = false;
              start = xml.indexOf("<!–", end);
             if (start > -1)
```

TEAMFLY

Team-Fly®

```
            {
                end = xml.indexOf("-->", start);
                if (end > -1)
                {
                    sb.append( xml.substring(oldStart, start) );
                    sb.append( xml.substring(end + 3) );
                    oldStart = end;
                    isComment = true;
                }
            }
        }
        if (sb.length() > 0)
        {
            xml = sb.toString();
        }
        return isComment;
    }
// Check for valid looking document
    public boolean validate()
    {
        boolean result = false;
        String line = getText("<?", "?>",0,true,true);
        int isXML = line.indexOf("<?xml");
        int isVersion_1 = line.indexOf("version=\"1.0\"");
        int isEncoded = xml.indexOf("encoding=\"UTF-8\"");
```

```
        if (isXML>-1 && isVersion_1>-1 && isEncoded >-1 )
        {
            result = true;
        }

        return result;
    }
// find text associated with a pair of tags
   public String getText(String open, String close, int offset,
                         boolean includeOpen, boolean includeClose)
    {
       int openPosition = xml.indexOf(open, offset);
        if (!includeOpen)
        {
            openPosition = openPosition + open.length();
        }

       int closePosition = xml.indexOf(close, offset);
        if (includeClose)
        {
           closePosition = closePosition + close.length();
        }

       return xml.substring(openPosition, closePosition);
```

```
    }
}
```

You can do many things to tidy up the parser. For one thing, it doesn't grab XML files over the Internet or on the hard disk. Another obvious problem is that the parser doesn't handle white space properly. The parser also does not accept some valid Unicode characters. The specification says that all XML processors must accept UTF-8 and UTF-16 (encodings of ISO/IEC 10646, the mechanisms for signaling which of the two formats is in use). We can fix a few of these limitations if we create a string stream out of the XML document and read it one character at a time. One approach for parsing one character at a time is shown with the following pseudo-code snippet:

```
for (i=0;i<xml.length();i++)
{
    char c = xml.charAt(i);
    switch (c)
    {
        case '<':
            parseOpenTag();
            continue;

        case '/':
            parseCloseTag();
            continue;

        case '\t':
        case ' ':
        case '\n':
            space = true;
            continue;

        case 'A': case 'B': case 'C': case 'D': case 'E': case 'F':
        case 'G': case 'H': case 'I': case 'J': case 'K': case 'L':
        case 'M': case 'N': case 'O': case 'P': case 'Q': case 'R':
        case 'S': case 'T': case 'U': case 'V': case 'W': case 'X':
        case 'Y': case 'Z':
        case 'a': case 'b': case 'c': case 'd': case 'e': case 'f':
```

```
        case 'g': case 'h': case 'i': case 'j': case 'k': case 'l':
        case 'm': case 'n': case 'o': case 'p': case 'q': case 'r':
        case 's': case 't': case 'u': case 'v': case 'w': case 'x':
         case 'y': case 'z':
              addLetterToElement(c);
              continue;

        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
              addDigitToElement(c);
              continue;

          default:
              doSomethingWithWeirdCharacter(c);
    }
}
```

Installing Java XML Extensions

Sun Microsystems knows that many developers will want to download all the XML APIs and libraries. That's why Sun built the Java XML (JAX) Pack, provided at http://www.java.sun.com/xml/jaxpack.html. From this page, you can download one package that contains all of the following APIs and architectures:

- *Java API for XML Processing (JAXP)*—Reads and interprets XML files by using a standard interface to a variety of parsers.
- *Java Architecture for XML Binding (JAXB)*—Allows you to write a Java object as an XML file and to read XML files into an object.
- *Java Document Object Model (JDOM)*—Converts an XML file into a tree of Java objects.
- *Java API for XML Messaging (JAXM)*—Allows programs to exchange XML-based messages.
- *Java API for XML Registries (JAXR)*—Allows you to publish the existence of XML services for other programs to find.
- *Java API for XML-based RPC (JAX-RPC)*—Allows you to make RPC calls using XML.

To use JAXP, you'll have to have the jaxp.jar and crimsom.jar files in your CLASSPATH. In addition, JAXP can interface with third-party parsers (or you can use Sun's parsers). If you use another parser, you'll need to follow the instructions for the parser you are using. Typically, you'll need to include more

files in your CLASSPATH and modify the **javax.xml.parsers.SAXParserFactory** and **javax.xml.parsers. DocumentBuilderFactory** properties.

Creating a Parser

When you want to create an XML parser, you'll follow these basic steps:

1. Create a factory class by calling a static method (**newInstance**) of the factory class.
2. Use the methods from the factory class to create a new instance of the parser.
3. Call the **parse** method of the parser.

When creating a parser, you have two choices: using a SAX parser and using a DOM parser. A SAX parser allows you to process an XML document serially. When the parser detects different elements of the XML file, the parser calls methods that you specify. For this style of parser, you'lluse the **SaxParserFactory** class as the factory. The **newSAXParser** method then creates the actual **SAXParser** object.

Here's code from a program that constructs the SAX parser:

```
// Use the default (non-validating) parser
SAXParserFactory factory = SAXParserFactory.newInstance();
// Parse the input
SAXParser saxParser = factory.newSAXParser();
```

The process of using a DOM parser is similar. You use the **DocumentBuilderFactory** class as a factory object. The factory's **newDocumentBuilder** method creates a **DocumentBuilder** object. This object then creates a DOM from XML input (resulting in a **Document** object). Here's the code that creates the parser (without exception handling):

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
```

The SAX parser was designed to be fast, so it sprints sequentially through a document and fires events when it finds something interesting. This approach is efficient (in particular, it doesn't require much memory), but it isn't conducive to manipulating the document as a whole. Because the DOM parser builds an entire document tree, it doesn't have this limitation. However, this parser is slow and consumes more memory.

One common use for a DOM parser is to read a document, change it, and write the document back to a file or other location (such as a network connection). You can even start with an empty DOM tree and build a new document from scratch.

Use SAX if you want a quick read, and use DOM if you want to change the document in any significant way. You can change little things (such as capitalizing character data) with SAX. But if you want to change the document structure (such as by adding or sorting nodes), you need DOM.

Using a SAX Parser

SAX is an event-based API. When the SAX parser identifies an XML element, the parser calls a function you specify so you can process the element. The SAX parser reads each character, marching through the document sequentially, and calls your code as it identifies various events in the document. For example, when the parser finds the start of an element, it fires the **startElement** event. The parser uses many event callback functions (see Table 9.1).

Table 9.1: Methods called by SAX parsers.

Method	Parameter(s)	Description
* **startDocument**	None	Is called once at the beginning of the document.
* **endDocument**	None	Indicates the end of the document.
* **startElement**	**String namespaceURI, String localName, String qName, Attributes atts**	Is called at the start of each element. The parameters indicate the element found and any attributes included.
* **endElement**	**String namespaceURI, String localName, String qName**	Is called at the end of each element.
* **characters**	**char ch[], int start, int length**	Is called to report each chunk of ordinary character data.
ignorableWhitespace	**char ch[], int start, int length**	Reports a chunk of white space in element content (see the W3C XML 1.0 recommendation, section 2.10).
processingInstruction	**String target, String data**	Is called for processing instructions (tags that begin with **?xml**).
skippedEntity	**String name**	Is invoked once for each part of the document skipped.

Those methods preceded with an asterisk in Table 9.1 are the ones you will work with most often. The others are used less frequently. Suppose that you parsed this XML file with a SAX parser:

```
<?xml version='1.0' encoding='UTF-8'?>


<item>
    <ball id="97">
        <color>blue</color>
    </ball>
</item>
```

Table 9.2 shows the first few XML elements and their corresponding SAX callbacks. Notice that the ends of line characters (**\n**) are not part of anything, so the parser treats them as character data. You'll find a complete example in Listing 9.2.

Table 9.2: SAX events occurring while the previous XML document is parsed.

XML	Method Call
<	**startDocument()**
<item>	**startElement**
\n	**characters**
<ball>	**startElement**
\n	**characters**
<color>	**startElement**
blue	**characters**
</color>	**endElement**
\n	**characters**

Listing 9.2: Listing 9.2 Building a SAX parser.

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;


public class ShowSAXevents extends DefaultHandler
{
      public static void main(String args[])
      {
      if (args.length != 1)
       {
            System.err.println("Provide an XML document filename!");
             System.exit(1);
       }

     // Instance of this class as the SAX event handler
      DefaultHandler handler = new ShowSAXevents();

      // Use the default (non-validating) parser
      SAXParserFactory factory = SAXParserFactory.newInstance();
       try
       {
            // Parse the file
            SAXParser saxParser = factory.newSAXParser();
            saxParser.parse( new File(args[0]), handler);
```

```
        } catch (Throwable t)
        {
            t.printStackTrace();
        }

        System.exit(0);
    }

    // SAX events
    public void startDocument() throws SAXException
    {
        print("startDocument()");
    }
    public void endDocument() throws SAXException
    {
        print("endDocument()");
    }
    public void startElement(String namespaceURI,
                String lName, String qName,
                Attributes attribute) throws SAXException
    {
       String result = namespaceURI + ", " +
          lName + ", " +
          qName + ", ";
```

```
        if (attribute != null)
        {
         for (int i = 0; i < attribute.getLength(); i++)
            {
// Attr name
            String aName = attribute.getLocalName(i);
             result += " " +
                aName+"=\""+attribute.getValue(i)+"\"";
            }
        }

      result = "<" + qName + ">\t" + result;
        print(result);
    }

    public void endElement(String namespaceURI, String sName,
                           String qName) throws SAXException
    {
      String result = namespaceURI + ", " +
       sName + ", " +
        qName + ", ";
      result = "</" + qName + ">\t" + result;
        print(result);
    }
```

TEAMFLY

Team-Fly®

```
    public void characters(char buf[], int offset, int len)
                                    throws SAXException
    {
      String s = new String(buf, offset, len);
     int min = len > 5 ? 5: len;
       s = "\"" + s.substring(0, min) + "\"" + "\tbuf[], " +
           offset + ", " + len;
       printWhiteSpace(s);
    }

    private void print(String s)
    {
          System.out.println(s);
    }
    private void printWhiteSpace(String s)
    {
       s = s.replace('\n', '^');
      s = s.replace(' ', '_');
       System.out.println(s);
    }
}
```

Here is a transcript of executing the program on an example file:

```
$ java ShowSAXevents files/inventory.xml
startDocument()
<item> , , item,
"" buf[],_46,_0
"^" buf[],_0,_1
```

```
"___" buf[],_48,_3
<ball> , , ball,
"" buf[],_57,_0
"^" buf[],_0,_1
"_____" buf[],_59,_6
<color> , , color,
"blue" buf[],_72,_4
</color> , , color,
"" buf[],_84,_0
"^" buf[],_0,_1
"_____" buf[],_86,_6
<id> , , id,
"97" buf[],_96,_2
</id> , , id,
"" buf[],_103,_0
"^" buf[],_0,_1
"___" buf[],_105,_3
</ball> , , ball,
"" buf[],_115,_0
"^" buf[],_0,_1
</item> , , item,
endDocument()
```

The **character** method replaces each new line with a **^** and replaces spaces with underlines so we can see them better. Be careful with this method. As I mentioned earlier, the **character** method receives calls for new lines outside of meaningful data. One way to decide if the characters are really data is to keep track of the events and then determine if the event immediately preceding the character event was a **startElement**. If it was, then you are looking at characters inside a node. Otherwise, the characters are garbage. Try this:

```
final int STARTELEMENT = 1;
int lastEvent;
//in startElement: lastEvent = STARTELEMENT;
//in character(): if(lastEvent == STARTELEMENT)
{
    //process real data
```

```
} else
{
      //garbage so ignore
}
```

Most of your work will usually occur in the **startElement** and **character** event methods. You will probably poll for key tags and take action when you find a match like this:

```
startElement()
{
     //scan for certain element, then apply HTML code to it
      if (name.equalsIgnoreCase("color"))
      {
           //do something with name such as:
           htmlBuffer.append("<b>" + aname + "</b>: ");

           //can't do this because you haven't
           // received color data yet:
            //print("<font color=\"" + color + "\">";

           //so incrementally build a buffer instead:
            htmlBuffer.append("<font color=\"");
      }
}
characters()
{
     if(lastEvent == STARTELEMENT)
      {
           //first complete startelement html tag
            htmlBuffer.append(characterData);//data=color name
            htmlBuffer.append("\">");
           //now do something with string
      }
}
endElement()
{
```

```
        //scan for certain element, then apply HTML code to it
         if (name.equalsIgnoreCase("color"))
         {
         htmlBuffer.append("</font>");
         }
    }
```

We can also keep track of tags with a **Hashtable**. This is useful if you want to generate a count of tags, for example. Of course, a real program would probably read its input from the Web.

Related solution:	Found on page:
Using a **Hashtable**	129

Generating HTML

You can also improve the output by formatting the report as a Web page instead of as plain text. Displaying HTML is easy with Java's Swing components. You'll find a simple browser in Listing 9.3.

Listing 9.3: Building a browser with a GUI.

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

import javax.swing.*;

import java.io.*;

import java.awt.*;


public class WebBrowser {


    public static void main(String[] args) {

    // get the first URL

     String page = "http://www.google.com";

    if (args.length > 0) page = args[0];


   // set up the editor pane
```

```
    JEditorPane browser = new JEditorPane();
     browser.setEditable(false);


     try {
     browser.setPage(page);
     }
    catch (IOException e) {
            System.err.println(e);
            System.exit(-1);
     }
    JScrollPane scroller = new JScrollPane(browser);
    JFrame frame = new JFrame("Web Browser");
     frame.setDefaultCloseOperation(WindowConstants.
            DISPOSE_ON_CLOSE);
     frame.getContentPane().add(scroller);
     frame.setSize(750, 550);
     frame.show();
     }
}
```

It is easy to add HTML generation code to the event functions. Your code could be this simple:

```
startDocument()
{
        print("<html>");
        print("<head>");
        print("<style>");
        print("<!--");
```

```
        String styleSheet = getStyles();
         print(styleSheet);
         print("-->");
         print("</style>");
         print("</head>");
         print("<body>");
    }

    endDocument()
    {
         print("</body>");
         print("</html>");
    }

    startElement()
    {
         print("<b>");
         print(lName);
        print("</b>: ");
    }

    endElement()
    {
         print("<b>");
         print(lName);
        print("</b>: ");
    }

    characters()
    {
         print("<i>");
         print(data);
        print("</i>: ");
    }
```

Using a DOM Parser

The DOM Level 2 Specification defines a way to represent an XML document as a tree structure. The analogy describes a root node from which branches extend. Each node is itself a root for its sub-tree. This is a common structure in programming, and you've probably seen it before.

Tip The Document Object Model (DOM) Level 2 Core Specification is available at http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113.

The earlier section titled "Building an XML Parser to Serve as an RPC Engine" shows a simple RPC system that uses XML. However, the parser is quite simple and doesn't correctly handle all legal XML. You can fix the program by rebuilding it using a DOM parser. First, you have to load the JAXP APIs:

```
//JAXP APIs
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;

//parsing exceptions
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
//file reading
import java.io.File;
import java.io.IOException;

//W3C interface (document definition)
import org.w3c.dom.Document;
import org.w3c.dom.DOMException;
import org.w3c.dom.*;

//utility classes import java.util.*;
```

The W3C packages are the standards set by the World Wide Web Consortium. Java implements these standards through interfaces that define the structure of an XML document. The SAX packages are used to handle the exceptions.

When XML is parsed, the strictness of the document definition forces the program to handle many exceptions. One of the main features of XML is guaranteeing the validity of a document; this is why exception handling plays a bigger role in XML than in ordinary Java programming.

Here are the changes to the RPC server:

```
public class RPCdom
{
    static Document document;
    public static void main(String args[])
    {
        if (args.length != 1)
        {
            System.err.println("Usage: java DomEcho filename");
            System.exit(1);
        }
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
            //factory.setValidating(true);
            //factory.setNamespaceAware(true);
```

Now we need to create a **Document** object from the XML document so we can walk through its tree. The **Document** object allows the program to handle a variety of input sources, including **InputStream** objects, files, URLs, and SAX **InputSources**. Here's the code that creates the **Document** object:

```
try
{
    String xmlFile = args[0];
    DocumentBuilder builder = factory.newDocumentBuilder();
    document = builder.parse( new File(xmlFile) );
```

Now, the **document** variable contains an object that represents the entire XML document. After the call to **builder.parse**, the **document** object is completely populated and ready for analysis.

The first step is to locate the requested mathematical operation. To do that, the program creates an **Element** object by calling **getDocumentElement**—this corresponds to the document's root tag. You call **getElementsByTagName** to retrieve a list of the elements with a specific name (in this case, the list has only one entry). Here's the code:

```
Element element = document.getDocumentElement();
//could use getChildNodes() and loop through them all
NodeList operationNode = element.getElementsByTagName("method");
```

```
String operation =
                operationNode.item(0).getFirstChild().
                  getNodeValue  ();
```

The order of the items in the **NodeList** is the same as their order in the document. You might think the value of the method node would contain the operation (for example, addition). However, the actual text will be a child node of the method node. That's why the code has to call **getFirstChild** after locating the method node. The **getNodeValue** call retrieves the actual text.

Now we can move forward and get the list of numbers. This requires another call to the **getElementsByTagName** method, like this:

```
NodeList nl =
            element.getElementsByTagName("argument");
int nodeCount = nl.getLength();
Node node;
List parameter = new ArrayList();
```

Because there can be any number of these nodes, the code loops through the list, like this:

```
for (int i=0 ; i<nodeCount ; i++)
{
      node = nl.item(i);

       //argument
      String nodeName = node.getNodeName();
      String nodeTypeName = "unidentified";

       //TEXT_NODE
       short type = node.getFirstChild().getNodeType();
       NodeType nodeType = new NodeType();
      nodeTypeName = nodeType.getTypeName(type);
       String nodeValue = node.getFirstChild().getNodeValue();
       parameter.add(nodeValue);
     System.out.print(nodeName + ": " + nodeTypeName );
      System.out.println(" = " + nodeValue);
}
```

The highlighted line shows a helper class (**NodeType**) that is not a part of Java XML. This class illustrates how you can expand the JAXP yourself. In particular, it isn't easy to get the type name of the node. This class converts the integer representation of a node type into the appropriate string:

```
class NodeType
{
    final short ELEMENT_NODE                = 1;
    final short ATTRIBUTE_NODE              = 2;
    final short TEXT_NODE                   = 3;
    final short CDATA_SECTION_NODE          = 4;
    final short ENTITY_REFERENCE_NODE       = 5;
    final short ENTITY_NODE                 = 6;
    final short PROCESSING_INSTRUCTION_NODE = 7;
    final short COMMENT_NODE                = 8;
    final short DOCUMENT_NODE               = 9;
    final short DOCUMENT_TYPE_NODE          = 10;
    final short DOCUMENT_FRAGMENT_NODE      = 11;
    final short NOTATION_NODE               = 12;
    public String typeName;

    public String getTypeName(short type)
    {
        switch (type)
        {
            case ELEMENT_NODE:
                this.typeName = "ELEMENT_NODE";
                break;
            case ATTRIBUTE_NODE:
                this.typeName = "ATTRIBUTE_NODE";
                break;
            case TEXT_NODE:
                this.typeName = "TEXT_NODE";
                break;
            case CDATA_SECTION_NODE:
                this.typeName = "CDATA_SECTION_NODE";
                break;
```

TEAMFLY

Team-Fly®

```
                case ENTITY_REFERENCE_NODE:
                    this.typeName = "ENTITY_REFERENCE_NODE";
                    break;
                case ENTITY_NODE:
                    this.typeName = "ENTITY_NODE";
                    break;
                case PROCESSING_INSTRUCTION_NODE:
                    this.typeName = "PROCESSING_INSTRUCTION_NODE";
                    break;
                case COMMENT_NODE:
                    this.typeName = "COMMENT_NODE";
                    break;
                case DOCUMENT_NODE:
                    this.typeName = "DOCUMENT_NODE";
                    break;
                case DOCUMENT_ TYPE_NODE:
                    this.typeName = "";
                    break;
                case DOCUMENT_FRAGMENT_NODE:
                    this.typeName = "DOCUMENT_FRAGMENT_NODE";
                    break;
                case NOTATION_NODE:
                    this.typeName = "NOTATION_NODE";
                    break;
                default:
                    this.typeName = "UNKNOWN_NODE";
                    break;
            }
            return this.typeName;
        }
    }
```

The last section of the progr am is calling on the **mathHelper** class (as was done in the "Building an XML Parser to Serve as an RPC Engine" section), like this:

```
mathHelper mathhelp = new mathHelper();
```

```
String result = mathhelp.math(operation, parameter);
System.out.println(result);
```

Many errors (such as **SAXException** and **ParserConfiguration-Exception**) can occur during parsing. The error handling in the XML classes is probably the most extensive of any area in Java, considering the size of packages. Here's a transcript of running the program against a test file:

```
$ java RPCdom files/rpc.xml
argument: TEXT_NODE = 5
argument: TEXT_NODE = 97
argument: TEXT_NODE = 30
argument: TEXT_NODE = 221
argument: TEXT_NODE = 97
argument: TEXT_NODE = 4813
5263.0
```

The entire listing is available on this book's Web site (http://www.inforobo.com/javacorelanguage), so don't worry about trying to piece together a fully functioning program from this section's discussion.

What about changing the XML document programmatically? You can add, remove, or change elements quite easily. The following code can be added to the RPCdom program:

```
//this is how you can add an element and text node with it
Element newArgument = document.createElement("argument");
newArgument.appendChild(document.createTextNode("10000"));
element.appendChild(newArgument);
```

Warning Be careful about changing the XML tree in memory. The tree is not saved; it's temporary. If you want to make the changes permanent, then you must overwrite the original file (or save the changed tree to another file).

If you do include the code that adds an element, you will see different output, like so:

```
argument: TEXT_NODE = 5
argument: TEXT_NODE = 97
argument: TEXT_NODE = 30
argument: TEXT_NODE = 221
argument: TEXT_NODE = 97
argument: TEXT_NODE = 4813
```

argument: TEXT_NODE = 10000

15263.0

Tip For a quick list of methods that represent the complete Java Language [Java] binding for the Level 2 Document Object Model Core, refer to http://www.w3.org/TR/DOM-Level-2-Core/java-binding.html.

Chapter 10: Advanced Techniques

In Brief

Java is such a big language that you might never have to use whole areas of it. If you plan to build professional applications, however, you'll undoubtedly need to use several techniques. In particular, a competitive professional program requires several things:

- Speedy execution
- Robust error handling
- The ability to accept data from older versions of the same program

Java has special features to help you accomplish these goals. For example, multithreading can make your program seem faster (or at least more responsive). Profiling tools can help you plan ways to speed up your program.

In this chapter, you'll investigate two major techniques that real-world programs often use:

- Advanced serialization features can help you handle data even if it originated in a different version of your program that stores data differently.
- Multithreading and profiling attack performance issues from two directions. A program that uses multiple threads can perform multiple tasks at once (at least, the user thinks the tasks are executed concurrently). Profiling allows you to identify slow areas in your program that would benefit from some extra work to make them faster or more efficient. You'll see several techniques that can speed up sluggish programs.

In addition, the use of custom and advanced exception handling can make error handling more robust and consistent in a program. If you've ever had a program mysteriously crash and wipe out the document you were using, you'll know how important exception handling can be to users. If you want to read more about it, read Chapter 3.

Installation Programs

Professional applications usually have one other important feature: an installation program. It is possible to write an installer completely in Java. To run this installer, however, the users must already have a Java Virtual Machine (JVM) installed on their computer. The best installer will load Java if necessary, but it isn't possible to write an installer like that in pure Java.

Fortunately, a variety of commercial installers handles this problem, and you are just as well off if you use these. One that we've used often—and that is available in a free version—is ZeroG's Install Anywhere (http://www.zerog.com). This program creates native executables for many platforms; these executables will install the Java runtime, if necessary, plus your program. Install Anywhere also

creates platform-specific program launching devices (such as Start Menu icons for Windows, for example).

Using Object Serialization

How can you tell objects apart? Objects of the same type have the same methods but have different data in their fields (sometimes called their *attributes*). Objects of different types, of course, have different methods and fields. So by examining the methods and fields of an object, you can identify the type of the object. The actual data in the fields distinguishes the objects themselves.
To save an object to a file (or other persistent storage), you only need to store the object's type and the actual unique data in the object's fields. To read the object back, you simply create an object of the appropriate type and restore its previous state by storing the saved values in the new object's fields. Java supports this process by way of a mechanism known as *serialization*.

Serialization has many possible uses. If you need to save and load a file (like a word-processing document), you can do so by serializing your program's objects and then restoring them later. Serialization can allow two network programs (for example, a multimedia conference system) to communicate in an object-oriented way. Java takes care of differences, such as byte ordering and other nuances, so you can send data between a Windows program and a Unix program, for example. The base support for serialization is in the **Object** class. However, not all objects support serialization. Any object that allows serialization implements the **Serializable** interface. If your class derives from a class that implements **Serializable** (for example, most of the Java standard classes), your class is automatically serializable.

Serializable is a special interface because it doesn't contain any members. Any class that declares the **Serializable** interface automatically implements it. Then the **writeObject** method writes out the class state to an **ObjectOutputStream**. Each object that supports serialization can provide a **writeObject** method to customize how Java serializes it. If that method is absent, Java provides a default (the **defaultWriteObject** method in the **ObjectOutputStream** class).

Here's a code fragment that serializes a string:

```
FileOutputStream fos = new FileOutputStream("test.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
String s = "Coriolis Group";
oos.writeObject(s);
```

Here's some code that deserializes the string:

```
String previousString;
FileInputStream fis = new FileInputStream("test.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
previousString = (String)ois.readObject();
```

The variable **previousString** now has the correct contents. You can serialize all the primitives, arrays, native objects (such as dates), and even your own classes.

What can be serialized? Most objects, but not all of them. The **Object** class itself isn't automatically serializable. If you create a class that you want to serialize, you must implement the **Serializable** interface. You can also explicitly prevent fields from being serialized with the transient modifier, like this:

public transient int colorFlag;

If you add or remove class members, you won't be able to deserialize objects that were saved before the changes were made. However, the Java code is smart enough to handle added and deleted members — almost. Part of the overhead that Java stores with each serialized class is a magic number. This number is really a **static long** named **serialVersionUID** . If you don't provide your own magic number, Java computes the number by examining the class fields. When you change the class, the computation generates a new value, and the serialization code stops executing.

If you include your own arbitrary magic number, however, Java will always attempt to load the object, even if the object's definition has changed (as long as the magic number remains the same). However, major changes (for example, a change in base class) can still cause an error. Of course, if you add members that are essential to your program's operation, you might not be able to work with the old object anyway.

Understanding Multithreading

Java supports multiple threads of execution per program. When your program has multiple threads, it can appear to do multiple things at once. Each thread is like a separate program, but it shares many resources (variables, for example) with other threads in the same program. Java parcels out time slices to each thread so that they appear to execute simultaneously. Computers that have multiple CPUs can even execute several threads concurrently.

At first glance, it might seem that this would do little to improve performance. However, your threads are often waiting for something (a network connection, for example) while the other threads can continue executing.

Each thread is—not surprisingly—represented by a Java object. The object must either extend the **Thread** object or implement the **Runnable** interface. In either case, you'll provide a **run** method that implements the programming logic you want that thread to execute.

If you derive your object from **Thread**, you can call the object's **start** method to begin execution. If you don't extend **Thread**, you'll have to pass an instance of your object to the constructor of a **Thread** object. Either way, you call **start**. A common mistake is to call **run** directly. Don't call **run** yourself. If you do, you'll simply execute the thread's code in the current thread. You won't create a new thread, as you intended.

When you have multiple threads accessing the same variables, it is crucial that multiple threads don't try to alter variables at the same time. You can declare the field in question as **private** and access it only through methods. Then you can declare each method as **synchronized**.

This technique works because each object has a lock associated with it. When any thread attempts to access a synchronized member, the thread must first acquire the associated object's lock. Once one thread owns the lock, no other thread can acquire the lock until the owning thread releases it (which happens automatically when the synchronous operation is completed). You can even make static members synchronized. Then the lock affects all objects of a given class, not just one particular instance.
Sometimes you don't want to lock an object for the entire duration of a method call. For these cases, you can explicitly synchronize on an object. For example, suppose you want exclusive use of an array named **myList**. Consider this code:

```
synchronized (myList) {
  mylist[10]=0;
  }
```

Because every object extends **Object**, all objects have **wait** and **notify** methods. When you call some object's **wait** method from within a thread, the thread goes into an efficient wait state. It will not wake up again until some other thread calls the object's **notify** method. This allows one thread to wait for another thread to complete an operation. Of course, both calls must occur in synchronized methods (or at least within a synchronized code block) to ensure that threads don't conflict with each other. Calling **wait** automatically releases the lock on the object (otherwise, no other thread could ever call **notify**). You can optionally provide a timeout to **wait** so that the thread will not stop indefinitely.
If more than one thread is waiting on an object, calling **notify** will release one thread (and you can't predict which one it is). However, you can call **notifyAll** to release all threads waiting.

By default, your program will continue to execute until all threads are complete. However, sometimes you want a thread that executes until the reset of the program is done. In that case, you can call the **Thread** object's **setDaemon** method with a **true** argument. This allows the program to exit even if this thread is still executing.

Programming for Speed

There's an old saying that you can never be too rich or too thin. For computer programs, it might be better to say that you can't be too fast or too small. Over the years, many people have tried to speed up programs by using a variety of techniques collectively known as *optimization*. Even Java is not immune to these attempts. Vendors have improved the Java Virtual Machine (JVM) and provided just in time (JIT) compilation.

If you analyze programs in any language, you'll find that most of the execution time occurs in just a small fraction of the code. For example, suppose you print a report that has a header line and 100 data items on each page. For a 10-page report, you'll print 10 header lines and 1,000 lines of data. Obviously, if you want to tune the program's speed, you are better off concentrating on the code that

prints the data instead of the code that prints the header. Before you try to optimize code, ask yourself what would happen if the code you are looking at simply disappeared. If it wouldn't make much difference, then why are you optimizing it?

Problems with Optimization

Java programs are more difficult to optimize than other types of programs because of Java's "write once, run everywhere" philosophy. Java programs don't execute directly on the user's computer. Instead, the computer's JVM executes the bytecode. That means the JVM determines, to a large extent, how fast or slow your code will run.

Even on the same operating system, you might find different JVMs. Some JVMs interpret code one byte at a time. Others first compile the code (at least partially) into machine language, spending extra time up front for potentially faster execution.

So is it worth your time to try to optimize Java programs? Yes. It just requires a little extra attention.

Compiler Optimization

One way to get some degree of optimization for free is to enable optimization during compilation. Exactly how much optimization occurs depends on your compiler. For the standard JDK compiler (javac), you can enable optimization with the **-o** switch. This causes the compiler to calculate some expressions at compile time, and it eliminates some unnecessary code in your output classes. Other compilers, of course, may differ. IBM offers the Jikes compiler, for example. It has an **-o** option that has nothing to do with optimization (instead, it suppresses certain outputs).

Using compiler-based optimization is a great idea for released code, but the compiler can make only very simple improvements to your programs. The optimization methods that really pay off are those that enhance your program's efficiency. Compilers can't do that. Instead, you'll have to study your program and make intelligent choices about algorithms and coding practices.

Optimization Strategies

Optimization isn't so much a science as it is an art. Instead of hard rules, most optimization strategies are really heuristics. Some optimization heuristics are universal and apply to most programs, including Java code.

For example, a common observation is that most programs spend 90 percent of their time executing 10 percent of their code. The first rule of optimization is to speed up that 10 percent. If a line of code in your program normally requires 10 milliseconds to execute and it runs only once, your maximum possible improvement to that line of code is losing 10 milliseconds (if you can eliminate it).

Consider the same line of code in a loop that executes a million times. Now eliminating that part of your code saves 10,000 seconds. Even cutting the time in half will save 5,000 seconds (that's more

than 1 hour and 20 minutes). This is an extreme case, of course, but it does illustrate how a simple improvement can make a big difference.

How can you know which parts of your code are executing more frequently than others? Good question. In some obvious cases, simple inspection will do the trick. The problem is what to do about the more subtle cases. Many development tools provide some sort of profiling that can tell you where your code spends most of its time. Even the standard Sun SDK has a crude form of profiling.

Generating a Profile

To generate a profile for your applet, you need to start the JVM with the **-prof** option. If you want to test an applet named in the xyz.html file, you might run this command:

```
java -prof sun.applet.AppletViewer xyz.html
```

This will generate a file (java.prof) that tells you quite a bit about your program's execution. Too bad it's about as easy to read as ancient Sanskrit. Several freeware tools on the Internet will help you display the file in a friendlier format.

However you read it, the profiler data can help you decide where you should focus your optimization efforts. You can also add **println** statements at key points in your code to trace your program's execution to determine how many times a particular piece of code executes.

Another way to instrument your code is to time particular operations to determine how much time they actually consume. Consider several things to ensure accurate results:

- Other programs can interfere with your timing; try to run the subject program with as few other programs running as possible.
- If the JVM initiates garbage collection during a test, the results will be inaccurate. To minimize the possibility of an unexpected collection, force garbage collection by calling **System.gc** before starting a test.
- Although Java provides the time down to the millisecond (via either the **Date** class or the **System.currentTimeMillis** method), your computer might not provide the time to this level of accuracy.
- You should repeat the operation many times to provide a meaningful measurement period and to average out unusually fast or slow operations.
- Of course, timing tests shouldn't require user input because it is wildly variable.

Don't forget: A code sequence that runs fast on one platform (or JVM) might run slowly on another. Be aware that not all optimizations will be effective on all platforms.

What to Do?

Once you know which parts of your code need optimization, what should you do? First, think about the algorithm that part of the code employs. For example, if you're searching an array for data,

perhaps you could use a binary search algorithm or an indexed search. These are the hard optimizations because they usually require you to tear out a portion of your code and rewrite it.

In extreme cases, you might have to change significant portions of your program. For example, you might decide to use a binary search algorithm, but that requires a sorted list. So not only does the search code change, but the code that inserts entries into the array changes also.

What about simpler optimizations? It is possible to alter small bits of your code to realize performance benefits. Most of these types of changes are either common sense or techniques that are well known (in particular, many C compilers are very good at applying optimization rules to your C programs).

Know Your Library

Often the built-in library will perform functions faster than you can by using code. Perhaps the library uses a native method in C or assembly language. Perhaps the library author simply spent more time optimizing code than you're willing to spend. As an example, consider this code fragment:

```
for (i=0;i<100;i++) a[i]=b[i];
```

You can't get much simpler than this, right? Not exactly. The Java runtime library has a routine, **System.arraycopy()**, that does this operation as quickly as possible. You specify a source array, a starting position, a destination array, the destination's starting position, and the length. So you could rewrite the previous code to read:

```
System.arraycopy(b,0,a,0,100);
```

Is this faster? There's only one way to know: Profile it. Your results will depend on your machine's speed, the JVM, and the operating system.

TEAMFLY

Number Tricks

If you remember a few rules when using numbers, they will help your program run at top speed. First, always use integers where possible. Most modern computers handle integers much faster than nonintegers. You should also try to compute common subexpressions. Consider this code fragment:

```
X=P*(discount/100);
Y=P*(1-discount/100);
```

A nonoptimizing compiler will calculate **discount/100** twice, even though the result of the first calculation will be the same as the second. You can save time if you perform the calculation once and reuse the result, like this:

```
T=discount/100;
X=P*T;
```


Team-Fly®

```
Y=P*(1-T);
```

Another common trick is to use strength reduction to replace multiplication and division with shift or add operations (because these are usually faster than multiplication and division). This is easiest when you're multiplying or dividing by a power of 2. For example:

```
X=Y*8; // 8 is 2 to the 3rd power
X=Y<<3; // Same result
X=Y/16; // 16 is 2 to the 4th power
X=Y>>4; // same result
```

For division, this is of no use unless you want to divide by a power of 2. However, multiplication can often be rewritten to use shifts even if it takes more than one shift:

```
X=Y*10;
X=Y*8 + Y*2; // same answer
X=Y<<3 + Y<<1; // same answer
```

You can also apply a similar technique to get rid of multiplication inside loops. For example, consider this loop:

```
for (i=0;i<10;i++) {
int z=i*10;
. . .
}
```

You could rewrite this as:

```
int z=0;
for (i=0;i<10;i++) {
. . .
z=z+10; }
```

Do all of these things really work? It depends. In informal testing, we found that shifting is usually faster than multiplying; however, for the JDK 1.1.7 JVM for Windows, the multiplications are still faster. For Internet Explorer 5, the shifts are faster. This just proves that you have to measure things instead of guessing.

Immediate Solutions

Serializing an Object

Listing 10.1 shows a program that creates an object (**someObject**), sets one attribute (**color**), and then serializes it.

Listing 10.1: Serializing objects.

```
import java.io.*;


public class RealSerialization {

   public static void main(String[] args)

    {

    String previousState;

   String filename = "object.ser";

   SomeObject dyingObject = new SomeObject();

   SomeObject resurrectedObject = new SomeObject();


    dyingObject.setColor("green");

    previousState = dyingObject.getColor();

   System.out.println("old object color state = " +

            previousState);


   FileOutputStream fileOutputStream = null;

   ObjectOutputStream objectOutputStream = null;


    try

    {
```

```
        fileOutputStream = new FileOutputStream(filename);
        objectOutputStream =
            new ObjectOutputStream(fileOutputStream);
        objectOutputStream.writeObject(dyingObject);
        objectOutputStream.writeObject("Little Black Book");
        objectOutputStream.flush();
        fileOutputStream.close();
    } catch(IOException ex)
    {
       ex.printStackTrace();
    }

// I'm available for the garbage collector
        dyingObject = null;
        System.out.println(
           "old object is dead but state preserved");
        System.out.println();

        FileInputStream fileInputStream = null;
        ObjectInputStream objectInputStream = null;

        try
        {
           fileInputStream = new FileInputStream(filename);
            objectInputStream =
```

```
            new ObjectInputStream(fileInputStream);
          resurrectedObject =
            (SomeObject)objectInputStream.readObject();
          previousState =
            (String)objectInputStream.readObject();
          fileInputStream.close();

          String newState = resurrectedObject.getColor();
          System.out.println("new object color state = "
            + newState);
          System.out.println(previousState);
          System.out.println("old object has been resurrected");
      } catch (Exception e)
       {
          System.out.println("Problem serializing: " + e);
       }
   }
}

class SomeObject implements Serializable
{
   private String color;

   public void setColor (String color)
   {
```

```
        this.color = color;
    }


    public String getColor ()
    {
        return this.color;
    }
}
```

The first two highlighted lines are where Java writes the two objects (instance of **SomeObject** and **String**) to the file (object.ser). The second set of highlighted lines is where Java reads the file and assigns the values it finds there to the object, or to objects if you serialized several to the same file. If you look at the object.ser file, you'll find only four things you can read—namely **SomeObject**, **color**, **green**, and **Little Black Book**. There are a few overhead bytes before each of these pieces of text. These bytes encode information that Java needs in order to reconstitute the object correctly.

Customizing Serialization

If you provide your own **objectWrite** method, you can customize how serialization works. For example, you might want to encrypt or compress data before saving it to disk. Of course, you'd also have to provide a custom **objectRead** method to deserialize. The following code is what you would add to the **SomeObject** class shown previously:

```
    private void writeObject(ObjectOutputStream stream)
            throws IOException
    {
        System.out.println("object color=" + this.color);
    // This code stores colors after adding "ish" so
    // green becomes greenish.
        this.color = this.color + "ish";
        stream.defaultWriteObject();
        System.out.println("serialized color=" + this.color);
    // to reconstitute the color we have to strip the "ish"
            suffix from it.
        this.color = this.color.substring(0,this.color.length()-3);
        System.out.println("object color=" + this.color);
```

```
    }

    private void readObject(ObjectInputStream stream)
            throws IOException, ClassNotFoundException
    {
       stream.defaultReadObject();
      System.out.println("deserialized color=" + this.color);
// When reading, strip the "ish" suffix
       this.color = this.color.substring(0,this.color.length()-3);
      System.out.println("object color=" + this.color);
    }
```

Adding the previous code to Listing 10.1 would result in the following:

```
old object color state = green
object color=green
serialized color=greenish
object color=green
old object is dead but its state preserved

deserialized color=greenish
object color=green
new object color state = green
Little Black Book
old object has been resurrected
```

Creating a Thread

To make a thread, you subclass **Thread** and override its empty **run** method. The **run** method does whatever task you want the thread to perform. Listing 10.2 shows you how to create the subclass. Notice that to start the thread, you create an instance of the object and call **start**.

Listing 10.2: Subclassing the Thread object.

```
public class TestThreads
{
   public static void main (String[] args)
    {
```

```
        new ColorThread("Green").start();
        new ColorThread("Red").start();
        new ColorThread("Purplish").start();
    }
}

class ColorThread extends Thread
{
    private String color;
    public ColorThread(String color)
    {
        this.color = color;
    }
// Each thread executes this code and
// terminates when the loop completes
    public void run()
    {
        for (int index = 0; index < 10; index++)
        {
            System.out.println(this.color);
            try {
// without this line, a fast machine
// will finish each thread before
// the next gets to execute
                Thread.sleep(1);
```

```
            }
        catch (InterruptedException e) { }
    }
  }
}
```

In some cases, you don't want to derive your class from **Thread**. Remember, in Java your class can only extend one base class. So if your class already has a base class, it can't also extend **Thread**. Fortunately, you can make any object implement the **Runnable** interface (which has only one method: **run**). Then you can pass your object to the **Thread** object's constructor to obtain a thread. After that, everything is just like the example in Listing 10.2 .

Suppose that the **ColorThread** class started like this:

```
class ColorThread implements Runnable
{
```

The **main** method would change to look like this:

```
    public static void main (String[] args)
     {
       new Thread(new ColorThread("Green")).start();
       new Thread(new ColorThread("Red")).start();
       new Thread(new ColorThread("Blue")).start();
     }
```

Related solution:	Found on page:
Understanding Inheritance	108

Making a Thread Wait for Another Thread

You place the method call in the thread that you want to have wait. You prefix the method with the thread that you want to wait for, like so:

```
ThreadB.join(1000);
```

Suppose this statement was placed in one thread (**ThreadA**). Upon executing this statement, **ThreadA** will pause until **ThreadB** dies. The **1000** is the number of milliseconds to wait. **ThreadA** will resume when **ThreadB** dies or in 1 second (1,000 milliseconds), whichever occurs sooner.
You can force one thread to step aside for another with the **yield** method, which allows other threads to execute. You can also put your thread to sleep for a number of milliseconds by using **sleep**. Listing 10.3 shows several operations involving threads.

Listing 10.3: Using the Thread object's methods.

```
public class TestThreads
{
    public static void main (String[] args)
    {
// Create three new threads
        Thread green = new ColorThread("Green");
        Thread red = new ColorThread("Red");
        Thread purple = new ColorThread("Purplish");

        green.setName("Green Thread"); //set name
         red.setName("Red Thread");
         purple.setName("Purplish Thread");

        //green.start(); //start threads
         //red.start();
         //purple.start();

         boolean true_false;
        int count, limit;
         String string;

         Thread current;
         Thread[] allThreads;
        ThreadGroup group;
```

```
        group = green.getThreadGroup();
// currently executing thread
        current = Thread.currentThread();
        System.out.println( current.getName() );
        count = Thread.activeCount();
        System.out.println( count ); // how many active threads?
        allThreads = new Thread[count];
//copies threads to variable
        limit = Thread.enumerate(allThreads);
        for (count = 0; count < limit; count++) {
            string = allThreads[count].getName();
            System.out.println( string );
        }
// adjust priority
        red.setPriority(red.MAX_PRIORITY);
        purple.setPriority(red.NORM_PRIORITY);
        green.setPriority(red.MIN_PRIORITY);
        System.out.println( red.getPriority() );

// set Daemon status
        purple.setDaemon(true);
        System.out.println( purple.isDaemon() );
        System.out.println( red.isDaemon() );
```

TEAMFLY

Team-Fly®

```
        red.interrupt();
        System.out.println( red.interrupted() );
        System.out.println( red.isInterrupted() );
        System.out.println( red.isAlive() );

// Try out yield, sleep, and join
        Thread.yield();
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {
        }

        try {
            green.join(1000);
        } catch(InterruptedException e) {
            Thread.dumpStack();
        }

        try {
            green.destroy();
        } catch(NoSuchMethodError e)
        {
        }
    }
}
```

```
//returns:

//main

//4

//main

//10

//true

//false

//false

//false

//false
```

Notice that the highlighted **start** statements have been commented out. If you don't do this in this example, the threads will complete and exit before the rest of the code can execute. You can fix this by increasing the time specified in the call to **sleep** in the **Color** class's **run** method.

You can also make one thread wait for another thread to signal it; you do this by using **wait** and **notify**. When a thread calls an object's **wait** method, the thread goes into an efficient wait state. The thread will not wake up again until some other thread calls the same object's **notify** method. Both calls must occur in a synchronized block to ensure that the threads don't conflict with each other. You can optionally provide a timeout to **wait** so that the thread will not stop indefinitely.

If more than one thread is waiting for an object, calling **notify** will release only one thread (and you can't predict which one it is). However, you can call **notifyAll** to release all threads waiting.

Optimizing Objects

Although object orientation is a powerful part of Java, it's also an element where efficiency can suffer if you aren't careful. Creating new objects is expensive. When possible, avoid calling **new** if you can reuse an existing object instead. This is especially true inside loops. If you can create one object outside of a loop and reuse it, your code will be faster than similar code that creates a new object during each loop iteration.

Another expensive operation is calling member functions that might have overrides in a derived class. Even if no class extends the current class, Java has to search for other functions on each call (this is similar to using virtual methods in C++). However, if your function is **private** or **final**, or if the class you're using is **final**, Java can directly call the function because no other class can override the function. Because of this optimization, it is best to declare everything **private** or **final** when possible.

Related solution:	Found on page:
Using Access Modifiers on a Field	92

Optimizing String Handling

Because strings are immutable, it is a bad practice to do the following:

String test = "The speed to do this is fair.";

test += "This, however, is horrendously slow!!";

test += "This is just as bad as the previous line!";

test += "Still no better!";

Each **test +=** causes Java to create a new intermediate **String** object. Suppose that this object is named **X**. When you write:

test += "This, however, is horrendously slow!!";

the compiler generates code equivalent to:

X = test + "This, however, is horrendously slow!!";

test = X;

The code works, but it isn't very efficient because you have to create (and destroy) a temporary object. If you want to make many changes in a string, you should consider using the **StringBuffer** class. For example:

StringBuffer testbuffer;

String test;

testbuffer.append("The speed to do this is fair.");

testbuffer.append("This is fine, too. ");

testbuffer.append("This is just as fast as the previous line!)";

testbuffer.append("Still no worse!");

test=testbuffer.toString();

How does the **StringBuffer** perform an **append**? The **StringBuffer** is essentially a character array wrapped with helpful methods. Every time you call the **append** method, the **StringBuffer** code compares the size of the proposed new string with the room available at the end of the array. If there is room, the code just copies the new string to the free space. If there isn't room, a new character array is created that is larger than the old buffer. The code moves the contents of the old array to the new array and then appends the string. Of course, in this case, the call still creates and destroys arrays, but many times the call simply uses the available space.

The improved speed and memory efficiency are very important when string manipulations are performed inside a loop. However, **String-Buffer** could be faster. One thing you can do to help

StringBuffer is to try to anticipate how much space you'll need and then initialize the buffer appropriately. Remember that if you run out of space Java will have to allocate and deallocate an array, and that's what you want to avoid. Here is a buffer set to 1,024 characters:

```
StringBuffer buffer = new StringBuffer(1024);
```

One way to speed up the **StringBuffer** class is to write your own **append** method like this:

```
public class BufferAppend
{
   private char buf[] = null;
   private int limit = 0;

   public BufferAppend(int size)
    {
       buf = new char[size];
    }

   public void append(char c)
    {
      if (limit == buf.length)
       {
            char tempChar[] = new char[limit * 2];
          System.arraycopy(buf, 0, tempChar, 0, limit);
           buf = tempChar;
       }

     buf[limit++] = c;

   }
}
```

This custom **append** method runs twice as fast as the native **append** method because the custom method checks the array size locally rather than by calling a method as the Sun library does. Of course, you will gain a lot of speed if you skip the array size check altogether (the highlighted code). Then you have to be careful not to overrun the buffer.

The next best thing you can do to increase the **StringBuffer** speed is to replace Sun's **synchronized** methods with your own unsynchronized ones (assuming that you don't need thread safety). You can't just extend **StringBuffer**, however, because it is a **final** class.

Instead, you can write your own version of the class with just a few methods that you will call the most often. The two things you want to do with your version are to exclude **synchronized** methods and to avoid the capacity check when appending. Calling a **synchronized** method takes several times as long to complete as does calling an equivalent method that is not synchronized. You can copy the method code from **StringBuffer** (you can download the source to the Java library from Sun's Web site) and paste it into your class; then remove the **synchronized** modifier to gain a lot of speed.
One last note on processing strings faster: Avoid using the **charAt** method. You pay for the method invocation overhead and the boundary check performed every time you call this method. We recommend converting the string to a character array and marching through the array by using normal indexing.

Optimizing Loops

Loops are an easy target for optimization because they are obviously repeating statements. The common way to write **for** loops is like this:

```
for (int index=0; index<limit; index++)
{
    // statements;
}
```

For most JVMs, however, comparing an **int** to a zero is very fast, so the loop will execute faster if you rewrite it like this:

```
for (int index=limit-1; index>0; index--) {
    // statements;
}
```

Often, a **for** loop is used to iterate through an array. Java has built-in boundary checking, which prevents you from going past the ends of the array. You can take advantage of exception handling to skip the test, like this:

```
try
{
    for (index=array.length; /*skip this test*/ ; --index)
    {
      array[index] = expression;
    }
}
catch (ArrayIndexOutOfBoundsException e) { ; }
```

Of course, throwing an exception has lots of overhead, so for a few iterations this is probably not worth the effort. But if you are working with large arrays, this technique might be more efficient. To be sure, you'll have to profile on the JVM you are interested in.

Tip Use local variables in loops whenever possible. If you don't, the JVM has to do extra work to get the reference. It is often useful to assign an object's fields to a local variable before entering a loop so that the loop can work with the local copy.

Method calling involves overhead, particularly in a loop. You have many opportunities to speed things up by writing your own code inline rather than calling a method. For example:

```
k = p.max(i, j); k = (i > j ? i : j);
```

The second line is much faster.

Optimizing I/O

If you are reading files, you should adopt a definite strategy. You want to read as many bytes or characters at a time as possible. Avoid reading one byte at a time. You can use Java classes (like **BufferedReader**)

that will read multiple characters and dole them out to you more slowly, if you like. The idea is to avoid asking the hardware for more data when possible. The following code demonstrates the difference between reading one byte at a time (**DataInputStream**) and reading many bytes at a time (**BufferedReader**):

```
import java.io.*;
public class SlowFileReader
{
    public static void main(String args[])
    {
      //single byte file reading
      int lineCount = 0;
      long startTime = System.currentTimeMillis();
       for (int index = 0; index < 1000; index++)
       {
         try
         {
        byte c;
          FileInputStream fis =
             new FileInputStream("SlowFileReader.java");
        DataInputStream dis =
             new DataInputStream(fis);
             while (true)
```

```
            {
               c=dis.readByte();
              if (c=='\n') lineCount++;
     }
   } catch (Throwable e) // will catch EOF
    {
  }
}
    elapsedTime(startTime, 1000*lineCount, "DataInputStream");
    //buffered file reading
    startTime = System.currentTimeMillis();
    lineCount = 0;
   for (int index = 0; index < 1000; index++)
    {
    try

    {
    FileInputStream fis =
       new FileInputStream("files \\genesis_1.txt");
    BufferedReader buffReader
      = new BufferedReader(new InputStreamReader(fis));
    while (buffReader.readLine() != null)
    {
     lineCount++;
    }
     fis.close();
    } catch (Throwable e)
    {
       System.err.println("exception");
    }
    }
    elapsedTime(startTime, 1000*lineCount, "BufferedReader");
    }
   public static void elapsedTime(long start,
```

```
        int iterations,
        String event)
        {
      long elapse = System.currentTimeMillis() - start + 50;
      float seconds = (float)elapse / 1000f;
     String report = event + " (" + iterations + " lines) = " +
      seconds + " seconds";
        System.out.println(report);
        }
}//returns: //DataInputStream (62000000 lines) = 9.714 seconds //BufferedReader (62000000 lines) =
2.123 seconds
```

Chapter 11: Security

In Brief

Cryptographers (people who conjure up ever-more-difficult encrypting schemes) and cryptanalysts (people who crack codes) have played a game of leap-frog since before the time of Julius Caesar. Encrypted messages have taken many forms over the years, but generally the idea is simple: pass a message (plain text) through an algorithm to get a secret message (cipher text). Security and encryption have taken a leap forward in reliability and effectiveness since the advent of computers.

Java supports the encryption and decryption of data. For the purposes of this chapter, security has several goals:

- *Confidentiality*—By encrypting data, you ensure that only the intended recipient can read the data.
- *Authenticity*—Using a digital signature ensures that data is from the intended source.
- *Nonrepudiation*—A digital signature prevents a sender from denying his or her origination of the data.
- *Integrity*—By using a digest (described later), a recipient can verify that data was not modified since it was signed.

It is natural to think of these items in terms of email because that is where we encounter them most often. However, any data might require security. For example, when you visit a secure Web site, you need to know that the site really is the one it says it is. So even if I managed to hijack your bank's URL, you would know that it was not your bank's Web site because I can't sign data with the bank's secret key. For the purposes of this chapter, data that you want to encrypt or decrypt will be called a *message*. That terminology doesn't presuppose any particular way of transmitting that message. One thing that is central to securing data is a *message digest*. A digest is like a very long checksum. You perform calculations on a message and generate a fixed-length signature that represents it (similar to a hash code). While it is possible that two messages will have the same digest, it is very unlikely. Digests are most important when it comes to signing messages. Signing a message allows the recipient to verify the message originator and detect any tampering that might have occurred during the message's transmission.

Security Overview

Most modern security schemes rely on algorithms that in turn rely on keys. These keys are nothing more than unique numbers (perhaps very large numbers) that are used in an algorithm to scramble (or unscramble) the bits in a message.

Companies like RSA Security (one of the big players in the security business) publish these algorithms, so they are not secret (the algorithms might be copyrighted or patented, but they aren't secret). The entire security, then, is in selecting keys that make it difficult to reverse the algorithm.

One of the strengths of this approach is that the algorithms are subject to public review; any possible weakness will be quickly exploited (and fixed).

Your bank probably has better locks and alarms on its doors than you have on your house. Fort Knox, Kentucky, where the United States stores its gold, probably has even better locks. No lock is unbreakable, however, and that also applies to encryption methods. Given enough time and computer power, any code is breakable. However, when the cost and trouble of breaking the lock exceeds the cost of the contents, you can deem the lock safe enough.

In general, longer keys make for tougher security. For example, old Web browsers often used 56-bit keys. As computers have become faster, breaking keys of any length has become easier. Now, 128-bit keys are the standard for secure Web sites. As computers become even faster, longer keys will become necessary.

Exactly how you use and select a key depends on the algorithm you are using. For most modern algorithms, however, the key will have some relationship to a very large prime number. Also, nearly all modern methods use something known as *public key cryptography*. In this scheme, there are two related keys. One is secret and is known as the *private key*. The other key—the *public key*—can be freely distributed to anyone. This allows someone to encrypt something with your public key that only you can decode. The mathematics are such that knowing the public key doesn't help you crack the message, nor does it help you guess the private key.

Java's library has many routines you can use to generate and use keys without much trouble. The library doesn't actually perform these tasks; instead, it interfaces with a security provider. Sun's default implementations of the security routines (in **java.security**) can handle most common situations. However, it would be possible for sophisticated systems to use, for example, identity cards read with special hardware to assist in key generation or authentication.

Immediate Solutions

Creating a Message Digest

A message digest is similar to a hash code. The digest typically contains more bits than an ordinary hash code does, and a digest applies to a block of data. The digest algorithm should have these characteristics:

- The algorithm is one-way; you can compute a digest from the message, but you can't recover the message from the digest alone.
- Small changes in the message should result in large changes in the digest.
- The algorithm should be fast and efficient to compute.
- The same message must always generate the same digest.
- The digest should be short (on the order of 1KB or so).

Team-Fly®

- Ideally, no two messages should produce the same digest. In practice, this isn't possible, but you'd like to have a wide distribution of digests for different messages.

You can write your own digest algorithm. For example, you can calculate the sum of all the Unicode character values in a string, but this algorithm doesn't meet the previously stated goals very well. The immediately obvious problem is that you can swap characters around and get the same digest (in other words, the digest of AB is the same as the digest for BA). You can fix that problem by multiplying each character by its position expressed as a number. Even then, small changes in the text cause only small permutations in the digest. You can partially solve this problem by using the random-number generator. At each character (times position), you seed the generator, and you use the resultant random number as part of the digest.

Given the same seed, Java's random-number generator will produce the same number. For a set of inputs, the random-number generator produces numbers that cover the entire output range. In this context, random doesn't mean unpredictable; it means randomly distributed over the output range. Listing 11.1 shows a Java digest function. The **MessageDigestTest** class uses the built-in digest function known as the SHA-1 algorithm. The **MessageDigest.getInstance** call returns an object that implements the specified algorithm.

Listing 11.1: Creating a message digest.

```
import java.io.*;

import java.security.*;

import java.lang.StringBuffer;


public class MessageDigestTest{

   public static void main(String[] args)

    {

        try

        {

           FileInputStream fis =

             new FileInputStream(args[0]); // get input file

           BufferedInputStream bis =

             new BufferedInputStream(fis); // buffer it

// We will copy the input stream to a byte array using
```

```
// a ByteArrayOutputStream
            ByteArrayOutputStream baos =
               new ByteArrayOutputStream();


            int iByte;
// This loop does the copying
           while ((iByte = bis.read()) != -1)
            {
                baos.write(iByte);
            }
// Now get the byte array and voila
// the array contains the stream contents
// there are other ways this could have been done, of course
            byte[] buffer = baos.toByteArray();


// Get a MessageDigest object that will use SHA-1 algorithm
             MessageDigest algorithm =
                MessageDigest.getInstance("SHA-1");
           algorithm.reset(); // start fresh
// compute digest
             algorithm.update(buffer);
// get digest as a byte array
             byte[] digest = algorithm.digest();
// Now we will turn the output array into a string to print
            StringBuffer hexString = new StringBuffer();
```

```
            for (int i=0;i<digest.length;i++)
            {
                hexString.append(
                Integer.toHexString(0xFF & digest[i]));
                hexString.append(" ");
            }
// print it
            System.out.println(hexString.toString());

        }

        catch(java.io.FileNotFoundException e) { }
        catch(java.io.IOException e) { }
        catch(java.security.NoSuchAlgorithmException e) { }
    }
}
//java MessageDigestTest files\genesis_1.txt
//returns: //33 b3 69 32 fc f1 3 34 ff 4d 82 e2 a1 a2 25 46 43 60 74 22
```

I added only one space to the input file and ran the program again. This time it returned the following:

cd e3 f db 6e 11 eb 94 51 f5 94 8c e7 17 16 e7 4d 35 2f d1

One extra space resulted in a very different digest. The program converted my 800-word test file into a digest in the blink of an eye.

Digests are important when it comes to signing messages (covered later in this chapter). You'll occasionally hear digests referred to as *fingerprints* because they tend to identify a message uniquely. The example code uses the SHA-1 digest algorithm. This algorithm was developed by the National Institute of Technology (NIST) and generates a 160-bit key. Another popular algorithm, MD5 (from Professor Ronald Rivest), generates a 128-bit key and is therefore somewhat faster than SHA-1.

Of course, there is no completely unique way to represent a document of arbi trary length in 128 bits (or even 160 bits). It is possible that two messages will generate the same digest, but the construction of the algorithms makes this as unlikely as possible and also ensures that small changes to the document will result in large changes in the document. Therefore, documents that do generate the same digest should be very dissimilar. Changes made by an interloper, or garbling during transmission, are almost sure to produce a different digest.

Creating Secure Random Numbers

Many encryption algorithms require randomly distributed numbers. You can create random numbers easily with the code in Listing 11.2.

Listing 11.2: Creating secure random numbers.

```
import java.util.Random ;



public class RandomGenerator

{

   public static void main(String args[])

    {

       if (args.length != 2)

        {

          System.out.println("Usage: java RandomGenerator " +

                        "howManyRandomNumbers maximumNumber");

           System.exit(1);

        }

       int count = Integer.parseInt(args[0]);

       int maximumNumber = Integer.parseInt(args[1]);

      long seed = 1000L;

      Random rm = new Random(seed);



        for(int index = 0; index < count; index++)
```

```
        {
            int randInt = rm.nextInt(maximumNumber);
            System.out.print(randInt + " ");
        }
    }
}

//java RandomGenerator 20 100

//returns:

//87 35 76 24 92 49 41 45 64 50 79 59 72 83 36 75 46 2 23 41
```

Java's random-number generator calculates numbers that are randomly distributed between two extremes. The previous program prints numbers between zero and the maximum number provided at the command (100 in Listing 11.2). Because the example uses the same seed every time, you will get the same sequence of random numbers each time you run the program. If you do not provide a seed, the default constructor uses the system clock as the seed. Using the system clock provides results that are difficult for a human to guess but not hard for a computer to guess (this method is good for things like games, but not good for encryption).

If you require an unpredictable sequence (mandatory in security applications), you will need something better than Java's **Random** class. Java also provides the **SecureRandom** class, which is more suitable. In the previous program, you'd replace the **Random** object with a **SecureRandom** object, like this:

```
byte[] seed = "A secret!".getBytes();
SecureRandom rm = new SecureRandom(seed);
```

Notice that the seed is now a **byte** array rather than the integer used with the **Random** object. This small change produces the following result:

```
//java SecureRandomGenerator 10 100
//returns:
//50 8 39 12 49 31 41 67 70 66 17 12 88 3 21 47 13 7 92 11
```

This sequence repeats, given the same seed. What's the advantage of using the **SecureRandom** class instead of the plain **Random** class? The **SecureRandom** class has a default constructor that

doesn't use the system clock to get a seed. This class uses something that is system-dependent and, presumably, harder to guess.

Generating Security Keys

Keys are a crucial part of Java's security system. You generate keys in pairs: one private key and one public key that go together. The idea is that you can give away your public key, but you keep your private key a secret.

Listing 11.3 demonstrates how to generate the pair of public and private keys, separate the two keys, and decompose each key into its constituent parts. (Each key has several parts, each represented by a **BigInteger**.) The actual key is the **x** (private) or **y** (public) portion of the key. The **p**, **q**, and **g** parts are different parts of the algorithm used to select the keys (the prime, subprime, and base numbers). The exact format of the key depends on the algorithm you select. Although the **KeyPair** object appears to contain **PublicKey** and **PrivateKey** types, these are abstract classes. In reality, the actual type of the key will depend on the algorithm you select (in this case, the types are **DSAPublicKey** and **DSAPrivateKey**).

Listing 11.3: Generating security keys.

```
import java.security.interfaces.*;

import java.security.*;

import java.io.*;

import java.math.BigInteger;


public class KeyTest

{

   public static void main(String args[])

    {

        String algorithm = "DES";


       KeyPairGenerator kpg;

       BigInteger g;

       BigInteger q;
```

```
        BigInteger p;
        BigInteger y;
        BigInteger x;
        String keyDescription;
        DSAParams parameters;
        DSAPublicKey publicKey;
        DSAPrivateKey privateKey;

        try
        {   //generate the pair of keys using DSA – could use RSA
            kpg = KeyPairGenerator.getInstance("DSA");
// key length is 512, use random number to generate unique key
// this greatly decreases the likelihood of generating the same
// key twice.
            kpg.initialize (512, new SecureRandom());
           KeyPair keyPair = kpg.genKeyPair();

            //get the public key and print out its features
            publicKey = (DSAPublicKey)keyPair.getPublic();
            parameters = publicKey.getParams();
           g = parameters.getG();
           p = parameters.getP();
           q = parameters.getQ();
           y = publicKey.getY();
```

```
        System.out.println("Public key parameters:");
       System.out.println("prime p = " + p);
        System.out.println("subprime q = " + q);
        System.out.println("base g = " + g);
       System.out.println("public key = " + y);
        keyDescription = publicKey.toString();
        System.out.println("\n" + keyDescription);
        System.out.println("\n");


        //get the private key and print out its features
       privateKey = (DSAPrivateKey)keyPair.getPrivate();
        parameters =publicKey.getParams();
       g = parameters.getG();
       p = parameters.getP();
       q = parameters.getQ();
        x = privateKey.getX();


        System.out.println("Private key parameters:");
       System.out.println("prime p = " + p);
       System.out.println("subprime q = " + q);
        System.out.println("base g = " + g);
        System.out.println("p rivate key = " + x);
        keyDescription = privateKey.toString();
        System.out.println("\n" + keyDescription);
    } catch(NoSuchAlgorithmException e)
```

```
        {
            System.out.println(e);
        }
    }
}
```

You can supply these keys to objects that need them, such as a **Cipher** object (see the next Immediate Solution). The program returns the following results:

```
Public key parameters:
prime p = 13232376895198612407547930718757728267435527029623408872245156039757713029036368719146452186041204237350521785240337048752071462798273003935646236777459223
subprime q = 857371293708094202104259627990318636601332086981
base g = 542164405743647514160964848832570512804742839438047437683466730076610826261390054268128908071372459731067307411935513608579598209739067089036718514118979600
public key = 709755322934384292860220344338322677871686915943162452774679613291054916138665996355355706525369900527583173576153088704140117949052732751069388672401147 1
```

Sun DSA Public Key

Parameters:

p: fca682ce 8e12caba 110e546d 26eb2cf7 b078b05e decbcd1e b4a208f3 ae1617ae 01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17

q:

962eddcc bb2602e6 36acba8e b6a126d9 346e38c5

g:

678471b2 e21aa9c5 779c244e 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e 35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4

y:

87841e56 11ba4d82 a0e8c768 41f7768c 7d68e955 f065bb89 d6c0bebc

d3c33c97 010a14ab 24968692 3b59a401 561dd694 3ddd3648 e7a7f5c5

8372f697 1941f5cf

Private key parameters:

prime p = 132323768951986124075479307187577282674355270296234088

722451560397577130290363687191464521860412042373505217852403370 4

875207146279827300393564623677745922 3

subprime q = 857371293708094202104259627990318636601332086981

base g = 542164405743647514160964848832570512804742839438047437 6

834667300766108262613900542681289080713724597310673074119355136 0

857959820973906708903671851411897 96

private key = 331232879280749710607215113044779854072026248481

Sun DSA Private Key

parameters:

p:

fca682ce 8e12caba 110e546d 26eb2cf7 b078b05e decbcd1e b4a208f3

ae1617ae 01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151

bdc43ee7 37592e17

q:

962eddcc bb2602e6 36acba8e b6a126d9 346e38c5

g:

678471b2 e21aa9c5 779c244e 147db1a9 aaf244f0 5a434d64 86931d2d

14271b9e 35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a

6c416e50 be794ca4

x: 3a04fefa00731ceb7d461ab7766e8f4964943521

The program selects a length of 512. The output would be twice as long if you selected 1,024 (the only other choice in the Sun security implementation). Did you notice that the **p**, **q**, and **g** numbers are the same? In the default implementation, these numbers are based partly upon hardware (such as a network interface card's ID). For a full discussion of the math used with keys, please see RSA Security's Web site (http://www.rsa.com).

Team-Fly®

Encrypting and Decrypting a String

The two most frequently performed security actions are encrypting and decrypting messages. After you have computed keys, you use them to alter data. For example, suppose you have the letter A. The Unicode value for A is 0x0041 (65 decimal). To encrypt it, you need to convert this value to another value. The trick is to convert it in such a way that only the intended recipient can decode it. The code in Listing 11.4 encodes and decodes a string. Like the previous example, it generates a key. This example then uses an instance of the **Cipher** class to encode the text (taken as a byte array) into another byte array, which is encrypted using the key. The program then decrypts the string. Of course, you usually won't have the encryption and decryption steps for the same data executing so close together. Usually, you'll encrypt the data and then send it or store it somewhere. Later, you'll receive or recall the data (perhaps using a different program), and decrypt the data.

Listing 11.4: Encrypting and decrypting a string.

```
import java.security.*;

import javax.crypto.*;

import javax.crypto.spec.*;

import java.io.*;

import java.lang.*;


public class CipherTest

{

   public static void main(String args[])

    {

        String sourceString = "Wisdom is judged by her fruit.";

       if (args.length != 1)

        {

           System.out.println("Usage: java EncryptTest " +

                              "string");

       } else

        {
```

```
            sourceString = args[0];
        }
       System.out.println("Source: " + sourceString);


        try
        {
           String algorithm = "DES";
           String mode = "ECB";



           String padding = "PKCS5Padding";
          String transformation = algorithm + "/" +
                                          mode + "/" +
                                           padding;
          KeyGenerator kg = KeyGenerator.getInstance(algorithm);
           Key key = kg.generateKey();


           Cipher cipher = Cipher.getInstance(transformation);
            cipher.init(Cipher.ENCRYPT_MODE, key);
// Encrypt the byte array of data
           byte input[] = sourceString.getBytes();
           byte encrypted[] = cipher.doFinal(input);


           String flag = "";
            for(int i=0;i<input.length;i++)
```

```
            {
              flag = ((i+1) % 10==0)? '\n' : " ";
                System.out.print(Byte.toString(input[i] ) + flag);
            }

            System.out.println("\n");
            System.out.println("Encrypted: ");
            for(int i=0;i<encrypted.length;i++)
            {
             flag = ((i+1) % 10==0)? "\n" : " ";
              System.out.print(Byte.toString(encrypted[i]) + flag);
            }
            System.out.println("\n");
// Decrypt it
            cipher.init(Cipher.DECRYPT_MODE, key);
           byte output[] = cipher.doFinal(encrypted);
            System.out.println("The string was:");
            System.out.println(new String(output));
        } catch (Exception e)
         {
             e.printStackTrace();
         }
    }
}
//returns:
```

```
//Usage: java EncryptTest string

//Source: Wisdom is judged by her fruit.

//87 105 115 100 111 109 32 105 115 32

//106 117 100 103 101 100 32 98 121 32

//104 101 114 32 102 114 117 105 116 46

//

//Encrypted:

//97 3 28 11 -80 11 23 24 96 37

//125 95 1 8 -118 124 60 113 -60 114

//4 -79 86 24 84 -24 -96 -21 -93 56

//-103 21

//

//The string was:

//Wisdom is judged by her fruit.
```

The two highlighted lines in this code are the beginning of the decrypt section. You can initialize a **Cipher** object to encode or decode. The key, in this case, comes from the code just above these two highlighted lines, in the same method. In real life, you'd probably have to supply the key in another way (for example, by using an input file).

Encrypting a File

The previous solution demonstrated how to encrypt a string. This next solution goes a step further and encrypts an entire file. Although this seems like it should be a similar process, there are several pitfalls you have to avoid. Fortunately, there is a special class, the **Cipher-OutputSream**, that handles most of these details for you. Listing 11.5 shows the program. The **CipherOutputStream** (see the highlighted section of Listing 11.5) allows you to encode anything that you would normally use as a stream. That might be a file, a network socket, or even a string if you wanted to use a string-based stream.

You pass a **Cipher** to the **CipherOutputStream** constructor. Obviously, if that **Cipher** is set to encrypt, the stream will encrypt. If the **Cipher** is set to decrypt, the stream will decrypt (see the next Immediate Solution).

Listing 11.5: Encrypting a file.

```
import java.security.*;

import javax.crypto.*;

import java.io.*;


class EncryptFile

{

    public static void main(String args[])

    {


   if (args.length != 3)

    {

       System.out.println("Usage: java EncryptTest " +

                   "sourceFile keyFile encryptedSourceFile");

    }


    try

    {

        String sourceFile = args[0];

        String encryptedFile = args[2];

        String keyFile = args[1];
```

```
String algorithm = "DES";
String mode = "ECB";
String padding = "PKCS5Padding";
String transformation = algorithm + "/" +
                        mode + "/" +
                        padding;

KeyGenerator kg = KeyGenerator.getInstance(algorithm);
SecretKey key = kg.generateKey();

// initialize in encrypt mode
Cipher cipher = Cipher.getInstance(algorithm);
cipher.init(Cipher.ENCRYPT_MODE, key);

// Create Input stream to get file contents
FileInputStream fis = new FileInputStream(sourceFile);

// Create Output stream save encrypted data to file
FileOutputStream fos =
                new FileOutputStream(encryptedFile);

// Create CipherOutputStream object
CipherOutputStream cos =
                new CipherOutputStream(fos, cipher);
// feeding bytes into CipherOutputStream
```

```
        // automatically encrypts them
        int ch;
        while ((ch = fis.read()) != -1)
        {
            cos.write(ch);
        }

        // Flush output stream
        cos.flush();

        fis.close();

        // write key to file else message is lost forever
        FileOutputStream keyfile =
                                    new FileOutputStream(keyFile);
        keyfile.write(key.getEncoded());
        keyfile.close();
  } catch (Exception e)
   {
      System.out.println("Caught Exception: " + e);
   }
 }
}
//usage: java EncryptFile genesis_1.txt key.key genesis_1.enc
```

Related solution:	Found on page:
Creating a File Stream	145

Decrypting a File

After you've encrypted a file, you use **CipherOutputStream** again to decode it. The difference is that in the **CipherOutputStream** constructor, you provide a **Cipher** object initialized to do decrypting. Listing 11.6 shows a program that will decode a file encrypted with Listing 11.5.

Listing 11.6: Decrypting a file.

```
import java.security.*;

import javax.crypto.*;

import javax.crypto.spec.*;

import java.io.*;


class DecryptFile

{

   public static void main(String args[])

    {

       if (args.length != 3)

        {

         System.out.println("Usage: java DecryptFile " +

                            "encryptedFile  keyFile decryptedFile");

        }


        try

        {

         String encryptedFile = args[0];

         String keyFile = args[1];
```

```
String decryptedFile = args[2];


String algorithm = "DES";
String mode = "ECB";
String padding = "NoPadding";
String transformation = algorithm + "/" +
                        mode + "/" +
                          padding;


// get the key to decrypt
FileInputStream keyStream =
                            new FileInputStream(keyFile);
byte[] keyBytes = new byte[keyStream.available()];
  keyStream.read(keyBytes);
  keyStream.close();


SecretKeyFactory keyFactory =
                    SecretKeyFactory.getInstance(algorithm);
DESKeySpec dk = new DESKeySpec(keyBytes);
SecretKey secretKey = keyFactory.generateSecret(dk);


  // initialize in decrypt mode
Cipher cipher = Cipher.getInstance(algorithm);
 cipher.init(Cipher.DECRYPT_MODE, secretKey);
```

```
// Create Input stream to get encrypted file contents
FileInputStream fis =
                    new FileInputStream(encryptedFile);

//Create Output stream to save decrypted data to file
FileOutputStream fos =
                    new FileOutputStream(decryptedFile);

// Create CipherOutputStream object
CipherOutputStream cos =
                    new CipherOutputStream(fos, cipher);

// feeding bytes into CipherOutputStream
// automatically decrypts them
int ch;
while ((ch = fis.read()) != -1)
{
cos.write(ch);
}

// Flush output stream
cos.flush();

fis.close();
cos.close();
```

TEAMFLY

Team-Fly®

```
    } catch (Exception e)
    {
        System.out.println("Caught Exception: " + e);
    }
  }
}
//usage: java DecryptFile genesis_1.enc key.key genesis.txt
```

The process of encrypting and decrypting files without the **CipherInputStream** and **CipherOutputStream** objects is prone to error. The streams take care of padding data so that the algorithm always works with a fixed-size block and handles other details you'd rather not worry about.

Related solution:	Found on page:
Creating a File Stream	145

Signing Messages and Files

How can you use encryption to sign a message? That is, given a message, how can you be sure that it originated from the supposed sender and wasn't modified in transit? One approach is to compute a digest of the message and encrypt it. The recipient can then compute the same digest of the message and decrypt the digest that was sent. If the digests don't match, something is amiss. Either the sender did not have the correct keys, or someone tampered with the message.

The **Signature** class encapsulates this logic. The program in Listing 11.7 demonstrates how to sign a file digitally by using **Signature**.

Listing 11.7: Digitally signing a file.

```
import java.security.*;
import java.io.*;

class MakeSignature
{
```

```
public static void main(String[] args)
 {
    if (args.length != 1)
    {
       System.out.println("Usage: GenSig nameOfFileToSign");
    } else try
    {
      /* random number generator to seed key maker */
       SecureRandom random =
                    SecureRandom.getInstance("SHA1PRNG", "SUN");
      /* setup utility objects */
       byte[] byteArray;
       FileSave file = new FileSave();

      /* Generate the private and public keys */
       KeyPairGenerator keyGen =
                     KeyPairGenerator.getInstance("DSA", "SUN");
       keyGen.initialize(1024, random);
       KeyPair pair = keyGen.generateKeyPair();
       PrivateKey privateKey = pair.getPrivate();
       PublicKey publicKey = pair.getPublic();

      /* Initialize Signature object with the private key */
       Signature dsa =
                    Signature.getInstance("SHA1withDSA", "SUN");
```

```
        dsa.initSign(privateKey);

      /* Save the private key in a file */
        byteArray = privateKey.getEncoded();
        file.saveFile("privatekey", byteArray);

      /* feed data from file to the Signature object */
       FileInputStream fis = new FileInputStream(args[0]);
      BufferedInputStream bin = new BufferedInputStream(fis);
       byte[] buffer = new byte[1024];
        int len;
       while (bin.available() != 0)
        {
            len = bin.read(buffer);
            dsa.update(buffer, 0, len);
        }
        bin.close();

      /* hash, sign data and return signature = byte array */
        byteArray = dsa.sign();

      /* Save the signature in a file */
        file.saveFile("signature", byteArray);

      /* Save the public key in a file */
```

```
            byteArray = publicKey.getEncoded();
             file.saveFile("publickey", byteArray);

        } catch (Exception e)
         {
            System.err.println("Caught exception " + e.toString());
         }
    }
}
//helper class
class FileSave
{
    private FileOutputStream fis;

/* if you want separate file setting action
     void setFileName(String fileName)
    {

        this.fileName = filename;
    }
*/
     void saveFile(String fileName, byte[] fileData)
    {
         try
         {
```

```
            this.fis  = new FileOutputStream(fileName);
             this.fis.write(fileData);
             this.fis.close();
        } catch (Exception e)
         {
           System.err.println("Exception " + e);
         }
    }
}
```

The two highlighted lines on the previous page are the heart of this program. They are where the file's bytes are sent to the **Signature** object. This object computes the digest and encrypts it with the private key. The **FileSave** class is simply a helper class to store the results. This program produces three new files—one each for the private key, the public key, and the signature itself.

Verifying Messages and Files with Digital Signatures

Once you have a signed file, you need a way to verify the digital signature. Remember that a digitally signed message doesn't alter the original message in any way, so it might be in plain-text form. (Of course, you could also encrypt the entire file to keep it private; encryption and signing both use encryption, but they use it for different reasons.)

To verify a message, you compute the digest and then use the public key to decrypt the message digest that resides in the signature file. Finally, you'll compare the two message digests. The program in Listing 11.8 demonstrates how to do this.

Listing 11.8: Verifying a digitally signed file.

```
import java.security.*;
import java.security.spec.*;
import java.io.*;


class VerifySignature
```

```
{
   public static void main(String[] args)
    {
       if (args.length != 1)
        {
            System.out. println("Usage: GenSig nameOfFileToVerify");
       } else
        {
         try
        {
          /* need random number generator to seed key maker */
            SecureRandom random =
                        SecureRandom.getInstance("SHA1PRNG", "SUN");

             /* setup utility objects */
           byte[] publicKeyBytes, signatureBytes;
           FileOpen file = new FileOpen();

            /* get public key */
           publicKeyBytes = file.getFileBytes("publickey");
           X509EncodedKeySpec pubKeySpec =
                              new X509EncodedKeySpec(publicKeyBytes);
           KeyFactory keyFactory =
                                KeyFactory.getInstance("DSA", "SUN");
           PublicKey publicKey =
```

```
                    keyFactory.generatePublic(pubKeySpec);

        /* Get signature */
        signatureBytes = file.getFileBytes("signature");

       /* Initialize Signature with the public key */
        Signature dsa =
                    Signature.getInstance("SHA1withDSA", "SUN");
         dsa.initVerify(publicKey);

       /* feed data from file to the Signature object */
        FileInputStream fis = new FileInputStream(args[0]);
       BufferedInputStream bin = new BufferedInputStream(fis);
       byte[] buffer = new byte[1024];
        int len;
       while (bin.available() != 0)
        {
            len = bin.read(buffer);
            dsa.update(buffer, 0, len);
        }

        bin.close();

       boolean verification = dsa.verify(signatureBytes);
        System.out.println("signature good? " + verification);
```

```
        } catch (Exception e)
         {
           System.err.println("Caught exception " + e.toString());
         }
        }
    }
}
//helper class
class FileOpen
{
    private FileInputStream fis;
     private byte[] fileBytes;

    byte[] getFileBytes(String fileName)
    {

        try
        {

            this.fis = new FileInputStream(fileName);
           int len = this.fis.available();
           this.fileBytes = new byte[len];
            this.fis.read(this.fileBytes);
            this.fis.close();
```

```
        } catch (Exception e)
         {
            System.err.println("Exception " + e);
         }
         return this.fileBytes;
    }
}
```

In this program, the highlighted lines on the previous page are the most important. The first line is where the data from the file is sent to the **Signature** object. Remember that you have to recompute the digest for comparison purposes. The highlighted line that calls **verify** actually does the comparison operation.

Chapter 12: Internationalization

In Brief

The prevalence of electronic communications has made the world seem like a smaller place. The Internet in particular has made the world seem positively tiny. Whereas international trade was once the province of big corporations, now even a small business can access the global marketplace. A U.S. company might routinely import materials from Hong Kong and Bulgaria, and, in turn, it might export electronic parts to customers on every continent. Books published in the United States might be translated into such diverse languages as Japanese, Russian, Polish, Spanish, and German.

This new international commerce often requires special care to make sure that software and Web sites can handle non-English languages. Once a specialty, internationalization is now part of mainstream software development. Often called I18N (because the word "internationalization" begins with an "i," has 18 other letters, and ends with an "n"), internationalization actually encompasses several techniques that facilitate the use of software with different languages.

Internationalization can deal only with technical issues, however, not with cultural ones. Also, it doesn't solve several common problems in user-interface design—that's up to you. What I18N can do is to help you replace (localize) the strings you use so that your prompts, error messages, and report titles are not tied to English. I18N also provides a way for you to format dates, numbers, and currency to conform to the user's preferences (the locale). However, I18N won't handle currency conversion; that's another task left for you.

I18N does not take care of problems that aren't immediately obvious. For example, some languages are not read from left to right; some are read up and down. Even languages similar to English can cause a problem. For example, German is superficially similar to English (at least, compared to Farsi or Cantonese), but the German language has many extremely long words composed of other words. This length often makes text boxes and other user-interface elements too short to contain their messages.

I18N Strings

At the heart of Java's I18N support is Unicode. How do your programs store characters? Internally, Java uses the Unicode format, which is a standard 16-bit character set used to represent glyphs for nearly every known language and a number of extra symbols.

For external data (that is, data in files or other external sources), Java uses an encoding scheme known as UTF-8. This is a particular way of storing characters in which the initial bit pattern determines the number of bytes in the character. (Remember that a byte contains eight bits, numbered from 0 to 7.) Having this variable bit pattern lets you store data in an efficient but versatile manner. The UTF-8 scheme provides three types of characters: one-byte, two-byte, and three-byte. Here's how it works:

Team-Fly®

- *One-byte characters*—If bit number 7 of the first byte is set to 0, then the character is made up of only one byte.
- *Two-byte characters*—If the first three bits (numbers 7, 6, and 5) are set to 110 (binary), then the character consists of two bytes. In this case, the second byte must begin with 10 (binary), which leaves 11 bits remaining to define the character.
- *Three-byte characters*—If the character requires more than 11 significant bits, the UTF-8 scheme says that Java must use three bytes to store it. In this case, the first byte must start with 1110 (binary). The next two bytes each start with 10 (binary). This scheme lets you store the full 16 bits.

The UTF scheme has several advantages. All pure ASCII files are already proper UTF-8 files, so you don't have to convert any existing data. (Remember that pure ASCII characters are all less than 0x80, so bit 7 will always be zero.) In addition, because of the bit patterns, it's easy to recognize—by looking at the starting bits—whether a byte starts a sequence, belongs to a sequence, or is its own character. Any byte that's part of a sequence starts with 10 (binary); any byte that does not start with 10 (binary) either begins a sequence or is a single byte.

Thanks to Unicode, Java has no problem representing strings in practically any language or alphabet. Unicode has characters for Cyrillic, Greek, and many other alphabets. You could create multiple versions of your program, each having strings in a different language. Of course, it is up to you to supply the translations.

There is a better way, however. First, extract all the display text from code, and place the text into a repository file. When the program needs a particular string, have the program retrieve the string from the repository. That way, you can change languages by simply replacing the repository.

For this scheme to work, you need two things. First, you need a way to store strings for later retrieval. And second, you need a way to detect which language the user wants to use. You could handle all the details yourself, as in the following program:

```
public class EnglishFrench
{
   public static void main(String args[])
    {
      String language = "English", message = "";


       if (args.length != 1)
       {
         System.out.println("Usage: java EnglishFrench " +
                                  "English|French");
       } else
       {
          language = args[0];
```

```
        }

      String messageInFrench = "Salut. Java est facile.";
      String messageInEnglish = "Hello. Java is easy.";

       if( language.equalsIgnoreCase("English") )
       {
           message = messageInEnglish;
       } else
       if( language.equalsIgnoreCase("French") )
       {
           message = messageInFrench;
       }

       System.out.println(message);
   }
}
//java EnglishFrench
//returns:
//Usage: java EnglishFrench English|French
//Hello. Java is easy.
//
//java EnglishFrench French
//returns:
//Salut. Java est facile.
```

This program uses a simple option on the command line to decide whether to display its text in English or French. However, the obvious problem is that the text cannot change unless you recompile the program. This is a harbinger of a nasty maintenance burden.

The better way to manage text translations is to move the text pieces from code to a repository outside the program. The following pseudocode walks through the steps:

```
String getTextMethod(String textName)
{
   open repository file;
    find textName;
```

```
    grab associated textValue;
    close repository file;
    return textValue;
}
```

We can write our own helper class to perform the steps outlined by the pseudo-code. However, the second Immediate Solution later in this chapter (see "Internationalizing Text with a **MessageBundle**") shows you how to use a class called the **ResourceBundle**, which takes care of the pseudo-code's chores automatically.

The other problem with this simple approach is that it can lead to a great deal of duplication. For example, localized versions for the United States, Canada, and the United Kingdom will all have strings that are mostly the same but might differ in a few details, such as word spellings.
With a **ResourceBundle**, Java can manage a hierarchy of repositories. Java will search in the most specific repository first (say, English, for the United Kingdom). If it can't find the requested string in that bundle, it will search a bundle for generic English (if one exists). If that fails, Java will continue looking in a generic bundle before giving up.

Locales

Before you can put resource bundles to use, it's important to understand locales. A *locale* (represented by the **java.util.Locale** class) indicates a language, a country, and a dialect or variant. For example, your user's language might be English. If you know the user's country, you can specify American English or British English.

Great Britain uses a different currency symbol and different date formats from those used in the United States. Besides that, some words are spelled differently ("honour" and "honor," for example). Even words for common objects are different ("petrol" and "gasoline"; "lift" and "elevator"). To make matters more complex, some countries have dialects, which are further subdivisions or variants of a single language. Java's **Locale** class also accounts for dialects.

Beyond words, languages can be different in other seemingly innocuous, but important, ways. For example, in Spanish, "ch" is often (depending on the country) treated as one letter when it's alphabetized. Thai and Lao vowels have peculiar sorting rules as well. The **java.text.Collation** class is sensitive to these rules. You can use this class to determine if one word is alphabetically before or after another word (which is the essence of sorting, of course). The **compare** method of this class tells you the order of two strings. You can also use other methods to set options like case sensitivity. The documentation for this class is a wealth of information about language-sorting issues.

Most modern operating systems also support the use of locales. Your program can call the static function **java.util.Locale.getDefault** to learn the default locale. You can then use member functions of the **Locale** object to get the specific country and language information. If you want to change the current locale, you can use the **setDefault** function.

Immediate Solutions

Using the Locale Object

The **Locale** object encapsulates the country and language of a location. You often need both the country and the language to decide how to display text, currency, and dates. Usually, you'll use the static function **java.util.Locale.getDefault** to obtain the system's idea of the current **Locale**. However, you can also construct your own arbitrary **Locale** objects.

For example, the following code initializes a **Locale** object to represent the English language:

```
Locale english = new Locale("en","","");
```

Java encodes as constants the languages and countries most often used in code. The previous line is equivalent to this:

```
Locale english = Locale.ENGLISH;
```

The previous English example is a language-specific **Locale**. The following is a country **Locale** that includes both a location and a language:

```
Locale UnitedStates = new Locale("en","US","");
```

The previous line is equivalent to this:

```
Locale UnitedStates = Locale.US;
```

The other language **Locale** constants are **FRENCH**, **GERMAN**, **ITALIAN**, **JAPANESE**, **KOREAN**, **CHINESE**, **SIMPLIFIED_CHINESE**, and **TRADITIONAL_CHINESE**. The other country **Locale** constants are **FRANCE**, **GERMANY**, **ITALY**, **JAPAN**, **KOREA**, **CHINA**, **PRC**, **TAIWAN**, **UK**, **CANADA**, and **CANADA_FRENCH**.

The program in Listing 12.1 demonstrates the most important methods of the **Locale** object.

Listing 12.1: Demonstrating the Locale object.

```
import java.util.*;

import java.io.*;



public class LocaleTest

{

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

```
{
    int i;
    boolean f;
    String s;
   Locale usa = new Locale("EN", "US");
    Locale canada;

    Locale.setDefault(usa);
    LocalReport tabloid = new LocalReport();
    tabloid.print(Locale.CANADA);
    tabloid.print(Locale.getDefault());

   //getDisplayCountry(Locale.US); //alternate method
   //getDisplayLanguage(Locale.US);//alternate method
    //getDisplayName(Locale.US);//alternate method
    //getDisplayVariant(Locale.US);//alternate method

   canada = (Locale)Locale.CANADA.clone();
    System.out.print("Is Locale.CANADA_FRENCH the same as "
                +
    "Locale.CANADA? ");
    System.out.println( canada.equals(Locale.CANADA_FRENCH));

   String[] countries = Locale.getISOCountries();
    tabloid.print(countries, "ISOCountries");
```

```
        String[] languages = Locale.getISOLanguages();
         tabloid.print(languages, "ISOLangu ages");

        Locale[] locales = Locale.getAvailableLocales();
         tabloid.print(locales, "AvailableLocales");
    }
}
class LocalReport
{
   private static final PrintStream o = System.out;

    public void print(Locale L)
    {
         o.println(L.getDisplayName() + ":");
        o.print(L.getCountry() + "\t");
        o.print(L.getISO3Country() + "\t");
         o.println(L.getDisplayCountry());
        o.print(L.getLanguage() + "\t");
        o.print(L.getISO3Language() + "\t");
         o.println(L.getDisplayLanguage());
        o.print(L.toString() + "\t");
         o.println(L.hashCode());
         o.println();

         //L.getDisplayVariant();
```

```
        //L.getVariant();
  }


  //overload the print method
  public void print(String[] s, String caption)
  {


      o.print(caption + ": ");
     for(int i=0; i<s.length; i++)
      {
        o.print(s[i] + " " );
      }
      o.println("\n");
  }
  //overload the print method
  public void print(Locale[] L, String caption)
  {
     o.print(caption + ": ");
     for(int i=0; i<L.length; i++)
      {
         o.print(L[i].toString() + " " );
      }
      o.println();
  }
}
```

```
//returns:
//English (Canada):
//CA        CAN       Canada
//en        eng       English
//en_CA     1271
//
//English (United States):
//US        USA       United States
//en        eng       English
//en_US     1591
//
//Is Locale.CANADA_FRENCH the same as Locale.CANADA? false
//ISOCountries: AD AE AF AG AI AL AM AN AO AQ AR AS AT AU AW AZ
//BA BB BD BE BF BG BH BI BJ BM BN BO BR BS BT BV BW BY BZ CA
//--removed output to save space--
//ISOLanguages: aa ab af am ar as ay az ba be bg bh bi bn bo br
//ca co cs cy da de dz el en eo es et eu fa fi fj fo fr fy ga gd
//--removed output to save space--
//AvailableLocales: en en_US ar ar_AE ar_BH ar_DZ ar_EG ar_IQ
//ar_JO ar_KW ar_LB ar_LY ar_MA ar_OM ar_QA ar_SA ar_SD ar_SY
//--removed output to save space—
```

Internationalizing Text with a MessageBundle

The **MessageBundle** class takes care of finding a string in an appropriate language. The scheme is simple—you place your translations into files that have the same name except for the addition of country and language identifiers.

Here are the steps you'll need to follow to use the **MessageBundle**:

1. Make a list of the program strings you want to localize.
2. Give each string a unique name.
3. Create a file that contains a line for each string. The line should contain the unique name (the key), an equals sign, and the string. For example:
4. YesMessage = Yes
5. Save the file using a base name (for example, "captions") and an extension of .properties.
6. You can create more files using the same base name, but with a particular language and, optionally, country code (for example, captions_fr_FR.properties or captions_en.properties). These files have the same format but provide strings in different languages.

For example, this might be your base captions.properties file:

```
//English
YesMessage = Yes
NoMessage = No
ThankYouMessage = Thank you
```

And the captions_fr_FR file could contain:

```
//French
YesMessage = Oui
NoMessage = Non
ThankYouMessage = Merci
```

The codes are defined by the ISO (International Organization for Standardization, the most authoritative international-standards body). You can find a complete list of language codes at http://www1.ics.uci.edu/pub/ietf/http/related/iso639.txt and the list of country codes at http://www.userpage.chemie.fu-berlin.de/diverse/doc/ISO/3166.html The code for English is **en**, and the code for the United States is **US**. You can find all the codes supported by Java by looking in the **Locale** class source.

The first file, captions.properties, contains the default key-value pairs. **MessageBundle** starts looking at the most specific file that matches the requested locale. If the string doesn't exist in this bundle, the class continues searching in the less specific bundles. The program in Listing 12.2 demonstrates how to use the **ResourceBundle** to get a string in one of several languages. The static **ResourceBundle.getBundle** accepts the base name of the file ("captions") and a locale and returns a **ResourceBundle** object. After making this call, the program simply calls **getString** to retrieve the correct string by its key.

Listing 12.2: Using the ResourceBundle.

```
import java.util.*;

public class ResourceBundleTest {

    static public void main(String[] args) {

        String language;
        String country;

        if (args.length != 2) {
            language = new String("en");
            country = new String("US");
        } else {
            language = new String(args[0]);
            country = new String(args[1]);
        }

        Locale currentLocale;
        ResourceBundle messages;
        currentLocale = new Locale(language, country);

        messages = ResourceBundle.getBundle("Captions",
                                            currentLocale);
        String YesMessage = "\"Yes\" in " + language +
                "  is: \"" +
```

TEAMFLY

Team-Fly®

```
                messages.getString("YesMessage") + "\"";
        String NoMessage = "\"No\" in " + language + " is: \"" +
                messages.getString("NoMessage") + "\"";
        String ThankYouMessage = "\"Thank You\" in " +
                language + " is: \"" +
                messages.getString("ThankYouMessage") + "\"";

        System.out.println(YesMessage);
        System.out.println(NoMessage);
        System.out.println(ThankYouMessage);
    }
}
//java ResourceBundleTest fr FR
//returns:
//"Yes" in fr is: "Oui"
//"No" in fr is: "Non"
//"Thank You" in fr is: "Merci"
```

Internationalizing Text with a ListResourceBundle

The **ListResourceBundle** serves the same function as the **ResourceBundle** . The differences between the two classes lie in where the data is stored and which data types are allowed. Because **ResourceBundle** uses text files, it can handle only strings for both the keys and the values. Of course, you can add sophistication by extending it and internally converting text into numbers and dates in addition to converting text to strings. This is somewhat clumsy, but it is possible.

The **ListResourceBundle** uses class files instead of text files. Of course, this means that to update a **ListResourceBundle**, you have to compile Java files to create the class files. The class files are small, however, and are not directly related to the program's operation, so distributing them isn't any harder than distributing new text. For the small price of a compile, you gain the ability to store any object type— including strings in the bundle.

Listing 12.3 shows an example **ListResourceBundle**. Notice that Listing 12.3 extends **ListResourceBundle** and simply returns an array of **Object** arrays. This array contains the keys and values contained in the bundle.

Listing 12.3: Demonstrating the ListResourceBundle object.

```
import java.util.*;

public class PoisonousAnimal_venomous_JELLYFISH extends
                    ListResourceBundle
{
   public Object[][] getContents()
   {
       return contents;
   }

   private Object[][] contents =
   {
      { "Name", "box jellyfish" },
       { "Habitat", "Australia" },
       { "Potency", new Integer(60) },
       { "Method", "touch" },
       { "Size", new Double(.3) }
   };
}
```

This example doesn't use languages; instead, it classifies poisonous animals. However, the same principle applies to classifying languages and countries by using the ISO codes; the **ListResourceBundle** doesn't care what strings you use. For this example, there are two subdivisions

of the **PoisonousAnimal** category. Notice that the first description (**venomous**) appended to the base name is lowercase. Also, the second description (**JELLYFISH**) is uppercase. You'll get an error if you don't follow this convention.

Listing 12.4 shows how you can use the **ListResourceBundle**. Notice that some of the data in the bundle is not a string. In particular, **Size** is a **Double**, and **Potency** is an **Integer** object.

Listing 12.4: Using a ListResourceBundle.

```
import java.util.*;


public class ListResourceBundleDemo
{
  static public void main(String[] args)
  {
  Locale[] spookyVermin = {
    new Locale("venomous","SNAKE"),
    new Locale("poison","FROG"),
    new Locale("venomous","JELLYFISH")
  };

  for (int i = 0; i < spookyVermin.length; i ++) {
    System.out.println("Spooky vermin = " +
        spookyVermin[i]);
      spookyReport(spookyVermin[i]);
      System.out.println();
    }
  }
```

```
  static void spookyReport(Locale currentLocale) {


       ResourceBundle animal =

          ResourceBundle.getBundle("PoisonousAnimal",

               currentLocale);


      String name = (String)animal.getObject("Name");

      System.out.print("The " + name);

      String habitat = (String)animal.getObject("Habitat");

     System.out.println(" lives in " + habitat + ".");

      Integer potency = (Integer)animal.getObject("Potency");

       Double size = (Double)animal.getObject("Size");

      System.out.print("This " + size.toString() +

                                              " meter creature");

      System.out.print(" can kill " + potency.toString());

       String method = (String)animal.getObject("Method");

     System.out.println(" people with one " + method + ".");

     }

}

//returns:

//Spooky vermin = venomous_SNAKE

//The Gaboon Viper lives in tropical Africa.

//This 1.83-meter creature can kill 30 people with one bite.

//

//Spooky vermin = poison_FROG
```

```
//The Poison Arrow Frog lives in South America.

//This 0.1-meter creature can kill 10 people with one touch.

//

//Spooky vermin = venomous_JELLYFISH

//The box jellyfish lives in Australia.

//This 0.3-meter creature can kill 60 people with one touch.
```

Internationalizing Numbers and Currency

Numbers and currency are two types of data that need to be localized for international applications. The program in Listing 12.5 demonstrates how to use Java's formatting objects. Java uses formatting appropriate to the current locale or to a locale you provide, if you wish.

Listing 12.5: Formatting numbers, currency, and percentages.

```
import java.util.*;

import java.text.*;


public class InternationalizedNumbers

{

   static public void showAstronomy(Locale L)

    {

       Integer mercuryRadius = new Integer(2430000);

       Double mercuryMass = new Double(3.18e23);

       Integer venusRadius = new Integer(6060000);

       Double venusMass = new Double(4.88e24);

       Integer earthRadius = new Integer(6370000);

       Double earthMass = new Double(5.98e24);


        NumberFormat prettyNumber;
```

```
        String s;


      prettyNumber = NumberFormat.getNumberInstance(L);


      s = prettyNumber.format(mercuryRadius);
      System.out.print("Mercury radius = " + s);
       s = prettyNumber.format(mercuryMass);
      System.out.println("; mass = " + s);


       s = prettyNumber.format(venusRadius);
        System.out.print("Venus radius = " + s);
       s = prettyNumber.format(venusMass);
      System.out.println("; mass = " + s);


       s = prettyNumber.format(earthRadius);
      System.out.print("Earth radius = " + s);
       s = prettyNumber.format(earthMass);
      System.out.println("; mass = " + s);
        System.out.println();
  }


 static public void showDebt(Locale L)
  {
     Double debt = new Double(5777412794412.94);
       NumberFormat money;
```

```
        String s;

        money = NumberFormat.getCurrencyInstance(L);
        s = money.format(debt);
        System.out.println("The US national debt is " + s);
    }

    static public void showBattingAverage(Locale L)
    {
        Double battingAverage = new Double(0.344);
        NumberFormat average;
        String s;

        average = NumberFormat.getPercentInstance(L);
        s = average.format(battingAverage);
        System.out.println("The batting average is " + s);
    }

    static public void main(String[] args)
    {
        Locale[] locales = Locale.getAvailableLocales();

        for (int i = 0; i < locales.length; i++)
        {
```

```
            System.out.println();
            System.out.println(locales[i].getDisplayName());
            showAstronomy(locales[i]);
            showDebt(locales[i]);
            showBattingAverage(locales[i]);
        }
    }
}
//returns:
//
//--removed output to save space--
//
//Albanian (Albania)
//Mercury radius=2.430.000; mass=318.000.000.000.000.000.000.000
//Venus radius=6.060.000; mass=4.880.000.000.000.000.000.000.000
//Earth radius=6.370.000; mass=5.980.000.000.000.000.000.000.000
//
//The US national debt is Lek5.777.412.794.412,94
//The batting average is 34%
//
//--removed output to save space--
//
//Chinese (Taiwan)
//Mercury radius=2,430,000; mass=318,000,000,000,000,000,000,000
//Venus radius=6,060,000; mass=4,880,000,000,000,000,000,000,000
```

```
//Earth radius=6,370,000; mass=5,980,000,000,000,000,000,000,000
//
//The US national debt is NT$5,777,412,794,412.94
//The batting average is 34%
```

The highlighted lines on the previous page show where the formatting takes place. Listing 12.5 uses the **Locale.getAvailableLocales** method to get all of the locales and then loops through each one, showing you the differences in formatting.

Formatting Dates

Dates are another data type that Java has already internationalized for you. The program in Listing 12.6 demonstrates how to display the five date styles in all the languages supported by Java.

Listing 12.6: Formatting dates.

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

public class InternationalizedDates
{
   static public void showDates(Locale L)
    {
        Date today;
        String s;
        DateFormat d;
       today = new Date();

       d = DateFormat.getDateInstance(DateFormat.DEFAULT,L);
       s = d.format(today);
        System.out.println(s);
```

```
        d = DateFormat.getDateInstance(DateFormat.SHORT,L);
        s = d.format(today);
         System.out.println(s);
        d = DateFormat.getDateInstance(DateFormat.MEDIUM,L);
        s = d.format(today);
         System.out.println(s);
        d = DateFormat.getDateInstance(DateFormat.LONG,L);
        s = d.format(today);
         System.out.println(s);
        d = DateFormat.getDateInstance(DateFormat.FULL,L);
        s = d.format(today);
         System.out.println(s);
    }

    static public void main(String[] args)
    {
        Locale[] locales = Locale.getAvailableLocales();

       for (int i = 0; i < locales.length; i++)
         {
             System.out.println();
             System.out.println(locales[i].getDisplayName());
             showDates(locales[i]);
         }
    }
```

TEAMFLY

Team-Fly®

```
}

//returns:

Dutch

//11-okt-01

//11-10-01

//11-okt-01

//11 oktober 2001

//donderdag 11 oktober 2001

//

//Portuguese (Brazil)

//11/10/2001

//11/10/01

//11/10/2001

//11 de Outubro de 2001

//Quinta-feira, 11 de Outubro de 2001

//

//Swedish (Sweden)

//2001-okt-11

//2001-10-11

//2001-okt-11

//den 11 oktober 2001

//den 11 oktober 2001
```

Related solution:	Found on page:
Working with Dates	187

Comparing Characters

When handling strings, be sure to use the **Character** class to test a character for set membership. For example, to test a character to see if it is a letter, you might be tempted to write:

```
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')) . . .
```

This is incorrect for many languages, however. A better test is the following:

```
if (Character.isLetter(letterCandidate)) . . .
```

You can also use the **java.text.Collation** object to compare strings to determine their lexical ordering. This class also uses locale rules.

Appendix A: Development Tools

Overview

In this book, we've used the Sun Microsystems development tools. After all, these tools represent the base level of Java, which is available for practically all platforms. Also, using Sun's tools allowed us to focus on the Java code and not on the various quirks and idiosyncrasies of tools. If you start writing a substantial amount of Java code, however, you'll probably want to use a different tool (although it's possible to write a lot of code with nothing more than the Emacs text editor and the basic Sun tools). Sun provides a graphical-user-interface tool called Forte. This product has a community edition you can download for free. Be warned: This program is written in Java, so you need a fast machine with plenty of memory in order to run it. (How fast and with how much memory depends on your operating system and which version of Forte you are running.) You can download Forte from Sun's Web site at http://www.java.sun.com.

Regardless of which tool you choose, you'll probably continue to use the same Java Virtual Machine (JVM) that you've always used. Every tool has the same job: converting source code to class files. The tools offer a graphical work environment, project management features, and enhanced debugging; in addition, many offer automated wizards that can produce some code automatically.

This appendix concentrates on the tools that Sun bundles with the Software Development Kit (SDK), plus a few other tools we think you should try for your Java development.

Note The popular integrated development environments (IDEs) either run Java's native compiler behind the scenes or come with the vendor's own compiler. Either way, the IDEs take care of many compiling chores for you, but the fundamental process is still the same behind the scenes.

For the purposes of this appendix, we will group development tools into three categories. The first group is the toolset that comes with the basic SDK. The majority of this appendix covers these tools. The second group consists of text editors that support Java syntax. They highlight keywords, automatically indent code, and display line numbers. Some of these editors also compile and execute code and provide rudimentary project management features. The third group consists of the enterprise IDE (integrated development environment). This IDE provides rapid J2EE application development, Java-to-database wizards, XML tools, and support for including JavaBeans. These tools often have version-control features that support team development.

Integrated Development Environments

Forte, VisualAge, Visual Café, and JBuilder are certainly the biggest names in IDEs, but hundreds of Java authoring tools are available. Even more tools exist that have nothing to do with authoring (for example, profiling tools). The top list of tools to keep your eye on is maintained by Sun at http://www.industry.java.sun.com/solutions/products/by_type/0,2359,all-28-0,00.html. An honorable

mention goes to JavaWorld's IDE list at http://www.javaworld.com/javaworld/tools/jw-tools-ide.html. (This list is more detailed than Sun's, but some parts get outdated easily.)

One thing you have to watch for: Many vendors' tools add libraries that are vendor-specific to perform certain tasks. (This is especially true if you use wizards or other tools to generate code automatically.) Then you have to remember to distribute these libraries with your application. Of the four top guns, only Forte doesn't include renegade classes. Another thing that is high-handed about IDEs is that many of them force you to use a certain coding convention or style.

One of the best features of most IDEs (and many code editors) is syntax coloring. By coloring keywords and common structures, an editor can help you browse your code more quickly.

Another handy feature in many IDEs is code completion. For example, JBuilder lets you insert code from an expanded list of user-defined templates. If you type "forb", highlight it, and press Ctrl+J, JBuilder replaces **forb** with a complete **for** block and places the cursor at the initiation statement point. Borland ships all the fundamental constructs and allows you to add your own. As you type, the editor starts guessing what you want. Another keyboard shortcut displays a pop-up window that provides a list of keywords, including your own classes and methods, to complete your expression. Most IDEs have a similar feature.

Code Editors

Many people prefer code editors to full IDEs. Although IDEs include everything but the kitchen sink, code editors are light and fast. Dozens of editors are similar in terms of features and reliability. A few expensive ones are very good, such as SlickEdit (http://www.slickedit.com), but the high price for a text editor makes us pause because a user can get 85 percent of the same functionality from free or inexpensive editors. Our favorites? We usually use one of these three:

- *UltraEdit*—http://www.idmcomp.com/products/index.html
- *NoteTab*—http://www.notetab.com
- *TextPad*—http://www.textpad.com

Other editors are better, but they are more difficult to learn. For example, Emacs isn't just a text editor; it is a way of life, and probably not useful to the casual user. (People who know Emacs use it for everything—text editing, document preparation, mail and news, compiling programs, browsing the Web, transferring files over the network, and more. Unfortunately, however, it takes a big investment in time to get familiar with this large, but powerful, program.)

javac—Java's Compiler

The compiler is what translates your source code into class files. You compile at the command prompt with the following command:

```
javac myProgram.java
```

This command will produce a file named myProgram.class. This class file is what runs on the JVM. You can supply several options for the javac compiler. Table A.1 lists the most common options.

Table A.1: Common options for the javac compiler.

Option	Description
-classpath classpath	Specifies the location for import classes (overrides the **CLASSPATH** environment variable).
-d directory	Specifies the destination directory for class files. (This option will append package name directories, so **-d c:\myclasses** for **com.mypackage.MyClass** produces **c:\myclasses\com\mypackage\ MyClass.class**.)
-deprecation	Prints each deprecated member or class.
-encoding encoding	Sets the source file's encoding method.
-g	Prints all debugging information instead of the default line number and file name.
-g:none	Turns off debugging information.
-g:{keyword list}	Provides specific debugging information for source, lines, and vars.
-help	Prints most of the information in this table.
-nowarn	Turns off warning messages.
-source release	When release is set to 1.4, the compiler accepts assertions.
-sourcepath sourcepath	Imports a class location.

Table A.1: Common options for the javac compiler.

Option	Description
-verbose	Produces additional output about each class loaded and each source file compiled.
-version	Displays version information.
-Xstdout filename	Places compiler mes sages in the named file.

One nice capability of the javac compiler is that you can place the list of options and source-file paths in files instead of retyping them each time. This is convenient when you are compiling more than one file or when you simply want to specify several options. You can create one file named args.txt and place the following in it:

-g

-deprecation

-d test1 \source \classes \

-verbose

Then create another file called classes.txt, and place the list of source files in it (use relative or absolute paths):

test1 \source\Splitter.java

test1 \source\flip.java

Development Tools

Now compile by typing the following command:

C:\DEV\java\jdk1.4\myPrograms>javac @args.txt @classes.txt

You will see something like this result in the command window:

[parsing started test1\source \Splitter.java]

[parsing completed 120ms]

[parsing started test1\source \flip.java]

[parsing completed 10ms]

[loading c:\dev\java\jdk1.4\jre\lib\rt.jar

(java/lang/Object.class)]

[loading c:\dev\java\jdk1.4\jre\lib\rt.jar

```
                                        (java/lang/String.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                        (java/lang/Exception.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                        (java/util/Random.class)]
[checking Splitter]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                    (java/lang/Throwable.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                (java/util/regex/Pattern.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                (java/util/regex/Matcher.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                (java/lang/CharSequence.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                    (java/io/Serializable.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                    (java/lang/Comparable.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                        (java/lang/System.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                    (java/io/PrintStream.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                (java/io/FilterOutputStream.class)]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                    (java/io/OutputStream.class)]
[wrote test1\source\classes\Splitter.class]
[checking flip]
[loading c:\dev\java\jdk1.4\jre\lib\rt.jar
                                (java/lang/StringBuffer.class)]
[wrote test1\source\classes\flip.class]
[total 591ms]
```

java—The Java Application Launcher

The java program starts a Java runtime environment, loads the class you specify, and then calls that class's **static main** method. Every Java application requires a **main** method to initialize it. You start a Java program like this:

java myProgram

The myProgram file is a class file, not a source file. The full file name is myProgram.class. Table A.2 lists the options for the java launcher.

Table A.2: Options for the java program launcher.

Option	Description
-classpath classpath (or just **–cp**)	Specifies the location for import classes (overrides the **CLASSPATH** environment variable).
-enableassertions[:<package name> "..." | :<class name>]	Enables assertions. Assertions are disabled by default.
-ea	Is the same as **-enableassertions** .
-disableassertions[:<package name> "..." | :<class> ;]	Disables assertions. This setting is the default.
-da	Is the same as **-disableassertions**.
-jar	Uses a JAR file name instead of a class name.
-verbose	Produces additional output about each dass loaded and each source file compiled.
-version	Displays version information and exits.
-showversion	Displays version information.

There are also extended options that begin with **—X**. These represent options that are not typically standard among different compilers. The **—Xprof** option is especially helpful when you are analyzing performance. For example, the **—Xprof** option was used to generate the following profile report:

Flat profile of 0.36 secs (36 total ticks): main

```
   Interpreted + native Method
 86.1%    0  +   31   java.io.FileInputStream.open
  2.8%    0  +    1   java.lang.Exception.<init>
  2.8%    1  +    0   java.util.Locale.getDefault
  2.8%    0  +    1   java.lang.StringBuffer.expandCapacity
  2.8%    0  +    1   sun.misc.URLClassPath$JarLoader.getJarFile
  2.8%    0  +    1   java.io.Win32FileSystem.canonicalize
100.0%    1  +   35   Total interpreted
```

javadoc—The Documenting Utility

The javadoc program extracts special comments from your Java source code and uses these comments to build HTML documentation. These special comments start with **/**** (which the Java compiler treats as an ordinary comment). Consider the following fictitious source file as an example:

```
/*
 * @(#)TruckSale.java 1.40 01/10/08
 *
 * Copyright 2001-2002 Truck Enterprises. All Rights Reserved.
 *
 * This software is the proprietary information of Truck
 * Enterprises. Use is subject to license terms.
 *
 */

/**
 * A <code>TruckSale</code> is a subclass of <code>Sales</code>
 * that manages a sale of a truck. It takes care of tallying all
 * the features and extras of the vehicle, including sales tax.
 * <p>
 * The methods of this class are required by many transactions.
 *
 * @author Truck Dude
 * @version 1.40 01/10/08
 * @see truck.sales
 * @since JDK1.4
 */
```

```
public class TruckSale extends Sale {
    static final long serialVersionUID = 6803508401927987242L;

     /**
    * Constructs a new TruckSale with <code>null</code> as its
     * detail customer.
    * The customer is not initialized, and may subsequently be
    * initialized by call to {@link #initCause}.
     */
   public TruckSale() {
        super();
    }

    /**
    * Constructs a new TruckSale with the specified detail
    * customer. The customer is not initialized, and may
     * subsequently be initialized by a call to
     * {@link #initCause}.
    * @param customer the detail customer. The detail
    * customer is saved for
    * later retrieval by the {@link #getMessage()} method.
     */
   public TruckSale(String customer) {
        super(customer);
    }

    /**
    * Constructs a new TruckSale with the specified detail
     * customer and occasion. <p>Note that the detail customer
     * associated with <code>occasion</code> is <i>not</i>
     * automatically incorporated in
     * this TruckSale's detail customer.
      *
    * @param customer the detail customer (which is saved for
```

TEAMFLY

Team-Fly®

```
     * later retrieval by the {@link #getMessage()} method).
    * @param occasion the occasion (which is saved for later
     *            retrieval by the
     *            {@link #getCause()} method). (A <tt>null</tt>
     *          value is permitted, and indicates that the
     *          occasion is nonexistent or unknown.)
     * @since 1.4
      */
    public TruckSale(String customer, Throwable occasion) {
          super(customer, occasion);
     }

     /**
     * Constructs a new TruckSale with the specified occasion
    * and a detail customer of <tt>(occasion==null ? null :
     * occasion.toString())</tt> (which
    * typically contains the class and detail customer of
     * <tt>occasion</tt>).
    * This constructor is useful for TruckSales that are little
    * more than wrappers for other throwables.
     *
    * @param occasion the occasion (which is saved for later
     *            retrieval by the
     *            {@link #getCause()} method). (A <tt>null</tt>
     *          value is permitted, and indicates that the
     *          occasion is nonexistent or unknown.)
     * @since 1.4
      */
    public TruckSale(Throwable occasion) {
          super(occasion);
     }

     /**
     * Gets the sale property indicated by the specified key.
```

```
     * <p>
    * First, if there is a security manager, its
    * <code>checkPropertyAccess</code> method is called with
    * the key as its argument. This may result in a
     * SecurityException.
     * <p>
     * If there is no current set of sale properties, a set
    * of sale properties is first created and initialized in
     * the same manner as
     * for the <code>getProperties</code> method.
      *
     * @param       key the name of the sale property.
      * @return      the string value of the sale property,
      *              or <code>null</code> if there is no property
      *              with that key.
     * @exception   SecurityException if a security manager
      *              exists and its
      *              <code>checkPropertyAccess</code> method
      *             doesn't allow access to the specified sale
      *              property.
     * @exception   NullPointerException if <code>key</code> is
      *              <code>null</code>.
      * @exception   IllegalArgumentException if <code>key</code>
      *              is empty.
      * @see         #setProperty
      * @see         java.lang.SecurityException
      * @see         java.lang.Sale#getProperties()
      */
    public static String getProperty(String key) {
     if (key == null) {
         throw new NullPointerException("key can't be null");
      }
     return props.getProperty(key);
    }
```

}

The code is not valid, but the comments and javadoc tags are. When you run javadoc on a source file like the one just shown, you get several new files, including overview-tree.html, index-all.html, deprecated-list.html, allclasses-frame.html, index.html, packages.html, TruckSale.html, package-list.html, help-doc.html, and stylesheet.css. These .html, Web pages have a nice layout, and all the links work. These documentation pages are the same as the ones generated for the Java libraries themselves.

The javadoc program looks for certain tags of the form **@tagname**. Table A.3 shows the most common tags.

Table A.3: javadoc documentation tags.

Tag	Description	Example
/**	Beginning of the documentation	**/**This is the start of the help file.**
@see	Cross-reference	**@see <a href="spec.html">Java Spec</a>**
@author	Author	**@author Devyn from heaven**
@version	Version	**@version 1.4.3beta**
@param	Method and constructor parameter	**@param key[] the key array**
@return	Return value	**@return the final sales summation**
@exception	Exception	**@exception IndexOutOfBoundsException**
{@docRoot}	Relative path	**<a href="{@docRoot}/ intro.html">Introduction</a>**
@deprecated	Signal to stop using	**@deprecated as of version 5.8**
{@link}	Hyperlink	**Use {@link #setDate()} method**

Table A.3: javadoc documentation tags.

Tag	Description	Example
@serial	Serializable field	**@serial count**
@serialData	Data description	**@serialData file contents**
@serialField	**ObjectStreamField** component	**@serialData osf**
@since	Introduction stamp	**@since 5.8**
@throws	Same as **@exception**	
@version	Version stamp	**@version 3.2beta**

Tip Remember that javadoc generates HTML. You can enclose HTML formatting in the comments. Also, you should encode the HTML special characters, such as the less-than (**<**), greater-than (**>**), and ampersand **&** symbols. For example, to write a less-than sign, you should write **&** to prevent the Web browser from treating your text as an HTML tag.

jar—The Java Archive Tool

The jar tool combines multiple files into a single JAR archive file. Archives are files that contain other files. These files are compressed in Zip archive format. Unlike ordinary Zip files, however, the JAR files contain extra information understood by the JVM.

A JAR archive allows you to distribute a single file that contains all the classes and other files required by your application. In addition to the class files, the JAR file can contain a manifest that specifies additional information for the JVM. The most common use for a manifest is to specify a class in the JAR file that contains a **main** method. Table A.4 lists the options for the **jar** command.

Table A.4: Options for the jar command.

Option	Description
c	Creates a new or empty archive.

Table A.4: Options for the jar command.

Option	Description
t	Lists the table of contents.
x file	Extracts all files or just the named files.
f	Specifies (in the second argument) a JAR file to process.
v	Generates verbose output.
m	Includes manifest information from the specified manifest file.
0	Stores class files without using Zip compression. (This option is the number zero, not the letter O.)
M	Does not create a manifest file for the entries.
u	Updates an existing JAR file by adding files or changing the manifest.
i	Generates index information for the specified JAR file.
-C	Temporarily changes directories during execution of the **jar** command.

The jar tool adds all the files in a particular directory to an archive (overwriting contents if the archive already exists). For example, the **dir** command lists the following files in a given directory:

```
Directory of C:\DEV\java\jdk1.4\myPrograms\test1\documentation

10/12/2001  08:34p    <DIR>          .
10/12/2001  08:34p    <DIR>          ..
10/12/2001  08:43p             4,461 TruckSale.java
10/12/2001  08:44p             1,965 Sale.java
10/12/2001  08:45p             3,542 overview-tree.html
```

```
10/12/2001    08:45p                 4,962 index-all.html
10/12/2001    08:45p                 3,331 deprecated-list.html
10/12/2001    08:45p                   616 allclasses-frame.html
10/12/2001    08:45p                   672 index.html
10/12/2001    08:45p                   664 packages.html
10/12/2001    08:45p                12,906 TruckSale.html
10/12/2001    08:45p                     0 package-list
10/12/2001    08:45p                 6,572 help-doc.html
10/12/2001    08:45p                 1,268 stylesheet.css
              12 File(s)             40,959 bytes
               2 Dir(s)      25,517,129,728 bytes free
```

You could then run the **jar** command in this directory:

```
jar cvf truckSale.jar *
```

The output from this command would look like this:

```
added manifest
adding: allclasses-frame.html(in = 616) (out= 391)(deflated 36%)
adding: deprecated-list.html(in = 3331) (out= 743)(deflated 77%)
adding: help-doc.html(in = 6572) (out= 2005)(deflated 69%)
adding: index-all.html(in = 4962) (out= 1133)(deflated 77%)
adding: index.html(in = 672) (out= 420)(deflated 37%)
adding: overview-tree.html(in = 3542) (out= 817)(deflated 76%)
adding: package-list(in = 0) (out= 0)(stored 0%)
adding: packages.html(in = 664) (out= 352)(deflated 46%)
adding: Sale.java(in = 1965) (out= 826)(deflated 57%)
adding: stylesheet.css(in = 1268) (out= 438)(deflated 65%)
adding: TruckSale.html(in = 12906) (out= 2585)(deflated 79%)
adding: TruckSale.java(in = 4461) (out= 1421)(deflated 68%)
```

Now the truckSale.jar file contains all the files in that directory in a compressed format.

javap—The Class Disassembler

If you want to get a snapshot of a class's members and methods, the javap tool is a quick way to get it. Suppose you have the following class:

```
import java.io.*;
import java.net.*;
```

```
public class WhatTimeIsIT
{
    public static void main(String[] args) throws Exception
    {
   String date = "", time = "", html = "";
   int dateStart = 6, dateEnd = 15;
   int timeStart = 15, timeEnd = 23;
    String url_NIST_clock = "http://www.132.163.4.101:14/";

   URL nistClock = new URL(url_NIST_clock);
   BufferedReader page = new BufferedReader(
            new InputStreamReader( nistClock.openStream() ) );

   StringBuffer pageBuffer = new StringBuffer();
   while ((html = page.readLine()) != null)
    {
          pageBuffer.append(html);
    }

    page.close();

    html = pageBuffer.toString();

   date = html.substring(dateStart, dateEnd);
   time = html.substring(timeStart, timeEnd);

    System.out.println(html);
    System.out.println(date);
    System.out.println(time);
 }
}
```

Running the command **javap WhatTimeIsIT** will yield this:

```
Compiled from WhatTimeIsIT.java
```

```
public class WhatTimeIsIT extends java.lang.Object {
    public  WhatTimeIsIT();
    public static void main(java.lang.String[])
                                    throws  java.lang.Exception;
}
```

This code works for any class file—even if you don't have the source code. The javap program can also dump out the bytecodes for a class, but this is difficult to interpret (see Appendix C).

jdb—The Java Debugger

The Sun JDK provides jdb, the Java debugger. This debugger is rather primitive and works only with the command line. Most programmers don't like jdb and with good reason. Fortunately, Sun provides hooks for debugging programs, and there are many others from which to choose.

If you use a large-scale IDE, it probably supports debugging right in its own environment. If you just use the command-line tools, you might want to look at the JSwat debugger at http://www.bluemarsh.com/java/jswat/.

Appendix B: References

Many Web sites discuss Java. The following is a collection of our favorites. We wanted to do more than just list a bunch of URLs, so we grouped them to make it easier to find sites of interest. Of course, links come and go, so if you find a broken link you'll just have to keep looking.

Table B.1: Sun tutorials.

Topic	Link
Getting Started	http://www.java.sun.com/docs/books/tutorial/getStarted/index.html
Basic Language	http://www.java.sun.com/docs/books/tutorial/java/index.html
Applets	http://www.java.sun.com/docs/books/tutorial/applet/index.html
Classes	http://www.java.sun.com/docs/books/tutorial/essential/index.html
Exceptions	http://www.java.sun.com/docs/books/tutorial/essential/exceptions/index.html
Threads	http://www.java.sun.com/docs/books/tutorial/essential/threads/index.html
I/O	http://www.java.sun.com/docs/books/tutorial/essential/io/index.html
Networking	http://www.java.sun.com/docs/books/tutorial/networking/index.html
Swing	http://www.java.sun.com/docs/books/tutorial/uiswing/index.html
Collections	http://www.java.sun.com/docs/books/tutorial/collections/index.html
Internationalization	http://www.java.sun.com/docs/books/tutorial/i18n/index.html

Table B.1: Sun tutorials.

Topic	Link
2D Graphics	http://www.java.sun.com/docs/books/tutorial/2d/index.html
Sound	http://www.java.sun.com/docs/books/tutorial/sound/index.html
JavaBeans	http://www.java.sun.com/docs/books/tutorial/javabeans/index.html
JDBC	http://www.java.sun.com/docs/books/tutorial/jdbc/index.html
RMI	http://www.java.sun.com/docs/books/tutorial/rmi/index.html
IDL	http://www.java.sun.com/docs/books/tutorial/idl/index.html
Servlets	http://www.java.sun.com/docs/books/tutorial/servlets/index.html
Security	http://www.java.sun.com/docs/books/tutorial/security1.2/index.html
JAR	http://www.java.sun.com/docs/books/tutorial/jar/index.html
Extensions	http://www.java.sun.com/docs/books/tutorial/ext/index.html
Java Native Interface	http://www.java.sun.com/docs/books/tutorial/native1.1/index.html
Reflection	http://www.java.sun.com/docs/books/tutorial/reflect/index.html
Security	http://www.java.sun.com/docs/books/tutorial/security1.1/api/index.html
Comprehensive	http://www.java.sun.com/docs/books/tutorial/together/index.html

Team-Fly®

Table B.2: Sun questions and exercises.

Topic	Link
Beginning	http://www.java.sun.com/docs/books/tutorial/getStarted/QandE/questions.html
Basic OOP	http://www.java.sun.com/docs/books/tutorial/java/concepts/QandE/questions.html
Variables	http://www.java.sun.com/docs/books/tutorial/java/nutsandbolts/QandE/questions_variables.html
Operators	http://www.java.sun.com/docs/books/tutorial/java/nutsandbolts/QandE/questions_operators.html
Strings	http://www.java.sun.com/docs/books/tutorial/java/data/QandE/characters-questions.html
Numbers	http://www.java.sun.com/docs/books/tutorial/java/data/QandE/numbers-questions.html
Arrays	http://www.java.sun.com/docs/books/tutorial/java/data/QandE/arrays-questions.html
Creating Classes	http://www.java.sun.com/docs/books/tutorial/java/javaOO/QandE/creating-questions.html
Managing Inheritance	http://www.java.sun.com/docs/books/tutorial/java/javaOO/QandE/inherit-questions.html
Nested Classes	http://www.java.sun.com/docs/books/tutorial/java/javaOO/QandE/nested-questions.html
Interfaces	http://www.java.sun.com/docs/books/tutorial/java/interpack/QandE/interfaces-questions.html
Packages	http://www.java.sun.com/docs/books/tutorial/java/interpack/QandE/packages-questions.html

Table B.2: Sun questions and exercises.

Topic	Link
Exceptions	http://www.java.sun.com/docs/books/tutorial/essential/exceptions/QandE/questions.html
Threads	http://www.java.sun.com/docs/books/tutorial/essential/threads/QandE/questions.html
I/O	http://www.java.sun.com/docs/books/tutorial/essential/io/QandE/questions.html
J2EE	http://www.java.sun.com/j2ee/tutorial/index.html
3D API	http://www.java.sun.com/products/java-media/3D/collateral
XML	http://www.java.sun.com/xml/tutorial_intro.html
JNDI (Java Naming and Directory Interface)	http://www.java.sun.com/products/jndi/tutorial
JDC (Java Developer Connection)	http://www.developer.java.sun.com/developer/onlineTraining
Swing	http://www.java.sun.com/products/jfc/tsc

Table B.3: Sun articles with sample code.

Topic	Link
Caching	http://www.java.sun.com/developer/technicalArticles/ALT/cachingservices/index.html
Collections	http://www.java.sun.com/developer/technicalArticles/Collections/Using/index.html
JDBC Order	http://www.developer.java.sun.com/developer/technicalArticles/Database/dukesb

Table B.3: Sun articles with sample code.

Topic	Link
Entry	akery
Java Reflection	http://www.java.sun.com/developer/technicalArticles/ALT/Reflection/index.html
Color	http://www.java.sun.com/developer/technicalArticles/Media/Simple2D/index.html
Localization	http://www.developer.java.sun.com/developer/technicalArticles/Intl/IntlIntro/index.html
XML JavaBeans Integration	http://www.java.sun.com/developer/technicalArticles/jbeans/XMLJavaBeans3/index.html
J2EE Client	http://www.developer.java.sun.com/developer/technicalArticles/J2EE/appclient
Tokenization	http://www.developer.java.sun.com/developer/technicalArticles/Programming/stringtokenizer
Embedding Java into Your Native Apps	http://www.developer.java.sun.com/developer/technicalArticles/Programming/embedjava/index.html
Javadoc	http://www.developer.java.sun.com/developer/technicalArticles/Programming/Javadoc
Java Source Parser	http://www.developer.java.sun.com/developer/technicalArticles/Parser/SeriesPt4/index.html
2D Printing	http://www.developer.java.sun.com/developer/technicalArticles/Printing/Java2DPrinting/index.html
Serialization	http://www.developer.java.sun.com/developer/technicalArticles/RMI/ObjectPersist/index.html
Servlets and Serialization and	http://www.developer.java.sun.com/developer/technicalArticles/RMI/rmi/

Table B.3: Sun articles with sample code.

Topic	Link
RMI	
Servlet	http://www.developer.javasun.com/developer/technicalArticles/Servlets/JavaServerTech1/index.html
Security	http://www.developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/abstract.html
Custom I/O Stream	http://www.developer.java.sun.com/developer/technicalArticles/Streams/WritingIOSC/index.html
Writing a Web Crawler	http://www.developer.java.sun.com/developer/technicalArticles/ThirdParty/WebCrawler/index.html
XML and JSP	http://www.developer.java.sun.com/developer/technicalArticles/xml/WebAppDev2/?frontpage-jdc
XML and Databases	http://www.developer.java.sun.com/developer/technicalArticles/xml/api/index.html
XML	http://www.developer.java.sun.com/developer/technicalArticles/xml/mapping
XML and DB and Cryptography	http://www.developer.java.sun.com/developer/technicalArticles/xml/metadata
Reporting a Java Bug	http://www.java.sun.com/cgi-bin/bugreport.cgi

[redacted]

[redacted]

Table B.4: Sun quizzes.

Topic	Link
Objects	http://www.java.sun.com/jdc/Quizzes/BegJavaObjects

Table B.4: Sun quizzes.

Topic	Link
JavaMail	http://www.java.sun.com/jdc/Quizzes/JavaMail
Advanced Programming	http://www.developer.java.sun.com/developer/Quizzes/advancedprogramming
Basic Language	http://www.developer.java.sun.com/developer/Quizzes/langessentials
Certification	http://www.developer.java.sun.com/developer/Quizzes/certification.html
Graphical User Interface (GUI)	http://www.developer.java.sun.com/developer/Quizzes/gui.html
JavaServer Pages Technology	http://www.developer.java.sun.com/developer/Quizzes/javaserverpages.html
JDBC	http://www.developer.java.sun.com/developer/Quizzes/jdbctechnology.html
Networking	http://www.developer.java.sun.com/developer/Quizzes/networking.html
Performance and Debugging	http://www.developer.java.sun.com/developer/Quizzes/performancedebugging.html
Remote Method Invocation (RMI)	http://www.developer.java.sun.com/developer/Quizzes/rmi.html

Table B.4: Sun quizzes.

Topic	Link
Security	http://www.developer.java.sun.com/developer/Quizzes/security.html
Wireless Technology	http://www.developer.java.sun.com/developer/Quizzes/wirelesstechnologies.html
Enterprise JavaBeans	http://www.developer.java.sun.com/developer/Quizzes/ejbtechnology.html

Table B.5: Sun certification.

Title	Link
Sun Certified Programmer for 2 Platform	http://www.suned.sun.com/US/certification/java/java_progj2se.html
Sun Certified Developer for Java 2 Platform	http://www.sun ed.sun.com/US/certification/java/java_devj2se.html
Sun Certified Enterprise Architect for Java 2 Platform, Enterprise Edition Technology	http://www.suned.sun.com/US/certification/java/java_archj2ee.html
Sun Certified Web Component Developer for the J2EE Platform	http://www.suned.sun.com/US/certification/java/java_web.html

Table B.6: Sun instructor-led courses.

Java Technology Core Learning Path

Title	Link
Java Programming Language for Non-Programmers (SL-110)	http://www.suned.sun.com/US/catalog/courses/SL-110.html
Migrating to Object-Oriented Programming with Java Technology (SL-210)	http://www.suned.sun.com/US/catalog/courses/SL-210.html
Java Technology for Structured Programmers (SL-265)	http://www.suned.sun.com/US/catalog/courses/SL-265.html
Java Programming Language (SL-275)	http://www.suned.sun.com/US/catalog/courses/SL-275.html
Object-Oriented Analysis and Design for Java Technology (UML) (OO-226)	http://www.suned.sun.com/US/catalog/courses/OO-226.html
Java	http://www.suned.sun.com/US/catalog/courses/SL-285.html

Java Technology Core Learning Path

Title	Link
Programming Language Workshop (SL-285)	

Enterprise Developer Learning Path

Title	Link
Java Programming Language Workshop (SL-285)	http://www.suned.sun.com/US/catalog/courses/SL-285.html
Developing J2EE Compliant Enterprise Java Applications (FJ-310)	http://www.suned.sun.com/US/catalog/courses/FJ-310.html
Developing J2EE Compliant Applications (iPlanet)	http://www.suned.sun.com/US/catalog/courses/FJ-311.html

Enterprise Developer Learning Path

Title	Link
(FJ-311)	
Web Component Development Using Java Technology (SL-314)	http://www.suned.sun.com/US/catalog/courses/SL-314.html
Database Application Programming with Java Technology (SL-330)	http://www.suned.sun.com/US/catalog/courses/SL-330.html
Enterprise JavaBeans Programming (SL-351)	http://www.suned.sun.com/US/catalog/courses/SL-351.html

Enterprise Architect Learning Path

Title	Link
Java	http://www.suned.sun.com/US/catalog/courses/SL-110.html

Enterprise Architect Learning Path

Title	Link
Programming Language for Non-Programmers (SL-110)	
Object-Oriented Analysis and Design for Java Technology (UML) (OO-226)	http://www.suned.sun.com/US/catalog/courses/OO-226.html
Java 2, Enterprise Edition: Technology Overview Seminar (SEM-SL-345)	http://www.suned.sun.com/US/catalog/courses/SEM-SL-345.html
Developing J2EE Compliant Enterprise Java Applications (FJ-310)	http://www.suned.sun.com/US/catalog/courses/FJ-310.html
Developing J2EE Compliant Applications (iPlanet) (FJ-311)	http://www.suned.sun.com/US/catalog/courses/FJ-311.html

TEAMFLY

Team-Fly®

Enterprise Architect Learning Path

Title	Link
Java 2 Enterprise Edition Patterns (SL-500)	http://www.suned.sun.com/US/catalog/courses/SL-500.html
Architecting and Designing J2EE Applications (SL-425)	http://www.suned.sun.com/US/catalog/courses/SL-425.html
Java 2 Enterprise Edition: Application Programming Model (SEM-SL-425)	http://www.suned.sun.com/US/catalog/courses/SEM-SL-425.html

Fast Track Courses

Title	Link
Java Technology Fast Track Program I (inclusive) (JF-341)	http://www.suned.sun.com/US/catalog/courses/JF-341.html

Fast Track Courses

Title	Link
Java Technology Fast Track Program I (non-inclusive) (JB-341)	http://www.suned.sun.com/US/catalog/courses/JB-341.html
Java Technology Fast Track Program II (inclusive) (JF-441)	http://www.suned.sun.com/US/catalog/courses/JF-441.html
Java Technology Fast Track Program II (non-inclusive) (JB-441)	http://www.suned.sun.com/US/catalog/courses/JB-441.html

Other Courses

Title	Link
Developing J2ME Mobile	http://www.suned.sun.com/US/catalog/courses/DTJ-360.html

Other Courses

Title	Link
Applications with CLDC and the MID Profile (DTJ-360)	
Introduction to CORBA using C++ and Java Programming (SI-330)	http://www.suned.sun.com/US/catalog/courses/SI-330.html
JavaBeans Component Development (SL-291)	http://www.suned.sun.com/US/catalog/courses/SL-291.html
Distributed Programming with Java Technology (SL-301)	http://www.suned.sun.com/US/catalog/courses/SL-301.html
Implementing Java Security (SL-303A)	http://www.suned.sun.com/US/catalog/courses/SL-303A.html
Beyond CGI: Developing Java Servlets (SL-310)	http://www.suned.sun.com/US/catalog/courses/SL-310.html
Java Server Pages	http://www.suned.sun.com/US/catalog/courses/SL-315.html

Other Courses

Title	Link
(SL-315)	
GUI Construction with Java Foundation Classes (SL-320)	http://www.suned.sun.com/US/catalog/courses/SL-320.html
Programming Distributed Services with Jini Technology (SL-370)	http://www.suned.sun.com/US/catalog/courses/SL-370.html
Managing XML with the Java Platform (SL-385)	http://www.suned.sun.com/US/catalog/courses/SL-385.html
Programming Distributed Management Systems with Jiro Technology (SL-450)	http://www.suned.sun.com/US/catalog/courses/SL-450.html

Table B.7: Sun CD-ROM courses.

Title	Link
Distributed	http://www.sun ed.sun.com/US/catalog/courses/JT-SL301.html

Table B.7: Sun CD-ROM courses.

Title	Link
Programming with Java Technology, Java 2 Platform (JT-SL301)	
Java Programming for Non-Programmers Library (Java 2 Platform) (JTL-SL110)	http://www.suned.sun.com/US/catalog/courses/JTL-SL110.html
Java Programming for Non-Programmers Library (Java 2 Platform)—5-User (JTL-SL110-05)	http://www.suned.sun.com/US/catalog/courses/JTL-SL110-05.html
Java Programming Language Library (Java 2 Platform) (JTL-SL275-2)	http://www.suned.sun.com/US/catalog/courses/JTL-SL275-2.html
Java Programming Language Library (Java 2 Platform)—5-User	http://www.suned.sun.com/US/catalog/courses/JTL-SL275-2-05.html

Table B.7: Sun CD-ROM courses.

Title	Link
(JTL-SL275-2-05)	
Distributed Programming Using Java JDK 1.1 Library—1-User (JTL-SL301)	http://www.suned.sun.com/US/catalog/courses/JTL-SL301.html
Distributed Programming Using Java JDK 1.1 Library—5-User (JTL-SL301-05)	http://www.suned.sun.com/US/catalog/courses/JTL-SL301-05.html
Implementing Java Security for JDK 1.2 Library—1-User (JTL-SL303)	http://www.suned.sun.com/US/catalog/courses/JTL-SL303.html
Implementing Java Security for JDK 1.2 Library—5-User (JTL-SL303-05)	http://www.suned.sun.com/US/catalog/courses/JTL-SL303-05.html

Table B.8: Sun Web-based courses.

Core Learning Suite

Title	Link
Introduction to	http://www.suned.sun.com/us/catalog/courses/WJ-1101-90.html

Core Learning Suite

Title	Link
the Java Programming Language	
Getting Started with the Java Programming Language	http://www.suned.sun.com/us/catalog/courses/WJ-1102-90.html
Basics of the Java Programming Language	http://www.suned.sun.com/us/catalog/courses/WJ-1103-90.html
Object-Oriented Functionality and the Java Programming Language	http://www.suned.sun.com/us/catalog/courses/WJ-1104-90.html
Inheritance and Advanced Java Technology Concepts	http://www.suned.sun.com/us/catalog/courses/WJ-1105-90.html
Migrating to Object-Oriented Programming	http://www.suned.sun.com/us/catalog/courses/WP-1201-90.html
Implementing Object-Oriented Programming with Java Technology	http://www.suned.sun.com/us/catalog/courses/WP-1202-90.html

Core Learning Suite

Title	Link
Advanced Object-Oriented Programming	http://www.suned.sun.com/us/catalog/courses/WP-1203-90.html
Getting Started with the Java Programming Language	http://www.suned.sun.com/us/catalog/courses/WJ-2501-90.html
Java Programming Language Basics	http://www.suned.sun.com/us/catalog/courses/WJ-2502-90.html
Using Objects and Exceptions in the Java 2 Platform	http://www.suned.sun.com/us/catalog/courses/WJ-2503-90.html
Using Applets, GUIs, and Java AWT	http://www.suned.sun.com/us/catalog/courses/WJ-2504-90.html
Using the Java Streams, Files, and Threads	http://www.suned.sun.com/us/catalog/courses/WJ-2505-90.html
Java Technology Networking, Distributed Computing, and	http://www.suned.sun.com/us/catalog/courses/WJ-2506-90.html

Core Learning Suite

Title	Link
Foundation Classes	
Introduction to the Java Programming Language	http://www.suned.sun.com/us/catalog/courses/WJ-2601-90.html
Base Class Design	http://www.suned.sun.com/us/catalog/courses/WJ-2602-90.html
Advanced Class Design	http://www.suned.sun.com/us/catalog/courses/WJ-2603-90.html
Graphical User Interface Design	http://www.suned.sun.com/us/catalog/courses/WJ-2604-90.html
Stream I/O, Networking, and Threads	http://www.suned.sun.com/us/catalog/courses/WJ-2605-90.html
Programming with the Java 3D API: A Technical Overview	http://www.suned.sun.com/us/catalog/courses/WJ-6701-90.html

Distributed Technologies Learning Suite

Title	Link
Distributed Computing Concepts and Techonology	http://www.suned.sun.com /us/catalog/courses/WJ-3201-90.html
Distributed Java Platform Technologies: JDBC, RMI, IDL	http://www.suned.sun.com/us/catalog/courses/WJ-3202-90.html
Java Platform Server Technologies: JNDI, JMS, JTS	http://www.suned.sun.com/us/catalog/courses/WJ-3203-90.html
Jiro Technology Overview	http://www.suned.sun.com/us/catalog/courses/WJ-5550-90.html
Getting Started With Jiro Technology	http://www.suned.sun.com/us/catalog/courses/WJ-5551-90.html

Enterprise Developer Learning Suite

Title	Link

TEAMFLY

Team-Fly®

Enterprise JavaBeans Overview	http://www.suned.sun.com/us/catalog/courses/WJ-3301-90.html
Understanding Servlets	http://www.suned.sun.com/us/catalog/courses/WJ-3050-90.html
Programming Java-Based Servlets	http://www.suned.sun.com/us/catalog/courses/WJ-3051-90.html

Consumer Devices

Title	Link
An Introduction to J2ME and the MID Profile	http://www.suned.sun.com/us/catalog/courses/WJ-4500-90.html
What Is Java Card Technology?	http://www.suned.sun.com/us/catalog/courses/WJ-4601-90.html
Jini Technology Bundle	http://www.suned.sun.com/us/catalog/courses/WJB -550-180.html
What Is Jini Technology?	http://www.suned.sun.com/us/catalog/courses/WJ-5501-90.html
Getting	http://www.suned.sun.com/us/catalog/courses/WJ-5502-90.html

Consumer Devices

Title	Link
Started with Jini Technology	

Table B.9: Quick links to Java SDK features.

CORE Features

Title	Link
Version Compatibility with Previous Releases	http://www.java.sun.com/j2se/1.4/compatibility.html
Fixed Bugs	http://www.java.sun.com/j2se/1.4/fixedbugs/BugIndex.html
Java 2 SDK Download Page	http://www.java.sun.com/j2se/1.4/index.html
Java 2 Platform API Specification	http://www.java.sun.com/j2se/1.4/docs/api/index.html
The Java Language Specification	http://www.java.sun.com/docs/books/jls/html/index.html
The Java Virtual Machine Specification	http://www.java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html
Virtual Machine	http://www.java.sun.com/j2se/1.4/docs/guide/vm/index.htmljava.sun.com/j2se/1.4/docs/guid

CORE Features

Title	Link
Security and Signed Applets	http://www.java.sun.com/j2se/1.4/docs/guide/security/index.html
Collections Framework	http://www.java.sun.com/j2se/1.4/docs/guide/collections/index.html
JavaBeans Component API	http://www.java.sun.com/j2se/1.4/docs/guide/beans/index.html
Internationalization	http://www.java.sun.com/j2se/1.4/docs/guide/intl/index.html
New I/O	http://www.java.sun.com/j2se/1.4/docs/guide/nio/index.html
I/O	http://www.java.sun.com/j2se/1.4/docs/guide/io/index.html
XML	http://www.java.sun.com/j2se/1.4/docs/guide/xml/index.html
Networking	http://www.java.sun.com/j2se/1.4/docs/guide/net/index.html
Language and Utility Packages	http://www.java.sun.com/j2se/1.4/docs/guide/lang/index.html
Logging	http://www.java.sun.com/j2se/1.4/docs/guide/util/logging/index.html
Remote Method Invocation (RMI)	http://www.java.sun.com/j2se/1.4/docs/guide/rmi/index.html
Arbitrary-Precision Math	http://www.java.sun.com/j2se/1.4/docs/guide/math/index.html

CORE Features

Title	Link
Reflection	http://www.java.sun.com/j2se/1.4/docs/guide/reflection/index.html
Package Version Identification	http://www.java.sun.com/j2se/1.4/docs/guide/versioning/index.html
Sound	http://www.java.sun.com/j2se/1.4/docs/guide/sound/index.html
Reference Objects	http://www.java.sun.com/j2se/1.4/docs/guide/refobs/index.html
Resources	http://www.java.sun.com/j2se/1.4/docs/guide/resources/index.html
Object Serialization	http://www.java.sun.com/j2se/1.4/docs/guide/serialization/index.html
Extension Mechanism	http://www.java.sun.com/j2se/1.4/docs/guide/extensions/index.html
Java Archive (JAR) Files	http://www.java.sun.com/j2se/1.4/docs/guide/jar/index.html
Java Native Interface (JNI)	http://www.java.sun.com/j2se/1.4/docs/guide/jni/index.html
Performance Enhancements	http://www.java.sun.com/j2se/1.4/docs/guide/performance/index.html
Miscellaneous Features (Applet tag, Deprecation)	http://www.java.sun.com/j2se/1.4/docs/guide/misc/index.html

Java Foundation Classes (JFC)

Topic	Link
Abstract Window Toolkit (AWT)	http://www.java.sun.com/j2se/1.4/docs/guide/awt/index.html
Project Swing Components	http://www.java.sun.com/j2se/1.4/docs/guide/swing/index.html
2D Graphics and Imaging	http://www.java.sun.com/j2se/1.4/docs/guide/2d/index.html
Image I/O	http://www.java.sun.com/j2se/1.4/docs/guide/imageio/index.html
Print Service	http://www.java.sun.com/j2se/1.4/docs/guide/jps/index.html
Input Method Framework	http://www.java.sun.com/j2se/1.4/docs/guide/imf/index.html
Accessibility	http://www.java.sun.com/j2se/1.4/docs/guide/access/index.html
Drag-and-Drop data transfer	http://www.java.sun.com/j2se/1.4/docs/guide/dragndrop/index.html

Enterprise Features

Topic	Link
RMI-IIOP	http://www.java.sun.com/j2se/1.4/docs/guide/rmi-iiop/index.html

Enterprise Features

Topic	Link
Java IDL	http://www.java.sun.com/j2se/1.4/docs/guide/idl/index.html
CORBA	http://www.java.sun.com/j2se/1.4/docs/guide/corba/index.html
JDBCTM (Java Database Connectivity)	http://www.java.sun.com/j2se/1.4/docs/guide/jdbc/index.html
Java Naming and Directory InterfaceTM (JNDI)	http://www.java.sun.com/j2se/1.4/docs/guide/jndi/index.html

Deployment

Topic	Link
Java Plug-in	http://www.java.sun.com/j2se/1.4/docs/guide/plugin/index.html
Java Web Start	http://www.java.sun.com/j2se/1.4/docs/guide/jws/index.html

Tool Support

Topic	Link
Java Platform Debugger Architecture (JPDA)	http://www.java.sun.com/j2se/1.4/docs/guide/jpda/index.html
Java Virtual Machine Profiler Interface (JVMPI)	http://www.java.sun.com/j2se/1.4/docs/guide/jvmpi/index.html

Table B.10: API and language documentation.

Topic	Link
Java 2 SDK FAQ	http://www.java.sun.com/products/jdk/faq.html
Java 2 Platform API Specification	http://www.java.sun.com/j2se/1.4/docs/api/index.html
Note about sun.* Packages	http://www.java.sun.com/products/jdk/faq/faq-sun-packages.html
The Java Languag	http://www.java.sun.com/docs/books/jls/html/index.html

Table B.10: API and language documentation.

Topic	Link
e Specifica tion	
The Java Virtual Machine Specifica tion	http://www.java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

Table B.11: Online Java books.

Title	Link
Essentials of the Java Programming Language, Hands-On Guide	http://www.developer.java.sun.com/developer/onlineTraining/Programming/BasicJava1/index.html
A Java GUI Programmer's Primer	http://www.scism.sbu.ac.uk/jfl/jibook/jicontents.html
Advanced Programming for the Java 2 Platform	http://www.developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/index.html
Computer Science Java Style	http://www.g.oswego.edu/%7Eblue/java/hyperbook/org/Cover.html
Cooking with Beans in the Enterprise	http://www.redbooks.ibm.com/SG247006/toc.htm
Developing Intranet	http://www.docs.rinet.ru:8080/Jintra/

Table B.11: Online Java books.

Title	Link
Applications Using Java	
Introduction to Programming Using Java	http://www.javafaq.nu/java/free-introduction-java-book/index.shtml
Jan Newmarch's Guide to JINI Technologies	http://www.pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml
Java: An Object-First Approach	http://www.scism.sbu.ac.uk/jfl/jflcontents.html
Java Code Convention	http://www.docs.rinet.ru:8080/codeconv
Java Developer's Reference	http://www.docs.rinet.ru:8080/JavDev
Java Expert Solution	http://www.docs.rinet.ru:8080/JSol
Securing Java on the Web	http://www.securingjava.com/toc.html
Teach Yourself Java 1.1 Programming in 24 Hours	http://www.docs.rinet.ru:8080/J11/index.htm
Teach Yourself Java in 21 Days	http://www.docs.rinet.ru:8080/J21
Thinking in Java	http://www.bruceeckel.com/TIJ2/index.html

Table B.12: Javacats lists of Java resources.

Topic	Link
Tools	http://www.javacats.com/US/tools
Books	http://www.javacats.com/US/books
Reference	http://www.javacats.com/US/reference

Table B.13: Miscellaneous resources.

Topic	Link
Java Developer Connection	http://www.developer.java.sun.com
Code Conventions	http://www.java.sun.com/docs/codeconv
Code Test Guide	http://www.jcp.org/aboutJava/communityprocess/speclead/tck/tsdg-10.pdf
About.com Focus on Java	http://www.java.about.com
"JavaWorld" Magazine	http://www.javaworld.com
Java Boutique (applet stuff)	http://www.javaboutique.internet.com
"Java Developer's Journal (best Java magazine)"	http://www.sys-con.com/java

TEAMFLY

Team-Fly®

Table B.13: Miscellaneous resources.

Topic	Link
Swing FAQs	http://www.mindspring.com/~scdrye/java/faq.html
Mac developers	http://www.rain.org/~da5e/macjava.html
Slashdot	http://www.slashdot.org/search.pl?topic=java
Coding tournament	http://www.topcoder.com
IBM's Java	http://www.www-106.ibm.com/developerworks/java
IBM's AlphaWorks	http://www.alphaworks.ibm.com
Apache's open source	http://www.xml.apache.org
Outstanding developer site	http://www.jguru.com
Regular Expressions	http://www.developer.java.sun.com/developer/technicalArticles/releases/1.4regex/
Exact Calculations with Floating-Point	http://www.java.sun.com/jdc/JDCTechTips/2001/tt0807.html
Certification FAQs	http://www.developer.java.sun.com/developer/technicalArticles/Interviews/Certification2/index.html

Table B.13: Miscellaneous resources.

Topic	Link
Question of the Week	http://www.developer.java.sun.com/developer/qow/archive
All Tech Tips	http://www.developer.java.sun.com/developer/JDCTechTips
HotSpot	http://www.java.sun.com/products/hotspot
Glossary of Java and Related Terms	http://www.java.sun.com/docs/glossary.print.html
Sun Trademark and Logo Usage Requirements	http://www.sun.com/policies/trademarks
jguru	http://www.jguru.com/jguru/faq
UML diagrams for the core Java packages	http://www.javareport.com/java2interactive/index.html
UML Resource Page	http://www.omg.org/uml

Appendix C: The Java Virtual Machine

Overview

You don't have to know how a car engine works in order to drive a car. Still, the best drivers do know how their engines work, and this knowledge often helps them coax that last ounce of performance from their cars. Understanding the Java Virtual Machine (JVM), its place in Java's architecture, and how it works will help you write code that takes advantage of the JVM and avoids its weaknesses.

The JVM is a make-believe CPU that executes a special machine language (bytecodes) tailored for Java. Although it is possible to make a hardware Java CPU (and some exist), most computers use a CPU chip that doesn't understand Java bytecodes. Therefore, a machine-specific program—the JVM—interprets the Java bytecodes. The Java compiler creates a file that is JVM-specific instead of machine-specific. As long as your machine has the correct JVM, that machine can run any Java program.

Many versions of the JVM are available from a variety of vendors. Some JVMs interpret each bytecode on a case-by-case basis. Others use a justin-time strategy to compile the bytecodes to native machine language instructions upon execution. This strategy takes a little longer to set up than does the simple interpreter, but the strategy should result in faster execution over the long run.

JVM Architecture

Each vendor can build its own JVM, but it must comply with the following architectural criteria:

- *PC register*—This register is a program counter used by each JVM thread; this counter executes and indicates the current instruction that each thread is executing (unless the thread is executing a native method). The JVM's PC register is one word wide, the width guaranteed to hold a **returnAddress** or a native pointer on the specific platform.
- *Java stack*—Each JVM thread has a private stack, created at the same time as the thread. A Java stack stores JVM frames. Each frame holds local variables and plays a part in method invocation and return. Because the stack is never manipulated directly except to push and pop frames, the JVM might actually implement the stack as a heap, and Java frames might be heap allocated. The memory for a Java stack does not need to be contiguous. A JVM implementation might give the programmer or the user control over the initial size of Java stacks, as well as, in the case of dynamically expanding or contracting Java stacks, control over the maximum and minimum sizes of Java stacks.
- *Heap*—A heap is a memory area shared among all threads. The heap is the runtime data area from which memory for all class instances, variables, and arrays is allocated. The Java heap is created on virtual machine startup. Heap storage for objects is reclaimed by the garbage collector.
- *Method area*—The method area is created on virtual machine startup and is shared among all threads. It stores per-class structures such as the constant pool, field and method data, and the code for methods and constructors.

- *Constant pool*—A constant pool is a per-class table of constants ranging from numeric literals known at compile time to method and field references that must be resolved at runtime. Each constant pool resides in the method area.
- *Native method stacks*—The native method stack is a conventional stack used to support native methods—that is, methods written in conventional languages such as C or written in assembly language.
- *Frames*—Frames store local variables, partial results, return values for methods, and exception information. A new frame is created each time a method invocation occurs.

JVM Instructions

A JVM instruction consists of a one-byte opcode specifying the operation to be performed, followed by zero or more operands supplying arguments or data that are used by the operation. Many instructions have no operands and consist only of an opcode. The steps taken by the typical JVM are:

1. Fetch an opcode.
2. Fetch operands (if any).
3. Execute the corresponding action.
4. Repeat steps 1 through 3.

For the majority of instructions that deal with data types, the data type is represented explicitly in the opcode mnemonic by a letter: **i** for an **int** operation, **l** for **long**, **s** for **short**, **b** for **byte**, **c** for **char**, **f** for **float**, **d** for **double**, and **a** for reference types. For example, the **iload** instruction loads the contents of a local variable, which must be an **int**, onto the operand stack. The **fload** instruction does the same with a **float** value. The load actions are summarized as follows:

- *Load a local variable onto the operand stack*—**iload**, **iload_<n>**, **lload**, **lload_<n>**, **fload**, **fload_<n>**, **dload**, **dload_<n>, aload**, **aload_<n>**
- *Store a value from the operand stack into a local variable*— **istore**, **istore_<n>**, **lstore**, **lstore_<n>**, **fstore**, **fstore_<n>**, **dstore**, **dstore_<n>**, **astore**, **astore_<n>**
- *Load a constant onto the operand stack*—**bipush**, **sipush**, **ldc**, **ldc_w**, **ldc2_w**, **aconst_null**, **iconst_m1**, **iconst_<i>**, **lconst_<l>**, **fconst_<f>**, **dconst_<d>**
- *Gain access to more local variables using a wider index, or to a larger immediate operand*—**wide**

Similarly, the arithmetic and logic instructions are as follows:

- *Add*—**iadd**, **ladd**, **fadd**, **dadd**
- *Subtract*—**isub**, **lsub**, **fsub**, **dsub**
- *Multiply*—**imul**, **lmul**, **fmul**, **dmul**
- *Divide*—**idiv**, **ldiv**, **fdiv**, **ddiv**
- *Remainder*—**irem**, **lrem**, **frem**, **drem**
- *Negate*—**ineg**, **lneg**, **fneg**, **dneg**
- *Shift*—**ishl**, **ishr**, **iushr**, **lshl**, **lshr**, **lushr**
- *Bitwise OR*—**ior**, **lor**

- *Bitwise AND*—**iand**, **land**
- *Bitwise exclusive OR*—**ixor**, **lxor**
- *Local variable increment*—**iinc**

The narrowing numeric conversion instructions, for example, are **i2b** for converting an **int** to a **byte**, and **i2c** for converting an **int** to a **char**. The others are **i2s**, **l2i**, **f2i**, **f2l**, **d2i**, **d2l**, and **d2f**. Many more opcodes, such as **newarray**, are available to create a new array and **arraylength**, which returns the length of an array. You can get the whole list of them at the JVM specification at http://www.java.sun.com/docs/books/vmspec/html/Overview.doc.html.

Java Compilation

The compiler translates your source code into bytecodes. Consider this method:

```
double doubleLocals(double d1, double d2)
{
    return d1 + d2
}
```

The bytecode representation looks like this:

```
Method double doubleLocals(double,double)
0 dload_1 // First argument in locals 1 and 2
1 dload_3 // Second argument in locals 3 and 4
2 dadd          // Each also uses two words on stack
3 dreturn
```

As another example, consider this loop:

```
void whileInt()
{
   int i = 0;
   while (i < 100)
    {
        i++;
    }
}
```

When compiled, it looks like this:

```
Method void whileInt()

0   iconst_0
```

```
1   istore_1
2   goto 8
5   iinc 1 1
8   iload_1
9   bipush 100
11 if_icmplt 5
14 return
```

You can use the javap tool with the **-c** option (see Appendix A) to disassemble the bytecodes for any class.

The Class File

The compiler generates a class file. The structure of this file looks like this:

```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
      }
```

The items in the structure are as follows:

- **magic**—Identifies this file as a class file (currently 0xCAFEBABE).
- **minor_version**, **major_version**—Version numbers of this class file.
- **constant_pool_count**—Number of entries in the **constant_pool** table plus one.
- **constant_pool[]**—Array of strings, class and interface names, field names, and other constants that are referred to within the **ClassFile** structure and its substructures.
- **access_flags**—Access permissions and properties of this class. For example, **ACC_PUBLIC** (0x0001) corresponds to **public**.
- **this_class**—A valid index into the **constant_pool** table. This index refers to the current class.
- **super_class**—Another valid index into the **constant_pool** table. This index refers to the base class.
- **interfaces_count**—Number of direct superinterfaces of this class.
- **interfaces[]**—An array of valid indices into the **constant_pool** table that correspond to the interfaces for this class.
- **fields_count**—Number of **field_info** structures in the **fields** table.
- **fields[]**—Array of **field_info** structures that contains an entry for each field.
- **methods_count**—Number of **method_info** structures in the **methods** table.
- **methods[]**—Array of **method_info** structures that describe each method.
- **attributes_count**—Number of attributes in the **attributes** table.
- **attributes[]**—Array of **attribute** structures that stores information about class attributes.

Appendix D: Active RFCs

Overview

If you need detailed information about anything on the Internet, the RFCs (Requests for Comments) are the actual source documents that define practi cally everything about the Internet. Unfortunately, with thousands of RFCs available, it is difficult to know which ones you need to read. To make matters worse, many RFCs are no longer active, which can be confusing.

Some RFCs (such as RFC1000) are an index to the other RFCs. Also, online search engines, such as the one at http://www.faqs.org/rfcs , can be very helpful. You can also find a complete list at http://www.ietf.org/iesg/1rfc_index.txt, and retrieve documents from http://www.rfc-editor.org/rfc.html. You can also get RFCs via email. The email address is rfc-info@isi.edu. You can get an index of RFCs by sending an email with the following text in the message body:

HELP: rfc_index
Complete help is available with this command:
HELP: help
To retrieve a specific RFC, send:
Retrieve: RFC
Doc-ID: RFC1000

Keep in mind that not all RFCs are standards. Some of them are even humorous (check out RFC 1149, for example). The true standards documents (and their corresponding RFCs) appear in Table D.1. A few of the standards are obsolete, but they appear in the table for completeness.

Table D.1: Internet standards documents.

Standard	Title	Authors	Date	Obsoletes	RFC(s)
STD0001	Internet Official Protocol Standards	J. Reynolds, R. Braden, S. Ginoza	May 2001	RFC2700	RFC2800
STD0002	Assigned Numbers	J. Reynolds, J. Postel	Oct. 1994	RFC1340	RFC1700
STD000	Requirement	R.	Oct.		RFC1122, RFC1123

Table D.1: Internet standards documents.

Standard	Title	Authors	Date	Obsoletes	RFC(s)
3	s for Internet Hosts	Braden, Ed.	1989		
STD0004	Reserved for Router Requirements				
STD0005	Internet Protocol	J. Postel	Sep. 1981		RFC0791, RFC0792, RFC0919, RFC0922, RFC0950, RFC1112
STD0006	User Datagram Protocol	J. Postel	Aug. 1980		RFC0768
STD0007	Transmission Control Protocol	J. Postel	Sep. 1981		RFC0793
STD0008	Telnet Protocol	J. Postel, J. Reynolds	May 1983		RFC0854, RFC0855
STD0009	File Transfer Protocol	J. Postel, J. Reynolds	Oct. 1985		RFC0959, RFC2228, RFC2640
STD0010	Simple Mail Transfer Protocol	J. Postel	Aug. 1982	RFC788, RFC780, RFC772	RFC0821, RFC2821
STD0011	Standard for the Format of	D. Crocker	Aug. 1982	RFC733	RFC0822, RFC2822

Table D.1: Internet standards documents.

Standard	Title	Authors	Date	Obsoletes	RFC(s)
	ARPA RFC2822 Internet Text Messages				
STD0012	Reserved for Network Time Protocol (NTP)				
STD0013	Domain Name System	P. Mockapetris	Nov. 1987		RFC1034, RFC1035
STD0014	Obsolete: Was Mail Routing and the Domain System				
STD0015	Simple Network Management Protocol	J. Case, M. Fedor, M. Schoffstall, J. Davin	May 1990		RFC1157
STD0016	Structure of Management Information	M. Rose, K. McCloghrie	May 1990	RFC1065	RFC1155
STD0017	Management Information Base	K. McCloghrie, M. Rose	Mar. 1991	RFC1158	RFC1213

TEAMFLY

Team-Fly®

Table D.1: Internet standards documents.

Standard	Title	Authors	Date	Obsoletes	RFC(s)
STD0018	Obsolete: Was Exterior Gateway Protocol (RFC 904)				
STD0019	NetBIOS Service Protocols	NetBIOS Working Group	Mar. 1987		RFC1001, RFC1002
STD0020	Echo Protocol	J. Postel	May 1983		RFC0862
STD0021	Discard Protocol	J. Postel	May 1983		RFC0863
STD0022	Character Generator Protocol	J. Postel	May 1983		RFC0864
STD0023	Quote of the Day Protocol	J. Postel	May 1983		RFC0865
STD0024	Active Users Protocol	J. Postel	May 1983		RFC0866
STD0025	Daytime Protocol	J. Postel	May 1983		RFC0867
STD0026	Time Server Protocol	J. Postel	May 1983		RFC0868

Table D.1: Internet standards documents.

Standard	Title	Authors	Date	Obsoletes	RFC(s)
STD0027	Binary Transmission Telnet Option	J. Postel, J. Reynolds	May 1983		RFC0856
STD0028	Echo Telnet Option	J. Postel, J. Reynolds	May 1983		RFC0857
STD0029	Suppress Go Ahead Telnet Option	J. Postel, J. Reynolds	May 1983		RFC0858
STD0030	Status Telnet Option	J. Postel, J. Reynolds	May 1983		RFC0859
STD0031	Timing Mark Telnet Option	J. Postel, J. Reynolds	May 1983		RFC0860
STD0032	Extended Options List Telnet Option	J. Postel, J. Reynolds	May 1983		RFC0861
STD0033	Trivial File Transfer Protocol	K. Sollins	July 1992		RFC1350
STD0034	Replaced by STD0056				
STD003	ISO	M. Rose,	May		RFC1006

Table D.1: Internet standards documents.

Standard	Title	Authors	Date	Obsoletes	RFC(s)
5	Transport Service on Top of the TCP (Version: 3)	D. Cass	1978		
STD0036	Transmission of IP and ARP over FDDI Networks	D. Katz	Jan. 1993		RFC1390
STD0037	An Ethernet Address Resolution Protocol	David C. Plummer	Nov. 1982		RFC0826
STD0038	A Reverse Address Resolution Protocol	Ross Finlayson, Timothy Mann, Jeffrey Mogul, Marvin Theimer	June 1984		RFC0903
STD0039	Obsolete: Was BBN Report 1822 (IMP/ Host Interface)				
STD0040	Host Access Protocol Specification	Bolt Beranek and Newman	Aug. 1993		RFC0907

Table D.1: Internet standards documents.

Standard	Title	Authors	Date	Obsoletes	RFC(s)
STD0041	Standard for the Transmission of IP Datagrams over Ethernet Networks	C. Hornig	Apr. 1984		RFC0894
STD0042	Standard for the Transmission of IP Datagrams over Experimental Ethernet Networks	J. Postel	Apr. 1984		RFC0895
STD0043	Standard for the Transmission of IP Datagrams over IEEE 802 Networks	J. Postel, J.K. Reynolds	Aug. 1993	RFC0948	RFC1042
STD0044	DCN Local-Network Protocols	D.L. Mills	Aug. 1993		RFC0891
STD0045	Internet Protocol on Network	K. Hardwick, J.	Aug. 1993		RFC1044

Table D.1: Internet standards documents.

Standard	Title	Authors	Date	Obsoletes	RFC(s)
	System's HYPERchannel: Protocol Specification	Lekashman			
STD0046	Transmitting IP Traffic over ARCNET Networks	D. Provan	Aug. 1993	RFC1051	RFC1201
STD0047	Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP	J.L. Romkey	Aug. 1993		RFC1055
STD0048	Standard for the Transmission of IP Datagrams over NetBIOS Networks	L.J. McLaughlin	Aug. 1993		RFC1088
STD0049	Standard for the Transmission of 802.2 Packets over IPX Networks	L.J. McLaughlin	Aug. 1993		RFC1132

Table D.1: Internet standards documents.

Standard	Title	Authors	Date	Obsoletes	RFC(s)
STD0050	Definitions of Managed Objects for the Ethernet-like Interface Types	F. Kastenholz	July 1994	RFC1623, RFC1398	RFC1643
STD0051	The Point-to-Point Protocol (PPP)	W. Simpson, Editor	July 1994	RFC1549	RFC1661, RFC1662
STD0052	The Transmission of IP Datagrams over the SMDS Service	D. Piscitello, J. Lawrence	Mar. 1991		RFC1209
STD0053	Post Office Protocol-Version 3	J. Myers, M. Rose	May 1996	RFC1725	RFC1939
STD0054	OSPF Version 2	J. Moy	Apr. 1998		RFC2328
STD0055	Multiprotocol Interconnect over Frame Relay	C. Brown, A. Malis	Sep. 1998	RFC1490, RFC1294	RFC2427
STD0056	RIP Version 2	G. Malkin	Nov. 1998	RFC1723	RFC2453

Table D.1: Internet standards documents.

Standard	Title	Authors	Date	Obsoletes	RFC(s)
STD0057	RIP Version 2 Protocol Applicability Statement	G. Malkin	Nov. 1994		RFC1722
STD0058	Structure of Management Information Version 2 (SMIv2)	K. McCloghrie, D. Perkins, J. Schoenwaelder	Apr. 1999	RFC1902	RFC2578, RFC2579
STD0059	Remote Network Monitoring Management Information Base	S. Waldbusser	May 2000	RFC1757	RFC2819
STD0060	SMTP Service Extension for Command Pipelining	N. Freed	Sep. 2000	RFC2197	RFC2920
STD0061	A One-Time Password System	N. Haller, C. Metz, P. Nesser, M. Straw	Feb. 1998	RFC1938	RFC2289

Often, it is easier to find the standard you need by referring to the common name of the protocol it defines. Table D.2 shows this relationship.

Table D.2: Standard documents listed by protocol mnemonic.

Mnemonic	Title	STD	RFC
ARP	Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware	37	826
CHARGEN	Character Generator Protocol 22	864	
Concise-MI	Concise MIB Definitions	16	1212
CONF-MIB	Conformance Statements for SMIv2	58	2580
CONV-MIB	Textual Conventions for SMIv2	58	2579
DAYTIME	Daytime Protocol	25	867
DISCARD	Discard Protocol	21	863
DOMAIN	Domain Names: Implementation and Specification	13	1035
DOMAIN	Domain Names: Concepts and Facilities	13	1034
ECHO	Echo Protocol	20	862
ETHER-MIB	Definitions of Managed Objects for the Ethernet-like Interface Types	50	1643
FTP	File Transfer Protocol	9	959

Table D.2: Standard documents listed by protocol mnemonic.

Mnemonic	Title	STD	RFC
ICMP	Internet Control Message Protocol	5	792
IGMP	Host Extensions for IP Multicasting	5	1112
IP	Internet Protocol	5	791
IP-ARC	Transmitting IP Traffic over ARCNET Networks	46	1201
IP-DC	DCN Local-Network Protocols	44	891
IP-E	Standard for the Transmission of IP Datagrams over Ethernet Networks	41	894
IP-EE	Standard for the Transmission of IP Datagrams over Experimental Ethernet Networks	42	895
IP-FDDI	Transmission of IP and ARP over FDDI Networks	36	1390
IP-FR	Multiprotocol Interconnect over Frame Relay	55	2427
IP-HC	Internet Protocol on Network System's HYPERchannel: Protocol specification	45	1044
IP-IEEE	Standard for the Transmission of IP datagrams over IEEE 802 Networks	3	1042
IP-IPX	Standard for the Transmission of 802.2 Packets over IPX Networks	49	1132
IP-NETBIOS	Standard for the Transmission of IP Datagrams over NetBIOS Networks	48	1088

Table D.2: Standard documents listed by protocol mnemonic.

Mnemonic	Title	STD	RFC
IP-SLIP	Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP	47	1055
IP-SMDS	Transmission of IP Datagrams over the SMDS Service	52	1209
IP-WB	Host Access Protocol Specification	40	907
MAIL	Standard for the Format of ARPA Internet Text Messages	11	822
MIB-II	Management Information Base for Network Management of TCP/IP-based Internets:MIB-II	17	1213
NETBIOS	Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Detailed Specifications	19	1002
NETBIOS	Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods	19	1001
ONE-PASS	A One-Time Password System	61	2289
OSPF2	OSPF Version 2	54	2328
POP3	Post Office Protocol-Version 3	53	1939
PPP	The Point-to-Point Protocol (PPP)	51	1661
PPP-HDLC	PPP in HDLC-like Framing	51	1662
QUOTE	Quote of the Day Protocol	23	865

Team-Fly®

Table D.2: Standard documents listed by protocol mnemonic.

Mnemonic	Title	STD	RFC
RARP	Reverse Address Resolution Protocol	38	903
RIP2	RIP Version 2	56	2453
RIP2-APP	RIP Version 2 Protocol Applicability Statement	57	1722
RMON-MIB	Remote Network Monitoring Management Information Base	59	2819
SMI	Structure and Identification of Management Information for TCP/IP-based Internets	16	1155
SMIv2	Structure of Management Information Version 2 (SMIv2)	58	2578
SMTP	Simple Mail Transfer Protocol	10	821
SMTP-Pipe	SMTP Service Extension for Command Pipelining	60	2920
SMTP-SIZE	SMTP Service Extension for Message Size Declaration	10	1870
SNMP	Simple Network Management Protocol (SNMP)	15	1157
TCP	Transmission Control Protocol	7	793
TELNET	Telnet Option Specifications	8	855
TELNET	Telnet Protocol Specification	8	854
TFTP	The TFTP Protocol (Revision 2)	33	1350

Table D.2: Standard documents listed by protocol mnemonic.

Mnemonic	Title	STD	RFC
TIME	Time Protocol	26	868
TOPT-BIN	Telnet Binary Transmission	27	856
TOPT-ECHO	Telnet Echo Option	28	857
TOPT-EXTOP	Telnet Extended Options: List Option	32	861
TOPT-STAT	Telnet Status Option	30	859
TOPT-SUPP	Telnet Suppress Go Ahead Option	29	858
TOPT-TIM	Telnet Timing Mark Option	31	860
TP-TCP ISO	Transport Services on top of the TCP: Version 3	35	1006
UDP	User Datagram Protocol	6	768
USERS	Active Users	24	866

Table D.3 lists the most useful or interesting (and active) RFCs as of the time this was written, along with information about the RFCs they update or make obsolete. The RFCs that are also standards are noted with their standard number in the Title column of the table. Notice that some standards have multiple RFCs, and these are not reflected in the table.

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes

0001	Host Software	S. Crocker	Apr-07-1969
0002	Host Software	B. Duvall	Apr-09-1969
0008	Functional Specifications for the ARPA Network	G. Deloche	May-05-1969
0009	Host Software	G. Deloche	May-01-1969
0012	IMP-Host Interface Flow Diagrams	M. Wingfield	Aug-26-1969
0013	Zero Text Length EOF Message	V. Cerf	Aug-20-1969
0015	Network Subsystem for Time Sharing Hosts	C.S. Carr	Sep-25-1969
0017	Some Questions re: Host-IMP Protocol	J.E. Kreznar	Aug-27-1969
0018	IMP-IMP and HOST-HOST Control Links	V. Cerf	Sep-01-1969
0020	ASCII Format for Network Interchange	V.G. Cerf	Oct-16-1969
0022	Host-Host Control Message Formats	V.G. Cerf	Oct-17-1969
0023	Transmission of	G. Gregg	Oct-16-1969

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Multiple Control Messages				
0024	Documentation Conventions	S.D. Crocker	Nov-21-1969	RFC0010, RFC0016	RFC0016
0027	Documentation Conventions	S.D. Crocker	Dec-09-1969	RFC0010, RFC0016, RFC0024	
0028	Time Standards	W.K. English	Jan-13-1970		
0029	Response to RFC 28	R.E. Kahn	Jan-19-1970		
0030	Documentation Conventions	S.D. Crocker	Feb-04-1970	RFC0010, RFC0016, RFC0024, RFC0027	
0031	Binary Message Forms in Computer	D. Bobrow, W.R. Sutherland	Feb-01-1968		
0033	New Host-Host Protocol	S.D. Crocker	Feb-12-1970		RFC0011
0036	Protocol Notes	S.D. Crocker	Mar-16-1970	RFC0033	
0038	Comments on Network Protocol from NWG/RFC #36	S.M. Wolfe	Mar-20-1970		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0039	Comments on Protocol Re: NWG/RFC #36	E. Harslem, J.F. Heafner	Mar-25-1970	RFC0036	
0040	More Comments on the Forthcoming Protocol	E. Harslem, J.F. Heafner	Mar-27-1970		
0042	Message Data Types	E. Ancona	Mar-31-1970		
0046	ARPA Network Protocol Notes	E. Meyer	Apr-17-1970		
0089	Some Historic Moments in Networking	R.M. Metcalfe	Jan-19-1971		
0093	Initial Connection Protocol	A.M. McKenzie	Jan-27-1971	RFC0066, RFC0080	
0097	First Cut at a Proposed Telnet Protocol	J.T. Melvin, R.W. Watson	Feb-15-1971		
0103	Implementation of Interrupt Keys	R.B. Kalin	Feb-24-1971		
0114	File Transfer Protocol	A.K. Bhushan	Apr-10-1971		
0128	Bytes	J. Postel	Apr-21-1971		
0137	Telnet Protocol: A	T.C.	Apr-30-1971		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Proposed Document	O'Sullivan			
0139	Discussion of Telnet Protocol	T.C. O'Sullivan	May-07-1971	RFC0137	
0141	Comments on RFC 114: A File Transfer Protocol	E. Harslem, J.F. Heafner	Apr-29-1971	RFC0114	
0147	Definition of a Socket	J.M. Winett	May-07-1971	RFC0129	
0163	Data Transfer Protocol	V.G. Cerf	May-19-1971		
0183	EBCDIC Codes and their Mapping to ASCII	J.M. Winett	Jul-21-1971		
0204	Sockets in Use	J. Postel	Aug-05-1971		
0205	NETCRT: A Character Display Protocol	R.T. Braden	Aug-06-1971		
0206	User Telnet: Description of an Initial Implementation	J.E. White	Aug-09-1971		
0208	Address Tables	A.M. McKenzie	Aug-09-1971		
0210	Improvement of Flow Control	W. Conrad	Aug-16-1971		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0236	Standard Host Names	J. Postel	Sep-27-1971		RFC0229
0281	Suggested Addition to File Transfer Protocol	A.M. McKenzie	Dec-08-1971	RFC0265	
0318	Telnet Protocols	J. Postel	Apr-03-1972	RFC0158	
0322	Well-Known Socket Numbers	V. Cerf, J. Postel	Mar-26-1972		
0328	Suggested Telnet Protocol Changes	J. Postel	Apr-29-1972		
0340	Proposed Telnet Changes	T.C. O'Sullivan	May-15-1972		
0347	Echo Process	J. Postel	May-30-1972		
0348	Discard Process	J. Postel	May-30-1972		
0385	Comments on the File Transfer Protocol	A.K. Bhushan	Aug-18-1972	RFC0354	
0412	User FTP Documentation	G. Hicks	Nov-27-1972		
0414	File Transfer Protocol (FTP) Status and Further Comments	A.K. Bhushan	Dec-29-1972	RFC0385	

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0429	Character Generator Process	J. Postel	Dec-12-1972		
0435	Telnet Issues	B. Cosell, D.C. Walden	Jan-05-1973	RFC0318	
0448	Print Files in FTP	R.T. Braden	Feb-27-1973		
0461	Telnet Protocol Meeting Announcement	A.M. McKenzie	Feb-14-1973		
0468	FTP Data Compression	R.T. Braden	Mar-08-1973		
0480	Host-Dependent FTP Parameters	J.E. White	Mar-08-1973		
0495	Telnet Protocol Specifications	A.M. McKenzie	May-01-1973		RFC0158
0559	Comments on the New Telnet Protocol and its Implementation	A.K. Bhushan	Aug-15-1973		
0560	Remote Controlled Transmission and Echoing Telnet Option	D. Crocker, J. Postel	Aug-18-1973		
0561	Standardizing Network Mail Headers	A.K. Bhushan, K.T. Pogran, R.S.	Sep-05-1973		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
		Tomlinson, J.E. White			
0562	Modifications to the Telnet Specification	A.M. McKenzie	Aug-28-1973		
0630	FTP Error Code Usage for More Reliable Mail Service	J. Sussmann	Apr-10-1974		
0640	Revised FTP Reply Codes	J. Postel	Jun-19-1974	RFC0542	
0652	Telnet Output Carriage-Return Disposition Option	D. Crocker	Oct-25-1974		
0653	Telnet Output Horizontal Tabstops Option	D. Crocker D. Crocker	Oct-25-1974 Oct-25-1974		
0654	Telnet Output Horizontal Tab Disposition Option				
0655	Telnet Output Formfeed Disposition Option	D. Crocker	Oct-25-1974		
0656	Telnet Output Vertical Tabstops Option	D. Crocker	Oct-25-1974		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0657	Telnet Output Vertical Tab Disposition Option	D. Crocker	Oct-25-1974		
0658	Telnet Output Linefeed Disposition	D. Crocker	Oct-25-1974		
0659	Announcing Additional Telnet Options	J. Postel	Oct-18-1974		
0678	Standard File Formats	J. Postel	Dec-19-1974		
0679	Feb., 1975, Survey of New-Protocol TelnetServers	D.W. Dodds	Feb-21-1975		
0680	Message Transmission Protocol	T.H. Myer, D.A. Henderson	Apr-30-1975	RFC0561	
0681	Network UNIX	S. Holmgren	Mar-18-1975		
0697	CWD Command of FTP	J. Lieb	Jul-14-1975		
0698	Telnet Extended ASCII Option	T. Mock	Jul-23-1975		
0706	On the Junk Mail Problem	J. Postel	Nov-08-1975		

TEAMFLY

Team-Fly®

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0717	Assigned Network Numbers	J. Postel	Jul-01-1976		
0726	Remote Controlled Transmission and Echoing Telnet Option	J. Postel, D. Crocker	Mar-08-1977		
0727	Telnet Logout Option	M.R. Crispin	Apr-27-1977		
0728	Minor Pitfall in the Telnet Protocol	J.D. Day	Apr-27-1977		
0730	Extensible Field Addressing	J. Postel	May-20-1977		
0732	Telnet Data Entry Terminal Option	J.D. Day	Sep-12-1977		RFC0731
0734	SUPDUP Protocol	M.R. Crispin	Oct-07-1977		
0735	Revised Telnet Byte Macro Option	D. Crocker, R.H. Gumpertz	Nov-03-1977		RFC0729
0736	Telnet SUPDUP Option	M.R. Crispin	Oct-31-1977		
0737	FTP Extension: XSEN	K. Harrenstien	Oct-31-1977		
0738	Time Server	K. Harrenstien	Oct-31-1977		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0743	FTP Extension: XRSQ/XRCP	K. Harrenstien	Dec-30-1977		
0748	Telnet Randomly-Lose Option	M.R. Crispin	Apr-01-1978		
0752	Universal Host Table	M.R. Crispin	Jan-02-1979		
0753	Internet Message Protocol	J. Postel	Mar-01-1979		
0756	NIC Name Server: A Datagram-Based Information Utility	J.R. Pickens, E.J. Feinler, J.E. Mathis	Jul-01-1979		
0759	Internet Message Protocol	J. Postel	Aug-01-1980		
0761	DoD Standard Transmission Control Protocol	J. Postel	Jan-01-1980		
0767	Structured Format for Transmission of Multimedia Documents	J. Postel	Aug-01-1980		
0768	User Datagram Protocol (STD0006)	J. Postel	Aug-28-1980		
0774	Internet Protocol Handbook: Table	J. Postel	Oct-01-1980		RFC0766

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	of Contents				
0775	Directory Oriented FTP Commands	D. Mankins, D. Franklin, A.D. Owen	Dec-01-1980		
0779	Telnet Send-Location Option	E. Killian	Apr-01-1981		
0781	Specification of the Internet Protocol (IP) Timestamp Option	Z. Su	May-01-1981		
0791	Internet Protocol (STD0005)	J. Postel	Sep-01-1981		RFC0760
0792	Internet Control Message Protocol	J. Postel	Sep-01-1981		RFC0777
0793	Transmission Control Protocol (STD0007)	J. Postel	Sep-01-1981		
0794	Pre-emption	V.G. Cerf	Sep-01-1981	IEN 125	
0795	Service Mappings	J. Postel	Sep-01-1981		
0796	Address Mappings	J. Postel	Sep-01-1981		IEN 115
0797	Format for Bitmap Files	A.R. Katz	Sep-01-1981		
0799	Internet Name	D.L. Mills	Sep-01-1981		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Domains				
0813	Window and Acknowledgement Strategy in TCP	D.D. Clark	Jul-01-1982		
0814	Name, Addresses, Ports, and Routes	D.D. Clark	Jul-01-1982		
0815	IP Datagram Reassembly Algorithms	D.D. Clark	Jul-01-1982		
0816	Fault Isolation and Recovery	D.D. Clark	Jul-01-1982		
0817	Modularity and Efficiency in Protocol Implementation	D.D. Clark	Jul-01-1982		
0818	Remote User Telnet Service	J. Postel	Nov-01-1982		
0826	Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48-bit Ethernet Address for Transmission on Ethernet Hardware (STD0037)	D.C. Plummer	Nov-01-1982		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0830	Distributed System for Internet Name Service	Z. Su	Oct-01-1982		
0854	Telnet Protocol Specification (STD0008)	J. Postel, J.K. Reynolds	May-01-1983		RFC0764
0855	Telnet Option Specifications	J. Postel, J.K. Reynolds	May-01-1983		NIC 18640
0856	Telnet Binary Transmission (STD0027)	J. Postel, J.K. Reynolds	May-01-1983		NIC 15389
0857	Telnet Echo Option (STD0028)	J. Postel, J.K. Reynolds	May-01-1983		NIC 15390
0858	Telnet Suppress Go Ahead Option (STD0029)	J. Postel, J.K. Reynolds	May-01-1983		NIC 15392
0859	Telnet Status Option (STD0030)	J. Postel, J.K. Reynolds	May-01-1983		RFC0651
0860	Telnet Timing Mark Option (STD0031)	J. Postel, J.K. Reynolds	May-01-1983		NIC 16238
0861	Telnet Extended Options: List Option (STD0032)	J. Postel, J.K. Reynolds	May-01-1983		NIC 16239

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0862	Echo Protocol (STD0020)	J. Postel	May-01-1983		
0863	Discard Protocol (STD0021)	J. Postel	May-01-1983		
0864	Character Generator Protocol (STD0022)	J. Postel	May-01-1983		
0865	Quote of the Day Protocol (STD0023)	J. Postel	May-01-1983		
0866	Active Users (STD0024)	J. Postel	May-01-1983		
0867	Daytime Protocol (STD0025)	J. Postel	May-01-1983		
0868	Time Protocol (STD0026)	J. Postel, K. Harrenstien	May-01-1983		
0869	Host Monitoring Protocol	R. Hinden	Dec-01-1983		
0872	TCP-on-a-LAN	M.A. Padlipsky	Sep-01-1982		
0876	Survey of SMTP Implementations	D. Smallberg	Sep-01-1983		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0879	TCP Maximum Segment Size and Related Topics	J. Postel	Nov-01-1983		
0885	Telnet End-of-Record Option	J. Postel	Dec-01-1983		
0886	Proposed Standard for Message Header Munging	M.T. Rose	Dec-15-1983		
0887	Resource Location Protocol	M. Accetta	Dec-01-1983		
0894	Standard for the Transmission of IP Datagrams over Ethernet Networks (STD0041)	C. Hornig	Apr-01-1984		
0895	Standard for the Transmission of IP Datagrams over Experimental Ethernet Networks (STD0042)	J. Postel	Apr-01-1984		
0896	Congestion Control in IP/TCP Internetworks	J. Nagle	Jan-06-1984		
0897	Domain Name System Implementation Schedule	J. Postel	Feb-01-1984	RFC0881	

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0903	Reverse Address Resolution Protocol (STD0038)	R. Finlayson, T. Mann, J.C. Mogul, M. Theimer	Jun-01-1984		
0906	Bootstrap Loading Using TFTP	R. Finlayson	Jun-01-1984		
0907	Host Access Protocol Specification (STD0040)	Bolt, Beranek and Newman, Inc.	Jul-01-1984		
0908	Reliable Data Protocol	D. Velten, R.M. Hinden, J. Sax	Jul-01-1984		
0913	Simple File Transfer Protocol	M. Lottor	Sep-01-1984		
0917	Internet Subnets	J.C. Mogul	Oct-01-1984		
0919	Broadcasting Internet Datagrams	J.C. Mogul	Oct-01-1984		
0920	Domain Requirements	J. Postel, J.K. Reynolds	Oct-01-1984		
0927	TACACS User Identification Telnet Option	B.A. Anderson	Dec-01-1984		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0932	Subnetwork Addressing Scheme	D.D. Clark	Jan-01-1985		
0933	Output Marking Telnet Option	S. Silverman	Jan-01-1985		
0934	Proposed Standard for Message Encapsulation	M.T. Rose, E.A. Stefferud	Jan-01-1985		
0935	Reliable Link Layer Protocols	J.G. Robinson	Jan-01-1985		
0936	Another Internet Subnet Addressing Scheme	M.J. Karels	Feb-01-1985		
0937	Post Office Protocol: Version 2	M. Butler, J. Postel, D. Chase, J. Goldberger, J.K. Reynolds	Feb-01-1985		RFC0918
0946	Telnet Terminal Location Number Option	R. Nedved	May-01-1985		
0947	Multi-network Broadcasting within the Internet	K. Lebowitz, D. Mankins	Jun-01-1985		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
0949	FTP Unique-Named Store Command	M.A. Padlipsky	Jul-01-1985		
0950	Internet Standard Subnetting Procedure	J.C. Mogul, J. Postel	Aug-01-1985	RFC0792	
0951	Bootstrap Protocol	W.J. Croft, J. Gilmore	Sep-01-1985		
0953	Hostname Server	K. Harrenstien, M.K. Stahl, E.J. Feinler	Oct-01-1985		RFC0811
0954	NICNAME/WHOIS	K. Harrenstien, M.K. Stahl, E.J. Feinler	Oct-01-1985		RFC0812
0956	Algorithms for Synchronizing Network Clocks	D.L. Mills	Sep-01-1985		
0959	File Transfer Protocol (STD0009)	J. Postel, J.K. Reynolds	Oct-01-1985		RFC0765
0972	Password Generator Protocol	F.J. Wancho	Jan-01-1986		
0977	Network News Transfer Protocol	B. Kantor, P. Lapsley	Feb-01-1986		

TEAMFLY

Team-Fly®

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
1000	Request For Comments Reference Guide	J.K. Reynolds, J. Postel	Aug-01-1987		RFC0999
1011	Official Internet Protocols	J.K. Reynolds, J. Postel	May-01-1987		RFC0991
1042	Standard for the Transmission of IP Datagrams over IEEE 802 Networks (STD0043)	J. Postel, J.K. Reynolds	Feb-01-1988		RFC0948
1047	Duplicate Messages and SMTP	C. Partridge	Feb-01-1988		
1049	Content-type Header Field for Internet Messages	M.A. Sirbu	Mar-01-1988		
1055	Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP (STD0047)	J.L. Romkey	Jun-01-1988		
1057	RPC: Remote Procedure Call Protocol Specification: Version 2	Sun Microsystems	Jun-01-1988		RFC1050
1058	Routing Information	C.L. Hedrick	Jun-01-1988		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Protocol				
1071	Computing the Internet Checksum	R.T. Braden, D.A. Borman, C. Partridge	Sep-01-1988		
1073	Telnet Window Size Option	D. Waitzman	Oct-01-1988		
1078	TCP Port Service Multiplexer (TCPMUX)	M. Lottor	Nov-01-1988		
1079	Telnet Terminal Speed Option	C.L. Hedrick	Dec-01-1988		
1082	Post Office Protocol: Version 3: Extended Service Offerings	M.T. Rose	Nov-01-1988		
1088	Standard for the Transmission of IP Datagrams over NetBIOS Networks (STD0048)	L.J. McLaughlin	Feb-01-1989		
1089	SNMP over Ethernet	M.L. Schoffstall, C. Davin, M. Fedor, J.D. Case	Feb-01-1989		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
1090	SMTP on X.25	R. Ullmann	Feb-01-1989		
1091	Telnet Terminal-Type Option	J. VanBokkelen	Feb-01-1989		RFC0930
1096	Telnet X Display Location Option	G.A. Marcy	Mar-01-1989		
1097	Telnet Subliminal-Message Option	B. Miller	Apr-01-1989		
1101	DNS Encoding of Network Names and Other Types	P.V. Mockapetris	Apr-01-1989	RFC1034, RFC1035	
1106	TCP Big Window and NAK Options	R. Fox	Jun-01-1989		
1110	Problem with the TCP Big Window Option	A.M. McKenzie	Aug-01-1989		
1112	Host Extensions for IP Multicasting	S.E. Deering	Aug-01-1989		RFC0988, RFC1054
1118	Hitchhikers Guide to the Internet	E. Krol	Sep-01-1989		
1122	Requirements for Internet Hosts: Communication Layers (STD0003)	R. Braden, Ed.	Oct. 1989		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
1123	Requirements for Internet Hosts: Application and Support	R. Braden, Ed.	Oct. 1989	RFC0822	
1129	Internet Time Synchronization: The Network Time Protocol	D.L. Mills	Oct-01-1989		
1141	Incremental Updating of the Internet Checksum	T. Mallory, A. Kullberg	Jan-01-1990	RFC1071	
1144	Compressing TCP/IP Headers for Low-Speed Serial Links	V. Jacobson	Feb-01-1990		
1146	TCP Alternate Checksum Options	J. Zweig, C. Partridge	Mar-01-1990		RFC1145
1149	Standard for the Transmission of IP Datagrams on Avian Carriers	D. Waitzman	Apr-01-1990		
1153	Digest Message Format	F.J. Wancho	Apr-01-1990		
1166	Internet Numbers	S. Kirkpatrick, M.K. Stahl, M. Recker	Jul-01-1990		RFC1117, RFC1062, RFC1020
1176	Interactive Mail	M.R. Crispin	Aug-01-1990		RFC1064

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Access Protocol: Version 2				
1180	TCP/IP Tutorial	T.J. Socolofsky, C.J. Kale	Jan-01-1991		
1184	Telnet Linemode Option	D.A. Borman	Oct-01-1990		RFC1116
1191	Path MTU Discovery	J.C. Mogul, S.E. Deering	Nov-01-1990		RFC1063
1203	Interactive Mail Access Protocol: Version 3	J. Rice	Feb-01-1991		RFC1064
1256	ICMP Router Discovery Messages	S. Deering	Sep-01-1991		
1263	TCP Extensions Considered Harmful	S. O'Malley, L.L. Peterson	Oct-01-1991		
1288	The Finger User Information Protocol	D. Zimmerman	Dec. 1991		RFC1196, RFC1194, RFC0742
1301	Multicast Transport Protocol	S. Armstrong, A. Freier, K. Marzullo	Feb. 1992		
1305	Network Time	David L.	Mar. 1992		RFC0958,

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Protocol (Version 3) Specification, Implementation	Mills			RFC1059, RFC1119
1332	The PPP Internet Protocol Control Protocol (IPCP)	G. McGregor	May 1992		RFC1172
1350	The TFTP Protocol (Revision 2) (STD0033)	K. Sollins	July 1992		RFC0783
1372	Telnet Remote Flow Control Option	C. Hedrick, D. Borman	Oct. 1992		RFC1080
1393	Traceroute Using an IP Option	G. Malkin	Jan. 1993		
1408	Telnet Environment Option	D. Borman, Editor	Jan. 1993		
1411	Telnet Authentication: Kerberos Version 4	D. Borman, Editor	Jan. 1993		
1412	Telnet Authentication: SPX	K. Alagappan	Jan. 1993		
1421	Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption	J. Linn	Feb. 1993		RFC1113

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	and Authentication Procedures				
1422	Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management	S. Kent	Feb. 1993		RFC1114
1423	Privacy Enhancement fo r Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers	D. Balenson	Feb. 1993		RFC1115
1424	Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services	B. Kaliski	Feb. 1993		
1429	Listserv Distribute Protocol	E. Thomas	Feb. 1993		
1436	The Internet Gopher Protocol (a distributed document search and retrieval protocol)	F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, B. Albert	Mar. 1993		
1437	The Extension of MIME Content-	N. Borenstein,	Apr-01-1993		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Types to a New Medium	M. Linimon			
1459	Internet Relay Chat Protocol	J. Oikarinen, D. Reed	May 1993		
1571	Telnet Environment Option Interoperability Issues	D. Borman	Jan. 1994	RFC1408	
1572	Telnet Environment Option	S. Alexander	Jan. 1994		
1579	Firewall-Friendly FTP	S. Bellovin	Feb. 1994		
1630	Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as Used in the World-Wide Web	T. Berners-Lee	June 1994		
1652	SMTP Service Extension for 8-bit-MIME Transport	J. Klensin, N. Freed, M. Rose, E. Stefferud, D. Crocker	July 1994		RFC1426
1661	The Point-to-Point Protocol (PPP) (STD0051)	W. Simpson, Editor	July 1994		RFC1548

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
1663	PPP Reliable Transmission	D. Rand	July 1994		
1681	On Many Addresses per Host	S. Bellovin	Aug. 1994		
1700	Assigned Numbers (STD0002)	J. Reynolds, J. Postel	Oct. 1994		RFC1340
1734	POP3 AUTHentication Command	J. Myers	Dec. 1994		
1736	Functional Recommendations for Internet Resource Locators	J. Kunze	Feb. 1995		
1737	Functional Requirements for Uniform Resource Names	K. Sollins, L. Masinter	Dec. 1994		
1738	Uniform Resource Locators (URL)	T. Berners-Lee, L. Masinter, M. McCahill	Dec. 1994		
1751	A Convention for Human-Readable 128-bit Keys	D. McDonald	Dec. 1994		
1777	Lightweight	W. Yeong,	Mar. 1995		RFC1487

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Directory Access Protocol	T. Howes, S. Kille			
1785	TFTP Option Negotiation Analysis	G. Malkin, A. Harkin	Mar. 1995	RFC1350	
1788	ICMP Domain Name Messages	W. Simpson	Apr. 1995		
1796	Not All RFCs Are Standards	C. Huitema, J. Postel, S. Crocker	Apr. 1995		
1808	Relative Uniform Resource Locators	R. Fielding	June 1995	RFC1738	
1834	Whois and Network Information Lookup Service, Whois++	J. Gargano, K. Weiss	Aug. 1995		
1844	Multimedia E-mail (MIME) User Agent Checklist	E. Huizer	Aug. 1995		RFC1820
1845	SMTP Service Extension for Checkpoint/Restart	D. Crocker, N. Freed, A. Cargille	Sept. 1995		
1846	SMTP 521 Reply Code	A. Durand, F. Dupont	Sept. 1995		
1847	Security Multiparts for MIME: Multipart/Signed	J. Galvin, S. Murphy, S. Crocker,	Oct. 1995		

TEAMFLY

Team-Fly®

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	and Multipart/Encrypted	N. Freed			
1848	MIME Object Security Services	S. Crocker, N. Freed, J. Galvin, S. Murphy	Oct. 1995		
1864	The Content-MD5 Header Field	J. Myers, M. Rose	Oct. 1995		RFC1544
1870	SMTP Service Extension for Message Size Declaration	J. Klensin, N. Freed, K. Moore	Nov. 1995		RFC1653
1873	Message/External-Body Content-ID Access Type	E. Levinson	Dec. 1995		
1896	The Text/Enriched MIME Content-Type	P. Resnick, A. Walker	Feb. 1996		RFC1523, RFC1563
1928	SOCKS Protocol Version 5	M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones	Mar. 1996		
1929	Username/Password Authentication for SOCKS V5	M. Leech	Mar. 1996		
1939	Post Office	J. Myers,	May 1996		RFC1725

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Protocol - Version 3 (STD0053)	M. Rose			
1945	Hypertext Transfer Protocol—HTTP/1.0	T. Berners-Lee, R. Fielding, H. Frystyk	May 1996		
1957	Some Observations on Implementations of the Post Office Protocol (POP3)	R. Nelson	June 1996	RFC1939	
1990	The PPP Multilink Protocol (MP)	K. Sklower, B. Lloyd, G. McGregor, D. Carr, T. Coradetti	Aug. 1996		RFC1717
1991	PGP Message Exchange Formats	D. Atkins, W. Stallings, P. Zimmermann	Aug. 1996		
2015	MIME Security with Pretty Good Privacy (PGP)	M. Elkins	Oct. 1996		
2017	Definition of the URL MIME External-Body Access-Type	N. Freed, K. Moore, A. Cargille	Oct. 1996		
2018	TCP Selective	M. Mathis,	Oct. 1996		RFC1072

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Acknowledgement Options	J. Mahdavi, S. Floyd, A. Romanow			
2030	Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI	D. Mills	Oct. 1996		RFC1769
2034	SMTP Service Extension for Returning Enhanced Error Codes	N. Freed	Oct. 1996		
2045	Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies	N. Freed, N. Borenstein	Nov. 1996		RFC1521, RFC1522, RFC1590
2046	Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types	N. Freed, N. Borenstein	Nov. 1996		RFC1521, RFC1522, RFC1590
2047	MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text	K. Moore	Nov. 1996		RFC1521, RFC1522, RFC1590

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
2048	Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures	N. Freed, J. Klensin, J. Postel	Nov. 1996		RFC1521, RFC1522, RFC1590
2049	Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples	N. Freed, N. Borenstein	Nov. 1996		RFC1521, RFC1522, RFC1590
2060	Internet Message Access Protocol - Version 4rev1	M. Crispin	Dec. 1996		RFC1730
2061	IMAP4 Compatibility with IMAP2bis	M. Crispin	Dec. 1996		RFC1730
2062	Internet Message Access Protocol - Obsolete Syntax	M. Crispin	Dec. 1996		
2066	TELNET CHARSET Option	R. Gellens	Jan. 1997		
2075	IP Echo Host Service	C. Partridge	Jan. 1997		
2076	Common Internet Message Headers	J. Palme	Feb. 1997		
2083	PNG (Portable	T. Boutell	Mar. 1997		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Network Graphics) Specification Version 1.0				
2086	IMAP4 ACL Extension	J. Myers	Jan. 1997		
2087	IMAP4 QUOTA Extension	J. Myers	Jan. 1997		
2088	IMAP4 Non-Synchronizing Literals	J. Myers	Jan. 1997		
2090	TFTP Multicast Option	A. Emberson	Feb. 1997		
2131	Dynamic Host Configuration Protocol	R. Droms	Mar. 1997		RFC1541
2132	DHCP Options and BOOTP Vendor Extensions	S. Alexander, R. Droms	Mar. 1997		RFC1533
2141	URN Syntax	R. Moats	May 1997		
2145	Use and Interpretation of HTTP Version Numbers	J. C. Mogul, R. Fielding, J. Gettys, H. Frystyk	May 1997		
2227	Simple Hit-Metering and Usage-Limiting for HTTP	J. Mogul, P. Leach	Oct. 1997		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
2228	FTP Security Extensions	M. Horowitz, S. Lunt	Oct. 1997	RFC0959	
2311	S/MIME Version 2 Message Specification	S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, L. Repka	Mar. 1998		
2312	S/MIME Version 2 Certificate Handling	S. Dusse, P. Hoffman, B. Ramsdell, J. Weinstein	Mar. 1998		
2315	PKCS #7: Cryptographic Message Syntax Version 1.5	B. Kaliski	Mar. 1998		
2318	The text/css Media Type	H. Lie, B. Bos, C. Lilley	Mar. 1998		
2347	TFTP Option Extension	G. Malkin, A. Harkin	May 1998	RFC1350	RFC1782
2348	TFTP Blocksize Option	G. Malkin, A. Harkin	May 1998	RFC1350	RFC1783
2349	TFTP Timeout Interval and Transfer Size Options	G. Malkin, A. Harkin	May 1998	RFC1350	RFC1784

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
2368	The mailto URL Scheme	P. Hoffman, L. Masinter, J. Zawinski	July 1998	RFC1738, RFC1808	
2387	The MIME Multipart/Related Content-type	E. Levinson	Aug. 1998		RFC2112
2388	Returning Values from Forms: Multipart/Form-data	L. Masinter	Aug. 1998		
2389	Feature Negotiation Mechanism for the File Transfer Protocol	P. Hethmon, R. Elz	Aug. 1998		
2392	Content-ID and Message-ID Uniform Resource Locators	E. Levinson	Aug. 1998		RFC2111
2414	Increasing TCP's Initial Window	M. Allman, S. Floyd, C. Partridge	Sept. 1998		
2424	Content Duration MIME Header Definition	G. Vaudreuil, G. Parsons	Sept. 1998		
2425	A MIME Content-Type for Directory Information	T. Howes, M. Smith, F. Dawson	Sept. 1998		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
2426	vCard MIME Directory Profile	F. Dawson, T. Howes	Sept. 1998		
2428	FTP Extensions for IPv6 and NATs	M. Allman, S. Ostermann, C. Metz	Sept. 1998		
2433	Microsoft PPP CHAP Extensions	G. Zorn, S. Cobb	Oct. 1998		
2437	PKCS #1: RSA Cryptography Specifications Version 2.0	B. Kaliski, J. Staddon	Oct. 1998		RFC2313
2440	OpenPGP Message Format	J. Callas, L. Donnerhacke, H. Finney, R. Thayer	Nov. 1998		
2449	POP3 Extension Mechanism	R. Gellens, C. Newman, L. Lundblade	Nov. 1998	RFC1939	
2460	Internet Protocol, Version 6 (IPv6) Specification	S. Deering, R. Hinden	Dec. 1998		RFC1883
2484	PPP LCP Internationalization Configuration	G. Zorn	Jan. 1999	RFC2284, RFC1994, RFC1570	

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Option				
2507	IP Header Compression	M. Degermark, B. Nordgren, S. Pink	Feb. 1999		
2516	A Method for Transmitting PPP over Ethernet (PPPoE)	L. Mamakos, K. Lidl, J. Evarts, D. Carrel, D. Simone, R. Wheeler	Feb. 1999		
2525	Known TCP Implementation Problems	V. Paxson, M Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, B. Volz	Mar. 1999		
2554	SMTP Service Extension for Authentication	J. Myers	Mar. 1999		
2557	MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)	J. Palme, A. Hopmann, N. Shelness	Mar. 1999		RFC2110
2577	FTP Security Considerations	M. Allman, S.	May 1999		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
		Ostermann			
2581	TCP Congestion Control	M. Allman, V. Paxson, W. Stevens	Apr. 1999		RFC2001
2616	Hypertext Transfer Protocol–HTTP/1.1	R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee	June 1999		RFC2068
2617	HTTP Authentication: Basic and Digest Access Authentication	J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, L. Stewart	June 1999		RFC2069
2632	S/MIME Version 3 Certificate Handling	B. Ramsdell, Ed.	June 1999		
2633	S/MIME Version 3 Message Specification	B. Ramsdell, Ed.	June 1999		
2634	Enhanced Security Services for S/MIME	P. Hoffman, Ed.	June 1999		

TEAMFLY

Team-Fly®

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
2637	Point-to-Point Tunneling Protocol	K. Hamzeh, G. Pall, W. Verthein, J. Taarud, W. Little, G. Zorn	July 1999		
2646	The Text/Plain Format Parameter	R. Gellens	Aug. 1999	RFC2046	
2659	Security Extensions For HTML	E. Rescorla, A. Schiffman	Aug. 1999		
2660	The Secure HyperText Transfer Protocol	E. Rescorla, A. Schiffman	Aug. 1999		
2774	An HTTP Extension Framework	H. Nielsen, P. Leach, S. Lawrence	Feb. 2000		
2779	Instant Messaging/ Presence Protocol Requirements	M. Day, S. Aggarwal, G. Mohr, J. Vincent	Feb. 2000		
2800	Internet Official Protocol Standards (STD0001)	J. Reynolds, R. Braden, S. Ginoza	May 2001		RFC2700
2810	Internet Relay Chat: Architecture	C. Kalt	Apr. 2000	RFC1459	
2811	Internet Relay Chat:	C. Kalt	Apr. 2000	RFC1459	

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	Channel Management				
2812	Internet Relay Chat: Client Protocol	C. Kalt	Apr. 2000	RFC1459	
2813	Internet Relay Chat: Server Protocol	C. Kalt	Apr. 2000	RFC1459	
2821	Simple Mail Transfer Protocol Editor (STD0010)	J. Klensin, Editor	Apr. 2001		RFC0821, RFC0974, RFC1869
2822	Internet Message Format (STD0011)	P. Resnick, Editor	Apr. 2001		RFC0822
2849	The LDAP Data Interchange Format (LDIF): Technical Specification	G. Good	June 2000		
2854	The 'text/html' Media Type	D. Connolly, L. Masinter	June 2000		RFC2070, RFC1980, RFC1942, RFC1867, RFC1866
2898	PKCS #5: Password-Based Cryptography Specification Version 2.0	B. Kaliski	Sept. 2000		
2912	Indicating Media Features for	G. Klyne	Sept. 2000		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	MIME Content				
2913	MIME Content Types in Media Feature Expressions	G. Klyne	Sept. 2000		
2941	Telnet Authentication Option	T. Ts'o, Editor, J. Altman	Sept. 2000		RFC1416
2942	Telnet Authentication: Kerberos Version 5	T. Ts'o	Sept. 2000		
2943	Telnet Authentication Using DSA	R. Housley, T. Horting, P. Yee	Sept. 2000		
2944	Telnet Authentication: SRP	T. Wu	Sept. 2000		
2945	The SRP Authentication and Key Exchange System	T. Wu	Sept. 2000		
2946	Telnet Data Encryption Option	T. Ts'o	Sept. 2000		
2947	Telnet Encryption: DES3 64-bit Cipher Feedback	J. Altman	Sept. 2000		
2948	Telnet Encryption:	J. Altman	Sept. 2000		

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
	DES3 64-bit Output Feedback				
2949	Telnet Encryption: CAST-128 64-bit Output Feedback	J. Altman	Sept. 2000		
2950	Telnet Encryption: CAST-128 64-bit Cipher Feedback	J. Altman	Sept. 2000		
2951	Telnet Authentication Using KEA and SKIPJACK	R. Housley, T. Horting, P. Yee	Sept. 2000		
2952	Telnet Encryption: DES 64-bit Cipher Feedback	T. Ts'o	Sept. 2000		
2953	Telnet Encryption: DES 64-bit Output Feedback	T. Ts'o	Sept. 2000		
2964	Use of HTTP State Management	K. Moore, N. Freed	Oct. 2000		
2965	HTTP State Management Mechanism	D. Kristol, L. Montulli	Oct. 2000		RFC2109
2980	Common NNTP Extensions	S. Barber	Oct. 2000		
3023	XML Media Types	M. Murata,	Jan. 2001	RFC2048	RFC2376

Table D.3: Useful RFCs.

RFC	Title	Author(s)	Date	Updates	Obsoletes
		S. St. Laurent, D. Kohn			
3030	SMTP Service Extensions for Transmission of Large and Binary MIME Messages	G. Vaudreuil	Dec. 2000		RFC1830
3075	XML-Signature Syntax and Processing	D. Eastlake, J. Reagle, D. Solo	Mar. 2001		
3076	Canonical XML Version 1.0	J. Boyer	Mar. 2001		
3143	Known HTTP Proxy/Caching Problems	I. Cooper, J. Dilley	June 2001		

Index

Symbols

?: operator, 41
/none/ members, 84
-count expression, 51
-o switch, 279
! operator, 36

A

abstract classes, 103-104
versus interfaces, 106
abstract members, 84
abstract method, 104
accept method, 216, 229
Access
managing with packages, 106
random file access, 162-163
Access modifiers
for fields, 92-95
for methods, 99-101
addAll(Collection c) method, 113
add(Object o) method, 113
Algorithms, security, 301
digest algorithms, 303
American Standard Code for Information Interchange. *See* ASCII files
AND (**&&**) operators, 47-49
AND algorithm, 39
append method, customized, 293-294
Applets, profiling, 280-281
appletviewer, 8
Arithmetic operations
data type precedence for, 51
optimizing, 282-283
ArithmeticException class, 60
arraycopy method, 116-117, 122
ArrayList class, 122, 125-128
inserting and removing elements with, 126-127
operations of, 127-128
speed of, 128
ArrayList collection, 115

Arrays, 23, 42-43, 112
assigning to another array, 118
boundary checking, 295
brackets for, 27, 42-43, 45
as collections, 115
comparing elements in, 117-118
copying elements, 116-117, 122
copying elements between, 42
declaring, 45-46
dimensional, 42
filling, 117-118
final modifier for, 96
initializing, 43
length field, 42
limitations of, 114
rules for, 43
searching, 117-118
shortcut initializer, 42
size of, checking, 293-294
sorting, 117-118
ASCII files, 6-7
Assignment expression, 36
Assignment operators, 40-41
Assignment statement, 66
Attributes, 275
Authenticity, 300
Autocommit feature, 168

B

Backward compatibility, 274
Base class
abstract method as, 103-104
for common operations, 138
methods, accessing in, 88
for multithreaded servers, 237-239
BigInteger, 307
Binary data
processing, 158-159
sending to server, 242
Binary operator, 34
Binary values

complement operator for, 49-50
shifting bits in, 39-40
binarySearch method, 114, 128
BindException, 236
Blanks, 25
Block statement, 66
Blocks, 56-58
defining, 27, 66
finally blocks in, 62-63
form of, 66
indenting statements in, 58
nesting, 57
boolean data type, 45
Boolean expressions, 56, 67-69
Boolean operators, 38-39, 49
NOT, 49
Boolean values
Boolean operators for, 49
changing value of, 36
operations on, 38-39
boolean variable type, 36
Brackets, 27
for arrays, 42-43, 45
break statement, 70-71, 75-76
labels, referring to, 76
for loops, 58
Browsers, building, 263-264
BufferedInputStream class, 141
BufferedInputStream object, 231
BufferedOutputStream class, 141
BufferedOutputStream object, 231
BufferedReader class, 141, 295-296
BufferedReader object, 146
BufferedWriter class, 141
Buffers
initializing, 293
size of, setting, 236
Byte arrays, encrypting, 311
byte data type, 45
ByteArrayInputStream class, 141
ByteArrayOutputStream class, 141
Bytecode, 4, 20
compiling files into, 9
executing, 9

C

Calendar class, 190
CallableStatement class, 206
Calling routines, exception handling by, 61
Capacity, of hash tables, 129
case statement
default label, 70
matching **switch** statement to, 70-71
Casting, 43-44, 51-54
implicit and explicit, 52-53
catch blocks, in methods, 62
catch keyword, multiple, 59
Char data type, 37, 45
Character class, for set membership test, 341
Character data types, 6
Characters, 29-30
comparing, 341
escape sequences for, 29-30
one-byte, 325
parsing individually, 254-255
reading, 156-158
reading more than one at a time, 295-296
sending to server, 242
three-byte, 325
two-byte, 325
characters event, 257, 261-262
CharArrayReader class, 141
CharArrayWriter class, 141
charAt method, avoiding, 294
checkError() method, 242
Cipher object, 309
encoding and decoding text with, 311-313
CipherOutputStream object
decoding streams with, 315-317
encoding streams with, 313-315
Class constructors, 104-105
Class definition, 16-17, 20, 23, 86, 102-104
curly braces for, 57
ending of, 23
.class extension, 10
Class files, storage of, 107
class keyword, 103
Class names, 29, 103

class creation by, 10
naming source files with, 17
Classes
abstract, 103-104
access modifiers for, 103
access to, managing with packages, 106-108
class keyword, 103
collections classes, 112
constructors for, 85
data-structure classes, 112
extends keyword, 103
fields in, 21
final modifier, 103
implements keyword, 103
importing, 22, 91
importing into packages, 107-108
instances of, 20-21
interface implementation, 106
items inside, 83-85
members of, 84-85
methods in, 21
naming, 29
package scope of, 21
in packages, 22, 90-91
private, 21
protected methods, access to, 87
public. *See* Public classes.
public methods, adding to, 87
serialized, **static long** name of, 276-277
visibility of, 105
ClassNotFoundError exception, 168
CLASSPATH, modifying, 255
CLASSPATH directory, 90
CLASSPATH variable, 15
clear() method, 113
Client connections, threads for, 229
Clock, creating, 240-241
close method, releasing resources with, 192
Code, 4-5. *See also* Execution of code.
braces, alignment of, 63-65
creating, 9
indenting, 63-65
lines, spacing between, 63-64
optimizing, 278-283

TEAMFLY

Team-Fly®

placeholders in, 67
profiling, 280-281
readability of, 63-65
reuse of, 82
spaces in, 65
style guidelines, 63-65
white space in, 63
Collection interface, 114-115
List interface, 115
Map interface, 115
Set interface, 115
Collections, 112-115
arrays as, 115
duplicate entries, 115
elements in, 114-115
elements in, order of, 115
elements in, processing, 122-123
elements in, removing, 123
elements in, retrieving, 123
iterator method, 122-123
key values, 115
list-type, 114
Map grouping, 114
methods in, 113
order of elements, 125-128
Set grouping, 114
Collections framework, 112
Collections.synchronizedList(list) class, 125
Collisions, 129
Comma (,), 27
Command-line programs, 4
Comments, 24-25
// syntax, 24
/* syntax, 24-25
/** syntax, 25
*/ syntax, 25
markers for, 24
placement of, 25
commit method, 168
compareTo method, 37
Compilation, 16-17
line terminators and, 20
path setup, 17
problems with, 14

Compile time data type, 45
Compiled files, 9
organization of, 10
Compiler, 4, 8
file creation, 10
package name use, 107-108
Compiler optimization, 279
Complement operator, 49-50
Compression, of socket data, 232-233
condition expression
false, 73
in **for** statements, 74
testing and executing, 72-73
Conditional operator **?:**, 41-42
if statement, replacing with, 69-70
Conditional operators, 47-49
Confidentiality, 300
connect method, 219, 221, 223
ConnectException, 236
Connection object, 166-168
autocommit feature, 168
Connections, keep-alive options, 235
Constants
defining, 95-96
in interfaces, declaring, 105-106
naming, 29, 85
Constructors, 33, 85
arguments for, 85
class constructors, 104-105
default, 85-87
naming conventions, 85
parameters names in, 90
specifying, 86
Containers, 112
containsAll(Collection c) method, 113
contains(Object o) method, 113
Content handlers, 221
custom, 226-227
ContentHandler class, 222-223, 226
ContentHandlerFactory class, 226
continue statement, 76
labels, referring to, 76
for loops, 58
copy method, 114

+count expression, 51
Country codes, 333
CR + LF line terminator, 26
CR line terminator, 26
createContentHandler method, 226
createStatement method, 192
createURLStreamHandler method, 225
crimson.jar file, 255
Curly braces, 27
alignment of, 63
for grouping statements, 23, 56-57
Currency
conversion of, 324
internationalizing, 337-339
Custom handlers
selecting, 225
setting, 225-226
Custom protocols, handling, 223-226

D

Data integrity, 300
Data security, 300. *See also* Security.
Data structures, 112
arrays, 112, 116-118. *See also* Arrays.
associative, 129-130
collections, 112-115
hash tables, 129-130
lists, 125-128
maps, 133-135
stacks, 123-125
vectors, 118-123
Data transmission, efficiency of, 235
Data types, 6-7. *See also* Primitive data types; Reference data types.
casting values to, 43-44, 52
compile time, 45
default values of, 32
get methods for, 199
implicit conversion, 52
literal, 46
mixing, 50-51
narrowing conversion, 51-53

precedence of, 51
reference data types, 33-34
widening conversion, 51-52
Database objects
closing, 199
creating, 195-199
Database queries
creating, 192
executing, 190-192
DatabaseMetaData class, 172-183
Databases
connecting to, 166-167
date formats, 187-190
driver for, selecting, 183-186
functionality, location of, 169
interrogating, 172-183
metadata, retrieving, 168, 172
rows, inserting, 204-205
SDK support for, 9
SQL for, 169-170
stored procedures, execution of, 206-207
table objects for, 195-199
transactions, committing, 205
updating, 199, 204-205
DatagramPacket class, 210
DatagramPacket object, 230
connect method, 230
disconnect method, 230
DatagramSocket class, 210
DataInput interface, 163
DataInputStream class, 159, 296-297
DataInputStream object, 231
DataOutput interface, 163
DataOutputStream class, 159
DataOutputStream object, 231
Dates
database library format, 187
formatting, 339-340
DCE, 248
Debugger, 4, 8
Declaration statement, 66
Declarations, 31-33
Decryption. *See also* Encryption.
of files, 315-317

of strings, 311-313
default constructors, 105
default label, 70
DELETE statement, 204
Development, 2
Digest algorithms
characteristics of, 303
custom, 303
MD5 algorithm, 305
SHA-1 algorithm, 304-305
Digests, 300
comparing for message validity, 317-322
Digital signatures, 300
verifying, 320-322
Directories
directory structure, 14
enumerating files in, 151-152
Distributed Computing Environment. *See* DCE
Divide-by-zero exception, 59
do statement
break statement in, 75-76
continue statement in, 76
Document object, creating, 266-267
Document Object Model. *See* DOM
Document Type Definition. *See* DTD
document variable, 267
Documentation
automatic generation of, 25
on SDK, 15
DocumentBuilder object, 256
DocumentBuilderFactory class, 256
Dollar sign ($), 27
DOM, 247
DOM parsers, 265-271
creating, 256
tree structure document representation, 265
uses of, 257
Domain names, as package names, 90, 107
DOS mode error, 14
double data type, 45
do...while statement, 66, 73
Download-management programs, 13
Driver class, 183-186
DriverManager class, 183-186

Drivers
discrepancies with database, 170-171
managing, 183-186
ODBC, 186
types of, 171
DSAPrivateKey object, 307
DSAPublicKey object, 307
DTD, 246
Dynamic loading, 5

E

EBCDIC, 6
Editions, 3
Editors, 7-8
Element object, creating, 267
empty() method, 123-125
Empty statements, 66-67
Empty strings, 29
Encapsulation, 82-83
encode method, 219
Encryption, 300
of files, 313-315
signing messages and files, 317-320
of strings, 311-313
End-of-line characters, 26
endDocument event, 257
endElement event, 257
Entity names, examples of, 29
Equality
equality expression, 36
testing for, 36-37
equals method, 37
equals(Object o) method, 113
Errors. *See also* Exceptions.
during compilation, 17
no interface methods
included, 90
Escape characters, 29-30
Exception class, 60
Exception-handler parameters, 67
Exception handling, 59
advanced techniques, 274

by calling routines, 61
of multiple exceptions, 60-61
for **SocketException**, 236
by **try** statement, 77-78
in XML, 266, 270
Exceptions, 58
BindException, 236
causes of, 58-59
checked, 62
ClassNotFoundError exception, 168
ConnectException, 236
creating, 61
detecting, 59-60
divide-by-zero exception, 59, 61-62
forcing, 79
InterruptedIOException, 233
IOException, 216, 222, 234, 236
NoRouteToHostException, 236
ParserConfigurationException error, 270
SAXException error, 270
SocketException, 235-236
types of, 61-62
unchecked, 61
UnknownHostException, 212
exceptions methods, 99
Exclusive OR (XOR) algorithm, 39
executeQuery method, 192
executeUpdate() method, 204, 206
Execution of code, 2, 9, 18
break statement in, 75-76
conditional, 56
continuing at **catch** keyword, 59
path setup, 18
problems with, 14
sequencing, 58-61
stopping, 58
tracing, 280
Expression statement, 66
Expressions, 56-58
Extended Binary Coded Decimal Interchange Code. *See* EBCDIC
extends keyword, 88, 103
Extensible Markup Language. *See* XML.

F

factory class, 225
creating, 256
Fields, 21, 83. *See also* Variables.
access modifiers for, 92-95
access to, 21
File access, random, 162-163
File class, 146-148
FileEnumerator class, 151-152
FileInputStream class, 141
FileOutputStream class, 141
FileReader class, 141
FileReader object, 145-146
Files
data, appending to, 148-149
decrypting, 315-317
digital signature, verifying, 320-322
encrypting, 313-315
enumerating, 151-152
importance of, 138
naming, 17
public classes in, 21
reading, 145-146
saving as .java, 16
signing, 317-320
writing to, 146-148
Files streams. *See also* Streams.
creating, 145-152
FileSave class, 320
FileWriter class, 141
fill method, 114
Filter classes, 142-143
creating, 154-156
FilterInputStream class, 142-143
FilterOutputStream class, 142-143
FilterReader class, 142-143
Filters
DataInputStream class, 159
DataOutputStream class, 159
FilterWriter class, 142-143
final members, 84
for classes, 103
final method, 95-96, 99

creating, 102
final variable, 32-33, 84-85, 95-96
finally blocks, 62-63, 78-79, 199
Fingerprints, message, 305
FlashGet download-management program, 13
float data type, 45
for loops, optimizing, 294-295
for statement, 66, 74-75
break statement in, 75-76
condition clause, 74
continue statement in, 76
initialization clause, 74-75
update clause, 74
Foreign languages. *See also* I18N.
determining language preference, 326-327
dialects, 327-328
language codes, 332-333
locales of, 327-332
sorting rules, 328
translating strings into, 326
Form-feed character, 25
Formatting
currency, 337-339
dates, 339-340
numbers, 337-339
percentages, 337-339
Forms, submitting, 218-219
Forte, 7
Functions, 83. *See also* Methods.
calling directly, 291-292
defining, 23
static, 22-23

G

Garbage collection
forcing, 281
for maps, 133
during tests, 281
get method, for data types, 199
getAddress method, 212, 228
getAllByName method, 212-214, 228

getAllowUserInteraction method, 220
getBundle method, 333
getByName method, 212, 228
getChildFirst method, 267
getConnection method, 186
getContent method, 217-218, 221, 226
getContentEncoding method, 221
getContentLength method, 221
getContentType method, 221
getDate method, 187, 221
getDay method, 187
getDefaultAllowUserInteraction method, 220
getDocumentElement method, 267
getDoInput method, 221
getDoOutput method, 221
getElementsByTagName method, 267
getExpiration method, 221
getFileNameMap method, 220
getHeaderField method, 221
getHeaderFieldDate method, 221
getHeaderFieldInt method, 221
getHeaderFieldKey method, 221
getHostAddress method, 212, 228
getHostName method, 212, 228
getInputStream method, 140, 221, 231
getInstance method, 304
getKeepAlive method, 235
getLastModified method, 221
getLocalHost method, 212, 228
getMetaData method, 168, 172
getMonth method, 187
getNodeValue method, 267
getOutputStream method, 140, 219, 221, 231
getReceiveBufferSize method, 236
getResponseCode method, 222
getSendBufferSize method, 236
getString method, 333
getTcpNoDelay method, 235
getYear method, 187
guessContentTypeFromInputStream method, 221
guessContentTypeFromName method, 220
GZIPInputStream object, 232

TEAMFLY

Team-Fly®

H

Hash index, 129
Hash tables, 129-130
capacity of, 129
load factor of, 129
Hash values
computation of, 131
retrieving, 134
hashcode() method, 113
HashMap class, 129-130, 133-135
methods for, 134-135
printing contents, 133-134
HashMap collection, 115
HashSet collection, 115
creating, 131-132
Hashtable class
key tags, tracking with, 263
methods of, 129-130
Hashtable collection, 112, 115
hasNext method, 123
Hidden variables, accessing, 108
Hostnames, resolving, 212-213, 228
HTML, 244
generating, 263-265
quotation marks in, 245
HTTP, JAR file retrieval with, 222
HTTP Post method, 219
HttpURLConnection class, 210, 218, 222
Hypertext Markup Language. *See* HTML.

I

Identifiers, 26-29
rules for, 27-28
IDEs, 7
if-else statement, 41, 68-69
if statement, 36, 56, 66-69
conditional operator **?:**, replacing with, 69-70
forms of, 68
statements, effect on, 56
zero arguments, detecting with, 59-60

ignoreWhitespace event, 258
implements keyword, 103, 105-106
Implicit conversion, 52
import statement, 21-22
Importing
classes, 91, 107-108
packages, 91
I18N, 324
of dates, 339-340
language codes, 332-333
locales, 327-332
of numbers and currency, 337-339
of strings, 325-327
of text, 332-336
include statement, 21
InetAddress class, 211-214, 228
Inheritance, 108-110
of **public** members, 87
Initialization of variables, 31-32
Input/output (I/O), optimizing, 295-297
InputStream class, 138
converting into **Reader**, 138
fetching, 217
filtering, 231
InputStreamReader class, 140
INSERT statement, 204
Install Anywhere (ZeroG), 275
Installation
installation directory, 13
installation path, setting, 14-15
of SDK, 13-14
Installers, 13, 274-275
Instances, 20-21, 102
Instantiation
constructors and, 104
of variables, 31-32
Integer bitwise operators, 38-39
integer data type, 45
Integers, 282
Integer.toString, 140
Integrated development environments. *See* IDEs
Integrity, data, 300
Interface bodies, 106
Interface methods, including in classes, 90

Interfaces, 88-90
versus abstract classes, 106
constants, declaring in, 105-106
declaring, 105-106
extending, 106
naming, 29
International Organization for Standardization. *See* ISO
Internationalization of software. *See* I18N.
Internet, downloading programs over, 5
InterruptedIOException, 233
Intersections, **HashSet** for, 131-132
IOException, 216, 222, 234, 236
IP address resolution, 212-213, 228
isEmpty() method, 113
ISO, 333
Iterator interface, 122-123
iterator() method, 113

J

Jar, 8
JAR files
class file storage in, 107
retrieving, 222
Jarsigner, 8
JarURLConnection class, 211
Java
editions of, 3
encryption support, 300
Internet awareness, 210
naming scheme, 3
network protocol support, 210
network support, 210-211
Unicode support, 6-7
Java API for XML-based RPC (JAX-RPC), 255
Java API for XML Messaging (JAXM), 255
Java API for XML Processing (JAXP), 255
Java API for XML Registries (JAXR), 255
Java Architecture for XML Binding (JAXB), 255
Java Authentication and Authorization Service (JAAS), 9
Java Cryptography Extensions (JCE), 9
Java Database Connectivity. *See* JDBC.

Java Document Object Model (JDOM), 255
.java extension, 10
Java Secure Sockets Extension (JSSE), 9
Java Virtual Machine. *See* JVM.
Java XML Pack (JAX), 255
javac, optimizing, 279
javac.exe, 8
executing, 16
Java.content.handler.pkgs property, 226-227
javadoc, 8
java.exe, 8
java.io package, 138-139
java.io.Serializable interface, 159-161
java.lang package, 22, 90-91
java.lang.reflection.Constructor , 237
java.net package, 210-211
java.protocol.handler.pkgs property, 225
java.security, 302
java.sql.Date class, 187-190
java.sql.Time class, 187
java.text.Collation object, 328 , 341
java.text.DecimalFormat class, 9
java.text.DecimalFormatSymbol class, 9
java.text.NumberFormat class, 9
java.util.Date class, 187, 190
java.util.Locale class, 327
getDefault method, 328-329
setDefault method, 328
java.util.zip package, 232
javax.xml.parsers.DocumentBuilder Factory property, 255
javax.xml.parsers.SAXParser Factory property, 255
jax.jar file, 255
JAXP
APIs, loading, 265-266
helper classes , 268
JBuilder (Inprise), 7
JDBC, 166
database compatibility and, 169-170
database interrogation, 172-183
driver management, 183-186
driver types, 171
Java 2 SDK 1.4 support for, 9
ODBC-to-JDBC bridge, 168, 171
JDBC 3, 168-171

SQL-99 compliance, 169
jdbc.drivers property, 186
jdb.exe, 8
J2EE (Enterprise Edition), 3
Jikes compiler, 279
J2ME (Micro Edition), 3
j2sdk-1_4_0-win.exe file, 13
J2SE (Standard Edition), 3-4
JVM, 2, 5-6, 274-275
code execution with, 9
cross-platform portability, 2, 5
installing, 5
optimization and, 279
package name use, 107
security features, 6
size of, 5
starting with **-prof** option, 280

K

Keep alives, setting, 235
Kerberos, Java 2 SDK 1.4 use of, 9
Key tags
polling for, 262-263
tracking, 263
Key values, pair value retrieval, 133-134
KeyPair object, 307
Keys, 301
algorithm for, 307
Cipher object for, 309
generating, 302, 307-310
p, **q**, and **g** numbers, 307-310
selecting, 301
values associated with, 129
Keytool, 8
Keywords, reserved, 27-28

L

Labeled statements, 76-77

Language codes, 332-333
length field, for arrays, 42
LF line terminator, 26
Library modules, dynamic loading of, 5
Library routines
for key generation, 302
optimizing performance with, 282
Line numbers, tracking, 161-162
Line terminators, 20, 25, 161
LineNumberInputStream class, 142
LineNumberReader class, 140, 142, 161-162
Lines of code, 25
LinkedList class, 125-128
speed of, 128
thread-safe, 125
LinkedList collection, 115
Linking, 5
List collections
BinarySearch method for, 128
methods for, 125-126
List interface, 115
ListResourceBundle class, 334-336
base-name description naming
conventions, 335
class file use, 334
classifying languages and
countries, 335
updating, 334
Lists
ArrayList class, 125-128
LinkedList class, 125-128
searching, 128
Literals, 46-47, 56-58
Load factor, 129
Local variables, 67
in loops, 295
scope of, 84, 93-95
Locale object, 329-332
country constants, 329
language constants, 329
methods of, 329-332
Locales, 327-332
default, 328
language-specific, 329

long data type, 45
Lookups, **ArrayList** for, 128
Loop variables, initial value for, 74
Loops
break statement in, 58
continue statement in, 58
do...while loops, 73
local variables in, 295
for loops, 73-75
object reuse in, 291
optimizing, 294-295
skipping iterations of, 76
stopping execution of, 75-76
string-handling optimization in, 293
while loops, 72-73

M

main method, 22-23, 86, 144
Map interface, 115
mark method, 152-154
markSupported method, 152
Math operators, 34-35
Mathematical operations, optimizing, 282-283
mathHelper class, 270
max method, 114
MD5 algorithm, 305
Member variables, 67, 84
scope of, 93-95
Members, 84-85
accessibility modifiers, 84
hiding, 108
overriding, 108
Message digests, 300-301
creating, 303-305
duplicate, 305
MessageBundle class, 332-334
MessageDigest class, getInstance method, 304
MessageDigestTest class, 304
Messages
digital signature, verifying, 320-322
encrypting and decrypting, 311-313

fingerprints of, 305
securing, 300
signing, 300-301, 317-320
Method bodies, 106
Method calls, 66
locking objects during, 277-278
overhead of, 295
Method definition, 97-102
curly braces for, 57
ending of, 23
Method nodes, locating, 267
Method parameters, 67
methodName methods, 99
Methods, 21, 83. *See also* Functions.
abstract, 104
access modifiers for, 99-101
access to, 21
arguments of, 97
in base class, accessing, 88
body clause, 98-99
calling of, 97
in collections, 113
constructors, 85
declaration clause, 98-99
definition of, 16-17, 97
deprecated, 190
exception handling by, 62
final, 102
inheritance of, 108-110
naming, 29
overloading, 98, 109-110
overriding, 108-109
parameters of, 97-98
private, 97
public, 97
reference types, 97-98
signatures of, 98, 108-109
static, 97
synchronization of, 125
terminating, 77
throws clause in, 101-102
Microsoft Access, 193
MIME, 217
min method, 114

minValue variable, 93
MTServerBase class
extending, 238-239
server creation with, 239-240
MulticastSocket class, 211
Multipurpose Internet Mail Extensions. *See* MIME
Multithreaded servers
automating, 237-240
creating, 236-237
Multithreading, 236-240, 274, 277-278
synchronized members, 277

N

Nagle's algorithm, 235
Namespaces, managing with packages, 106
Narrowing conversion, 51-53
native methods, 99
Native2ascii, 8
Network programs, socket programming, 210-217
Networking, 217
new method, avoiding, 291
new operator, 33, 42, 85
for array creation, 46
newDocumentBuilder method, 256
newInstance method, 237-239
newSaxParser method, 256
next method, 123
Node types, converting integer representation to strings, 268-270
NodeList object, element order in, 267
NodeType class, 268
Nonrepudiation, 300
NoRouteToHostException, 236
Notepad, 7-8
notify method, 278
with **wait** method, 291
notifyAll method, 278, 291
null, 31, 43
Number class, 104
Numbers
internationalizing, 337-339
optimization and, 282-283

Numeric data types, 6
maximum and minimum values for, 47

O

Object class, 275
Object creation statement, 66
Object locks, 277-278
releasing, 278
Object names
reserved keywords and, 28
reusing, 28-29
Object-oriented programming, 82
base classes and, 138
ObjectIntputStream class, 141
ObjectOutputStream class, 141, 231
objectRead method, custom, 286
Objects, 20-21, 82-83
converting data into, 226
creating, 24
declaring, with **new** operator, 33
equality, testing for, 37
extending, 87-88
final modifier for, 96
instantiating, 85
optimizing, 291-292
reading, 156-158, 275
reconstituting, 159-161
saving to file, 275
serializing, 159-161, 275-277. *See also* Serialization.
strings, 22
synchronizing on, 278
type of, identifying, 275
object.ser file, 286
objectWrite method, custom, 286
ODBC, 166, 170
ODBC drivers, 186
ODBC-to-JDBC bridge, 168, 171
Open Database Connectivity. *See* ODBC.
openStream method, 217
Operator, 34-42
assignment operators, 40-41

TEAMFLY

Team-Fly®

Boolean logical operators, 38-39
Boolean NOT operator, 49
complement operator, 49-50
conditional operator **?:**, 41-42
equality of, testing, 36-37
integer bitwise operators, 38-39
math operators, 34-35
precedence of, 38
prefix and postfix, 35-36
relational operators, 36
shift operators, 39-40
Optimization, 278-283
compiler-based, 279
hard, 281
of I/O, 295-297
library routines for, 282
of loops, 294-295
multithreading for, 274. *See also* Multithreading.
numbers, rules for use, 282-283
of objects, 291-292
platform variability, 281
problems with, 279
profiling for, 274
serialization for, 274. *See also* Serialization.
simple, 281
strategies for, 280
of string handling, 292-294
OR algorithm, 39
OR (||) operators, 47-49
Output streams, filtering, 231
OutputStream class, 138
converting into Writer, 138
writing to, 219
Overloaded methods, 98, 109-110
Overridden methods, 108-109
accessing, 108

P

package fields, 92-95
package keyword, 107
package methods, 99

Package scope, 21
Packages, 22, 90-91
creating, 90, 107
importing, 22, 91
importing classes into, 107
naming, 29, 90, 107
use of, 106-108
Parameters, 23
rules for use, 98
parameters methods, 99
Parentheses, 27
parse method, 256
ParserConfigurationException error, 270
Parsers, 246-247
building, 248-255
characters, parsing individually, 254-255
creating, 256-257
DOM, 247, 256-257, 265-271
SAX, 247, 256-263
UTF encoding, acceptance of, 254
white space, handling, 254
PATH variable, 8
setting, 14
peek() method, 123-125
Performance
code in stored procedures and, 170-171
optimizing. *See* Optimization.
PipedInputStream class, 141
PipedOutputStream class, 141
PipedReader class, 141
PipedWriter class, 141
Placeholders, in code, 67
Platform, 3
Policytool, 8
Polymorphism, 82-83, 88
pop() method, 123-125
Port numbers, for servers, 215-216
Ports, scanning for use of, 217
PreparedStatement objects, 205
previousString variable, 276
Primitive data types, 45
casting, 54
initializing variables with, 32
literals for, 46-47

Printing to screen, 23
println statement, 280
PrintStream class, 141, 242
PrintWriter class, 141, 242
Private classes, 21
private constructors, 105
private fields, 92-95
Private keys, 301
generating, 307-310
private members, 84
private methods, 97, 99
PrivateKey object, 307
process method, 78
processingInstruction event, 258
Professional applications
backward compatibility of, 274
exception handling, 274
execution speed, 274
installation programs, 274-275
Profiling tools, 274, 280-281
Properties collection, 115
protected constructors, 105
protected fields, 92-95
protected members, 84
protected methods, 86, 99
Protocol handlers, custom, 225-226
public classes, 10, 17, 21, 103, 107
naming, 23
public constructors, 105
public fields, 92-95
Public key cryptography, 301
Public keys, 301
generating, 307-310
public members, 84
public members, inheritance of, 87
public methods, 97, 99
publicKey object, 307
push() method, 124-125
PushbackInputStream class, 142
PushbackReader class, 142

Q

Quotation marks, single and double, 29

R

Random class, 307
Random file access, 162-163
Random-number generator, 68-69, 303, 306
Random numbers
generating, 306-307
seeds for, 306-307
unpredictable sequence of, 307
RandomAccessFile class, 162-163
Read timeouts for sockets, 233-234
readByte method, 159
Reader class, 138
Readers, 140-142
readFloat method, 159
Reading
characters, 156-158
files, 145-146
objects, 156-158
optimizing, 295-297
with random file access, 162-163
Reference data types, 33-34, 45
Reflection functions, 237
registerOutParameter method, 207
Relational operators, 36
Remote Procedure Calls. *See* RPCs.
remove method, 123, 126
removeAll(Collection c) method, 113
remove(Object o) method, 113
Repositories, text, 327
management of, 327
Requests
setting type, 222
submitting, 218-219
Requests for Comments. *See* RFCs
Reserved keywords, 27-28
reset method, 152-154
ResourceBundle class, 327

getBundle method, 333
getString method, 333
strings for keys and values, 334
Resources, releasing, 192
ResultSet object
interrogating, 199-204
persistence of, 199
populating objects with, 194-199
processing, 190-192
updatable, 204
updating database with, 204-205
ResultSetMetaData object, 199-204
retainAll(Collection c) method, 113
return statement, 77
returnThisIfFalse expression, 70
returnThisIfTrue expression, 70
returnType methods, 99
Reuse, 82-83, 291
reverse method, 114
RFCs, 210
Root directory, 13
RPCs, 248-255
XML for, 265-267
RSA Security, 301
Web site for, 310
run method, of **Thread** class, 277, 287-288
Runnable interface, 277, 288
Runtime engine, 8
Runtime system, 4

S

Sandbox, 6
SAX parsers, 247, 257-263
creating, 256
event functions, 257-258
event functions, adding code to, 264-265
key tags, polling for, 262
serial processing of XML document with, 256
SAXException error, 270
SaxParser object, 256
SaxParserFactory class, 256

Schemas, 246
Scope, 29, 92-93
defining, 57, 67
SDK, 2-5
appletviewer, 8
compiler, 4, 8
components of, 8
debugger, 8
directory, 10
directory structure, 14
documentation on, 15
downloading, 11-13
features of, 9
installing, 13-14
jar, 8
jarsigner, 8
javadoc, 8
JDBC APIs, 9
keytool, 8
multiple versions, 8
native2ascii, 8
policytool, 8
redistributing, 9
runtime engine, 8
runtime system, 4
security features, 9
source code, 4-5
Unicode support, 9
versions of, 8-9
XML parser, 9
search() method, 124-125
Searching
arrays, 117-118
lists, 128
SecureRandom class, 307
Security
algorithms, 301
goals of, 300
Java 2 SDK 1.4 features for, 9
key generation, 307-310
keys, 301
message digest, 300-301, 303-305
random-number generation, 306-307
SELECT statement, 204

Semicolon (;), 27
Separators, 26-27
SequenceInputStream class, 142, 149-151
Sequencing execution, 58-61
Serializable interface, 159-161, 275-276
Serialization, 159-161, 275-277
advanced techniques, 274
customizing, 286-287
functions of, 275
object support for, 275-276
process of, 284-286
serialVersionUID name for classes, 276-277
serialVersionUID name, 276-277
Server accept timeouts, 234
Server sockets
client connections, 229
opening, 229
Servers
clients, maximum number of, 216
compressed files, sending to, 233
connecting to, 211, 214-215
copies of, 216
incoming connections, listening for, 211-212, 216
multithreading, 236-237
passing data to, 219-220
port assignment, 211
port numbers for, 215-216
queue size, 216
requests, submitting, 218-219
server sockets, opening, 229
simple, 236
TCP sockets, 228-229
thread creation, 229
UDP sockets, 230
writing to, 211, 215-217, 241-242
ServerSocket class, 211, 215-217, 229
ServerSocket.setSoTimeout method, 234
Set interface, 115
setAllowUserInteraction method, 220
setAutoCommit method, 168
setContentHandlerFactory method, 226
setDaemon method, 278
setDate method, 187
setDefaultAllowUserInteraction method, 220

setDoInput method, 219, 221
setDoOutput method, 219, 221
setFileNameMap method, 220
setIfModifiedSince method, 221
setMonth method, 187
setReceiveBufferSize method, 236
setRequestMethod method, 219, 222
setRequestProperty method, 219, 221
setSendBufferSize method, 236
setSoLinger method, 234-235
setSoTimeout method, 233-234
setTcpNoDelay method, 235
setTime method, 187
setYear method, 187
SHA-1 algorithm, 304-305
Shadowing, 108
Shift operators, 39-40
short data type, 45
shuffle method, 114
Signature object, 317-320
sending bytes to, 320
sending data to, 322
Signatures, method, 98, 108-109
Simple API for XML. *See* SAX parsers.
Single sign-on, 9
size() method, 113
skip method, 153
skippedEntity event, 258
Slash characters (//), 24
sleep method, 289
Socket class, 211, 217
connecting to server with, 214-215
creating, 214-215
destination address, specifying, 228
exception-handling code in, 228-229
local port, specifying, 228
outgoing interface, specifying, 228
port, specifying, 228
Socket classes
DatagramSocket class, 211
ServerSocket class, 211
Socket class, 211
Socket data, compressing, 232-233
Socket programming, 210-217

function of, 211
Socket read timeouts, 233-234
SocketException, 235-236
handling, 236
Sockets. *See also* TCP sockets; UDP sockets.
addressing, 212-214
buffer size, setting, 236
delay behavior, setting, 235
encrypting, 313
keep-alive options, 235
lingering before closing, 234-235
name of, 213
reading from, 233
TCP sockets, 215-217
socket.setKeepAlive method, 235
Socket.setReceiveBufferSize method, 236
Socket.setSendBufferSize method, 236
Socket.setSoTimeout method, 233-234
socket.setTcpNoDelay method, 235
Software Development Kit. *See* SDK.
Software internationalization, 324. *See also* I18N.
sort method, 114
Sorting arrays, 117-118
Source code. *See* Code; Execution of code.
Source files, 10
naming, 17
organization of, 10
sourceString.charAt(next++) expression, 139
SQL
date formatter, 187
DELETE statement, 204
INSERT statement, 204
SELECT statement, 204
UPDATE statement, 204
SQL-99, 169-170
driver-database discrepancies, 170-171
SQL queries
creating, 192
executing, 190-192
information about, retrieving, 192-193
information about, setting, 192-193
SQL statements
stored procedures, 206-207
storing, 205

Stack class, 123-125
Stacks, 123-125
startDocument event, 257
startElement event, 257, 261-262
Start|Programs|Accessories|Command Prompt, 16
Start|Run command, 16
startServer method, passing into Class object, 237-239
Statement object
creating, 192
executeUpdate method, 204
properties, modifying, 192-193
resources, releasing, 192
Statements, 56-58
grouping, with curly braces, 56-57
indenting, 58
types of, 66
static class, 96-97
static field, 96-97
Static functions, 22-23
static long class name, 276-277
static members, 84-85
instances, creating with, 237-239
synchronizing, 278
static methods, 97, 99
static variable, 96-97
Stored procedures, 170-171, 206-207
output parameters, 207
Stream filters, 231
StreamLineNumber class, 161-162
Streams
associating with objects, 138
characters as, 139
creating, 145-152
data processing in, 158-159
definition of, 138
encrypting, 313-315
filtering, 154-156, 231
in java.io package, 138-139
joining sequentially, 149-151
position in, marking, 152-154
reading files, 145-148
rewinding, 152-154
System.out stream, 138-140
TCP socket use, 231

TEAMFLY

Team-Fly®

types of, 140-142
writing to files, 146-148
StreamTokenizer class, 157-158
String handling, optimizing, 292-294
StringBuffer class, 292-293
append operations, 293
optimizing, 292-294
synchronized method, removing from, 294
StringBufferInputStream class, 141
StringReader class, 139, 141
read method, 139
Strings, 22, 29-30
changing, 292-293
comparing for lexical order, 341
creating, 85
declaring and initializing, 33
encrypting and decrypting, 311-313
equality of, 30, 37
internationalization of, 325-327
literals for, 46-47
serializing and deserializing, 276
storing, 326-327
StringWriter class, 141
Subclasses, 104
inheritance of, 108
sun package, 223
super keyword, 88, 108
Superclasses, 104, 106
Swing components, 263-265
switch statement, 66, 70-71
break statement in, 75-76
form of, 70-71
testing conditions with, 71
Synchronization
during method calls, 278
of methods, 125
on objects, 278
synchronized members, 277
synchronized method, 99
custom, 294
Syntax, 20
System object, 23
System.arraycopy() method, 122, 282
System.gc, 281

System.out stream, 139-140
converting to **OutputStreamWriter** object, 144
println method, 140
System.out.print statement, 110
System.out.println statement, 23

T

Tab character, 25
Table objects, creating, 194-199
TCP, Java socket support for, 210
TCP sockets, 211, 215-217
opening, 228-229
receiving data from, 231-232
sending data to, 231
Ternary operator, 34, 42
Tests
garbage collection during, 281
switch statement for, 71
testVariableScope method, 93
Text
internationalizing, with ListResourceBundle, 334-336
internationalizing, with **MessageBundle**, 332-334
repository for, 326
Text editor, 2
writing code with, 15-16
Text translations, 326-327
this keyword, 90
Thread class, 277
extending, 237, 277, 287-288
run method, 287-288
Threads. *See also* Multith reading.
beginning execution, 277
creating, 287-288
for incoming requests, 229
object lock ownership, 277-278
pausing, 289-291
reads and processing, separating in, 234
releasing, 278, 291
sleeping, 289-291
wait method, calling from within, 278
wait state, 278, 291

yielding, 289-291
throw statement, 61-62, 79, 101-102
Time
level of accuracy of system, 281
retrieving, 240
Time checker, 240-241
Time servers, opening, 223-224
Timeouts
server accept timeouts, 234
socket read timeouts, 233-234
Timer class, **finally** statement in, 78-79
toArray() method, 113
Token list, refining, 26
Tokenizing, 157
Tokens, 20, 25-26
multiple-line comments between, 25
Tools
editors, 7-8
IDEs, 7
Transactions, committing, 205
transient members, 84
transient modifier, 276
Transmission Control Protocol. *See* TCP
TreeMap collection, 115
TreeSet collection, 115
creating, 132-133
try blocks, 168
in methods, 62
try-catch-finally statement, 78-79
try-catch statement, 77-78
try statement
exception detection with, 59-60
finally blocks with, 62-63

U

UDP, Java socket support for, 210
UDP sockets, 211, 230
Unary operator, 34
Underscore character (_), 27
Unicode, 6-7, 325
mapping bytes into, 138-139

SDK support for, 9
Unicode Transformational Format. *See* UTF encoding.
Unions, **HashSet** for, 131-132
Unix, root-to-start servers on, 216
UnknownHostException, 212
UPDATE statement, 204
Updates, database, 204-205
URL class, 211, 217-218
content handlers, custom, 226-227
creating, 224
openConnection member, 218
protocols, custom, 223-226
retrieving contents of, 217-218
time, retrieving with, 240
URLConnection class, 218-222
communication with server, 223
creating, 224
methods of, 220-221
subclasses of, 222
URLDecoder class, 211
URLEncoder class, 211, 219
URLs, for database access, 168
URL.setURLStreamHandler Factory method, 223, 225-226
URLStreamHandler class, 222
URLStreamHandlerFactory class, 223-225
custom handler selection, 225
User Datagram Protocol. *See* UDP
User-interface design, 324
Users, determining language preference, 326-327
UTF-8 encoding, 325
advantages of, 325
characters in, 325
UTF encoding, 7, 254

V

Values, keys associated with, 129
Variable scope, 67, 92-93. *See also* Scope.
Variables, 21, 31, 83. *See also* Fields.
accessibility of, 57, 67. *See also* Scope.
casting, 43-44
declaring, 16-17, 31, 45-46

default value, initializing to, 32
exception-handler parameters, 67
final variables, 32-33, 84-85
inheritance of, 108-110
initialization of, 31-32
instantiation of, 31-32
local variables, 67
member variables, 67
method parameters, 67
multithreading and, 277
naming, 29, 31
as object references, 33-34
as simple types, 33
values, assigning, 16-17
visibility of, 67
Vector class, 118-122
ArrayList class, 125-128
iterating, 123
objects in, 119
Vector collection, 115
Vectors, 118-122
adding elements to, 121-122
internal algorithms, 112
resizing, 119-122
verify method, 322
Versions, backward compatibility of, 8
Visibility
of classes, 105
of fields, 92-95
VisualAge for Java (IBM), 7
volatile members, 84

W

wait method
calling from within thread, 278
notify method with, 291
WeakHashMap collection, 115, 133-135
methods for, 134-135
Web browsers, building, 263-264
Web pages
data presentation, describing

with HTML, 244
reading, 217-218
Web servers, writing to, 241-242
Web services, 241-242
WebSphere (IBM), 7
while statement, 66, 72-73
break statement in, 75-76
continue statement in, 76
White space, 20, 25-26
parser handling of, 254
Widening conversion, 51-52
Wildcard (*), 108
Windows systems
ODBC standard for, 166
PATH variable, setting
permanently, 14
writeInt method, 159
writeObject method, 276
Writer class, 138
Writers, 140-142
Writing
code, 15-16
data to sockets, 231
to files, 146-148
with random file access, 162-163

X

XML, 244-246. *See also* Parsers.
n element, 258
characters in, 261
<color> element, 258
DTDs, 246
elements, retrieving, 267
elements of, 258
event functions, 257-258
event functions, adding code to, 264-265
exception handling in, 266, 270
helper classes, 268
<item> element, 258
Java extensions for, 255
Java parser for, 9

portability of, 244
RPC project, 248-255
version information, retrieving, 249
XSchema, 246
XML documents, 245
argument list, 249
building, with DOM parser, 257
comments, 248
communicating to program, 247
Document object from, 266-267
elements, adding, removing, changing, 270-271
encoding, verifying, 249
method name, 249
processing, 246-247
prolog, 248
root, 248
serial processing of, 256
tree structure, 265, 270
validity of, 266
verifying, 249
XML tags, 245-246
case sensitivity, 244
closing tags, 244-245
formatting, 244-245
predefined, 245
quotation marks for attributes, 245
XOR algorithm. *See* Exclusive OR algorithm
XSchema, 246

Y

yield method, 289

List of Figures

Chapter 1: Getting Started with Java

Figure 1.1: The Java 2 Platform, Standard Edition home page.
Figure 1.2: Selecting the operating system.
Figure 1.3: The download page.
Figure 1.4: The recommended directory structure for the local hard drive.
Figure 1.5: Setting the **PATH** system variable.

Chapter 2: Essential Java Syntax

Figure 2.1: Result of using **&** on the numbers 2 and 3.

List of Tables

Chapter 2: Essential Java Syntax

Table 2.1: Examples of entity names.
Table 2.2: The escape sequences in Java.
Table 2.3: Default values for given data types.
Table 2.4: Java operators.
Table 2.5: Operator precedence.
Table 2.6: Boolean operations.
Table 2.7: Shift operators.
Table 2.8: Assignment operators.
Table 2.9: Primitive data types.
Table 2.10: List of literal data types.

Chapter 3: Blocks and Statements

Table 3.1: Common Java statements.

Chapter 4: Methods, Classes, and Packages

Table 4.1: Member modifiers.
Table 4.2: Method declaration elements.
Table 4.3: Class declaration elements.
Table 4.4: Constructor access modifiers.

Chapter 5: Data Structures

Table 5.1: Methods common to most collections.
Table 5.2: Methods that only some collections implement.

Chapter 6: Files and Streams

Table 6.1: Stream sources and sinks.
Table 6.2: Manipulating streams.

Chapter 9: XML and Java

Table 9.1: Methods called by SAX parsers.
Table 9.2: SAX events occurring while the previous XML document is parsed.

Appendix A: Development Tools

Table A.1: Common options for the javac compiler.
Table A.2: Options for the java program launcher.
Table A.3: javadoc documentation tags.
Table A.4: Options for the jar command.

Appendix B: References

Table B.1: Sun tutorials.
Table B.2: Sun questions and exercises.
Table B.3: Sun articles with sample code.
Table B.4: Sun quizzes.

Table B.5: Sun certification.
Table B.6: Sun instructor-led courses.
Table B.7: Sun CD-ROM courses.
Table B.8: Sun Web-based courses.
Table B.9: Quick links to Java SDK features.
Table B.10: API and language documentation.
Table B.11: Online Java books.
Table B.12: Javacats lists of Java resources.
Table B.13: Miscellaneous resources.

Appendix D: Active RFCs

Table D.1: Internet standards documents.
Table D.2: Standard documents listed by protocol mnemonic.
Table D.3: Useful RFCs.

TEAMFLY

Team-Fly®

List of Listings

Chapter 2: Essential Java Syntax

Listing 2.1: This simple program does all of its work in the **main** function.
Listing 2.2: This simple program creates an object.
Listing 2.3: This program demonstrates some string comparisons.
Listing 2.4: Using the shift operators.
Listing 2.5: Getting the maximum and minimum values for Java's numerical data types.
Listing 2.6: A conditional operator demonstration program.

Chapter 3: Blocks and Statements

Listing 3.1: A program that generates an exception.
Listing 3.2: This program catches exceptions that occur.
Listing 3.3: This code handles multiple exceptions differently.
Listing 3.4: Calling routines can handle exceptions.
Listing 3.5: The **finally** block executes at the end of its main block in all cases.

Chapter 4: Methods, Classes, and Packages

Listing 4.1: A scope demonstration program.
Listing 4.2: An access modifier demonstration program.
Listing 4.3: A **static** keyword demonstration program.
Listing 4.4: A method access demonstration program.
Example 4.5: A method overriding demonstration program.

Chapter 5: Data Structures

Example 5.1: Using the **Vector** class.
Example 5.2: Iterating a **Vector**.
Example 5.3: Using a **HashMap**.

Chapter 6: Files and Streams

Example 6.1: This filter class converts characters to uppercase.
Example 6.2: A simple file-reader program.
Example 6.3: A file stream demonstration program.
Example 6.4: A **SequenceInputStream** demonstration program.
Example 6.5: This class enumerates the files in a directory.
Example 6.6: A **skip** and **mark** program.
Example 6.7: A filtering stream program.
Example 6.8: A **StreamTokenizer** demonstration program.
Example 6.9: A serialization demonstration program.
Example 6.10: A **StreamLineNumber** demonstration program.
Example 6.11: A random access demonstration program.

Chapter 7: Java Database Connectivity

Example 7.1: Connecting to a database.
Example 7.2: Interrogating a connection.

Example 7.3: Using the **DriverManager** and **Driver** classes.
Example 7.4: Using the **java.sql.Date** object.
Example 7.5: Executing a database query.
Example 7.6: Modifying the **Statement** object's properties.
Example 7.7: Creating table objects in Java.
Example 7.8: Interrogating a result set.

Chapter 8: The Internet and Networking

Listing 8.1: A simple console program that can resolve a name or IP address.
Listing 8.2: How to use the **getAllByName** method.
Listing 8.3: A simple program that connects to a Web server.
Listing 8.4: A program that provides a server on port 8123.
Listing 8.5: A program that can be used to scan your computer for ports in use.
Listing 8.6: A simple way to read a Web page.
Listing 8.7: Submitting data with **URLConnection**.
Listing 8.8: A subclass of **URLConnection** that opens a time server.
Listing 8.9: A simple class that creates the correct **URLConnection** subclass on behalf of the **URL** class.
Listing 8.10: The **URLStreamHandlerFactory** class selects a custom handler for a protocol.
Listing 8.11: This simple program uses the custom protocol handler.
Listing 8.12: Finding a hostname or IP address by using **InetAddress**.
Listing 8.13: Opening a socket requires exception-handling code.
Listing 8.14: The **ServerSocket** class makes it easy to accept client connections.
Listing 8.15: UDP sockets use **DatagramPacket** objects to send and receive data.
Listing 8.16: A TCP socket uses streams to send data.
Listing 8.17: TCP sockets deliver data via streams.
Listing 8.18 : Using Java's compressions classes to reduce the amount of data sent over a socket.
Listing 8.19: Sending a compressed file to the server in Listing 8.18.
Listing 8.20: A basic multithreaded server.
Listing 8.21 : Using a base class to build multithreaded servers easily.
Listing 8.22: An example server using the **MTServerBase** object.
Listing 8.23: Building a NIST standard time checker.
Listing 8.24: Building a Yahoo stock-quote fetcher.

Chapter 9: XML and Java

Example 9.1: Listing 9.1 Building a basic XML parser.
Example 9.2: Listing 9.2 Building a SAX parser.
Example 9.3: Building a browser with a GUI.

Chapter 10: Advanced Techniques

Listing 10.1: Serializing objects.
Listing 10.2: Subclassing the **Thread** object.
Listing 10.3: Using the **Thread** object's methods.

Chapter 11: Security

Listing 11.1: Creating a message digest.
Listing 11.2: Creating secure random numbers.
Listing 11.3: Generating security keys.
Listing 11.4: Encrypting and decrypting a string.
Listing 11.5: Encrypting a file.
Listing 11.6: Decrypting a file.
Listing 11.7: Digitally signing a file.
Listing 11.8: Verifying a digitally signed file.

Chapter 12: Internationalization

Listing 12.1: Demonstrating the **Locale** object.
Listing 12.2: Using the **ResourceBundle**.
Listing 12.3: Demonstrating the **ListResourceBundle** object.
Listing 12.4: Using a **ListResourceBundle** .
Listing 12.5: Formatting numbers, currency, and percentages.
Listing 12.6: Formatting dates.