



Learn Java for Android Development

Migrating Java SE Programming Skills
to Mobile Development

Fourth Edition

Peter Späth
Jeff Friesen

Apress®

Learn Java for Android Development

Migrating Java SE Programming
Skills to Mobile Development

Fourth Edition

**Peter Späth
Jeff Friesen**

Apress®

Learn Java for Android Development: Migrating Java SE Programming Skills to Mobile Development

Peter Späth
Leipzig, Sachsen, Germany

Jeff Friesen
Winnipeg, MB, Canada

ISBN-13 (pbk): 978-1-4842-5942-9
<https://doi.org/10.1007/978-1-4842-5943-6>

ISBN-13 (electronic): 978-1-4842-5943-6

Copyright © 2020 by Peter Späth and Jeff Friesen

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spaehr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Editorial Operations Manager: Mark Powers

Cover designed by eStudioCalamar

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484259429. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

Table of Contents

About the Authors.....	xv
About the Technical Reviewer	xvii
Introduction	xix
Fourth Edition Notes	xxi
Chapter 1: Getting Started with Java	1
What Is Java?.....	2
Java Is a Language.....	2
Java Is a Platform.....	3
Java SE and Java EE.....	6
Installing the JDK and Exploring Example Applications.....	7
Hello, World!	8
DumpArgs.....	12
EchoText	13
Installing and Exploring the Eclipse IDE.....	16
Java Meets Android.....	21
What Is Android?	21
History of Android.....	22
Android Architecture.....	22
Android Says Hello	26
Summary.....	28

TABLE OF CONTENTS

Chapter 2: Learning Language Fundamentals	31
Learning Application Structure	31
Learning Comments	33
Single-Line Comments	33
Multiline Comments.....	34
Javadoc Comments	34
Learning Identifiers.....	38
Learning Types	39
Primitive Types	40
Object Types	42
Array Types	43
Learning Variables	43
Learning Expressions.....	44
Simple Expressions	45
Compound Expressions	51
Learning Statements.....	80
Assignment Statements	80
Decision Statements.....	81
Loop Statements	87
Break Statements.....	94
Continue Statements	96
Summary.....	98
Chapter 3: Discovering Classes and Objects	101
Declaring Classes	102
Classes and Applications.....	102
Constructing Objects.....	103
Default Constructor	105
Explicit Constructors.....	105
Objects and Applications	109

TABLE OF CONTENTS

Encapsulating State and Behaviors	111
Representing State via Fields.....	111
Representing Behaviors via Methods.....	121
Hiding Information	136
Initializing Classes and Objects	142
Class Initializers	143
Instance Initializers	144
Initialization Order	146
Collecting Garbage.....	146
Revisiting Arrays	147
Summary.....	154
Chapter 4: Discovering Inheritance, Polymorphism, and Interfaces	157
Building Class Hierarchies	157
Extending Classes	158
The Ultimate Superclass.....	165
Composition.....	173
Changing Form.....	174
Upcasting and Late Binding.....	176
Abstract Classes and Abstract Methods	180
Downcasting and Runtime Type Identification	182
Covariant Return Types.....	185
Formalizing Class Interfaces.....	186
Declaring Interfaces	186
Implementing Interfaces.....	188
Extending Interfaces.....	192
Why Use Interfaces?.....	193
Summary.....	201

TABLE OF CONTENTS

Chapter 5: Mastering Advanced Language Features, Part 1.....	203
Mastering Nested Types.....	203
Static Member Classes.....	203
Nonstatic Member Classes.....	208
Anonymous Classes.....	210
Local Classes.....	212
Interfaces Within Classes	214
Mastering Packages	215
What Are Packages?.....	215
Package Names Must Be Unique	216
The Package Statement	216
The Import Statement.....	217
Searching for Packages and Types.....	218
Playing with Packages	220
Packages and JAR Files	225
Mastering Static Imports.....	226
Mastering Exceptions.....	227
What Are Exceptions?.....	228
Representing Exceptions in Source Code.....	228
Throwing Exceptions	233
Handling Exceptions	236
Performing Cleanup.....	241
Automatic Resource Management	244
Summary.....	247
Chapter 6: Mastering Advanced Language Features, Part 2.....	249
Mastering Annotations	249
Discovering Annotations	249
Declaring Annotation Types and Annotating Source Code	253
Processing Annotations	259

TABLE OF CONTENTS

Mastering Generics	261
Collections and the Need for Type Safety	261
Generic Types	263
Generic Methods	275
Arrays and Generics	279
Mastering Enums	282
The Trouble with Traditional Enumerated Types	282
The Enum Alternative	284
The Enum Class	290
Summary.....	296
Chapter 7: Exploring the Basic APIs, Part 1.....	299
Exploring Math.....	299
Exploring Number and Its Children	302
BigDecimal	303
BigInteger	307
Primitive Type Wrapper Classes	311
Exploring String, StringBuffer, and StringBuilder	319
String	319
StringBuffer and StringBuilder	322
Exploring System	324
Exploring Threads	327
Runnable and Thread	328
Synchronization	334
Thread-Local Variables	350
Summary.....	355
Chapter 8: Exploring the Basic APIs, Part 2.....	359
Exploring Random	359
Exploring Reflection	363
The Class Entry Point	363
Constructor, Field, and Method	369

TABLE OF CONTENTS

Package.....	373
Array.....	378
Exploring StringTokenizer	379
Exploring Timer and TimerTask.....	381
Summary.....	384
Chapter 9: Exploring the Collections Framework	387
Exploring Collections Framework Fundamentals.....	387
Comparable vs. Comparator	389
Iterable and Collection.....	391
Exploring Lists.....	395
ArrayList	396
LinkedList	397
Exploring Sets	398
TreeSet	398
HashSet	400
EnumSet	400
Exploring Sorted Sets.....	401
Exploring Navigable Sets	404
Exploring Queues	406
PriorityQueue.....	407
Exploring Deques	409
ArrayDeque	410
Exploring Maps	411
TreeMap.....	414
HashMap	414
IdentityHashMap.....	418
WeakHashMap.....	419
EnumMap	419
Exploring Sorted Maps	419
Exploring Navigable Maps.....	420

TABLE OF CONTENTS

Exploring the Arrays and Collections Utility APIs.....	420
Exploring the Legacy Collection APIs	423
Summary.....	428
Chapter 10: Functional Programming	431
Functions and Operators	432
Lambda Calculus.....	432
Entering a Stream	433
Mapping	435
Filtering.....	436
Terminating a Stream.....	437
Performing Actions on Each Element.....	437
Limiting, Skipping, Sorting, and Distinct	438
Ranges	438
Reducing	438
Collecting	439
Methods As Functions.....	442
Single-Method Interfaces.....	443
Streams and Parallelization	444
Protonpack, a Stream Utility Library	444
Summary.....	446
Chapter 11: Exploring the Concurrency Utilities.....	449
Introducing the Concurrency Utilities.....	449
Exploring Executors	451
Exploring Synchronizers	454
Countdown Latches	454
Cyclic Barriers	454
Exchangers.....	455
Semaphores	455

TABLE OF CONTENTS

Exploring the Concurrent Collections.....	456
Exploring the Locking Framework.....	457
Lock.....	457
ReentrantLock	459
Condition	459
ReadWriteLock	460
ReentrantReadWriteLock.....	460
Exploring Atomic Variables.....	461
Summary.....	464
Chapter 12: Performing Classic I/O	467
Working with the File API	467
Constructing File Instances	468
Learning About Stored Abstract Pathnames.....	470
Learning About a Pathname's File or Directory	471
Obtaining Disk Space Information.....	472
Listing Directories	473
Creating and Manipulating Files and Directories.....	475
Setting and Getting Permissions	476
Working with the RandomAccessFile API.....	478
Working with Streams.....	479
Stream Classes Overview.....	479
ByteArrayOutputStream and ByteArrayOutputStream.....	481
FileOutputStream and FileInputStream	483
PipedOutputStream and PipedInputStream.....	485
FilterOutputStream and FilterInputStream	488
BufferedOutputStream and BufferedInputStream	491
DataOutputStream and DataInputStream	492
Object Serialization and Deserialization	495
PrintStream	501
Standard I/O Revisited.....	503

TABLE OF CONTENTS

Working with Writers and Readers	505
Writer and Reader Classes Overview	506
Writer and Reader	508
OutputStreamWriter and InputStreamReader.....	508
FileWriter and FileReader	510
Summary.....	516
Chapter 13: Accessing Networks	519
Accessing Networks via Sockets	519
Socket Addresses	521
Socket Options	522
Socket and ServerSocket	524
DatagramSocket and MulticastSocket	530
Accessing Networks via URLs.....	535
URL and URLConnection	535
URLEncoder and URLDecoder.....	539
URI	541
Accessing Network Interfaces and Interface Addresses.....	542
Managing Cookies.....	546
Summary.....	551
Chapter 14: Migrating to New I/O.....	555
Working with Buffers	556
Buffer and Its Children.....	557
Working with Channels	559
Channel and Its Children	560
Working with Selectors	566
Selector Fundamentals.....	566
Selector Demonstration.....	572
Working with Regular Expressions	577
Pattern, PatternSyntaxException, and Matcher	577
Character Classes.....	580

TABLE OF CONTENTS

Capturing Groups.....	582
Boundary Matchers and Zero-Length Matches	583
Quantifiers	584
Practical Regular Expressions	587
Working with Charsets.....	588
A Brief Review of the Fundamentals	588
Working with Charsets	589
Charsets and the String Class	593
Working with Formatter and Scanner	595
Working with Formatter.....	596
Working with Scanner	601
Summary.....	605
Chapter 15: Accessing Databases	607
Introducing Apache Derby	608
Apache Derby Installation and Configuration	611
Apache Derby Demos	611
Apache Derby Command-Line Tools	611
Starting an Apache Derby Server	613
Embedded Apache Derby Example	614
Introducing SQLite	615
Accessing Databases via JDBC.....	617
Data Sources, Drivers, and Connections.....	617
Statements	621
Metadata	632
Summary.....	637
Chapter 16: Working with XML and JSON Documents.....	641
What Is XML?	641
XML Declaration	643
Elements and Attributes	645
Character References and CDATA Sections.....	647

TABLE OF CONTENTS

Namespaces.....	649
Comment and Processing Instructions.....	654
Well-Formed Documents	655
Valid Documents.....	656
Parsing XML Documents with SAX.....	658
Exploring the SAX API.....	658
Demonstrating the SAX API	661
Parsing and Creating XML Documents with DOM	672
A Tree of Nodes.....	673
Exploring the DOM API.....	676
What Is JSON?	688
JSON Processing in Java	689
Generating JSON	689
Parsing JSON.....	693
Summary.....	697
Chapter 17: Date and Time	699
The Traditional Date and Time API.....	699
About Dates and Calendars	700
Date and Time Formatters	708
Parsing	710
The New Date and Time API.....	713
Local Dates and Times.....	713
Instants.....	716
Offset Dates and Times	717
Zoned Dates and Times	720
Duration and Periods	722
Clock.....	725
Summary.....	728

TABLE OF CONTENTS

Appendix A: Solutions to Exercises	729
Chapter 1: Getting Started with Java	729
Chapter 2: Learning Language Fundamentals	731
Chapter 3: Discovering Classes and Objects.....	734
Chapter 4: Discovering Inheritance, Polymorphism, and Interfaces.....	741
Chapter 5: Mastering Advanced Language Features, Part 1	750
Chapter 6: Mastering Advanced Language Features, Part 2	758
Chapter 7: Exploring the Basic APIs, Part 1.....	765
Chapter 8: Exploring the Basic APIs, Part 2.....	775
Chapter 9: Exploring the Collections Framework.....	780
Chapter 10: Functional Programming	788
Chapter 11: Exploring the Concurrency Utilities.....	791
Chapter 12: Performing Classic I/O	794
Chapter 13: Accessing Networks	805
Chapter 14: Migrating to New I/O.....	811
Chapter 15: Accessing Databases.....	818
Chapter 16: Working with XML and JSON Documents	820
Chapter 17: Date and Time.....	826
Index.....	831

About the Authors

Peter Späth consults, trains/teaches, and writes books on various subjects, with a primary focus on software development. With a wealth of experience in Java-related languages, authoring the new edition of a “Java for Android” book seemed to be a very good idea with respect to improving the community’s and other audience’s proficiency for developing Android apps. He also graduated in 2002 as a physicist and soon afterward became an IT consultant, mainly for Java-related projects.

Jeff Friesen is a freelance tutor and software developer with an emphasis on Java (and now Android). In addition to authoring *Learn Java for Android Development* and co-authoring *Android Recipes*, Jeff has written numerous articles on Java and other technologies for JavaWorld, informIT, Java.net, and DevSource.

About the Technical Reviewer

Chad (“Shod”) Darby is an author, instructor, and speaker in the Java development world. As a recognized authority on Java applications and architectures, he has presented technical sessions at software development conferences worldwide (in the United States, the United Kingdom, India, Russia, and Australia). In his 15 years as a professional software architect, he’s had the opportunity to work for Blue Cross/Blue Shield, Merck, Boeing, Red Hat, and a handful of startup companies.

Chad is a contributing author to several Java books, including *Professional Java E-Commerce* (Wrox Press), *Beginning Java Networking* (Wrox Press), and *XML and Web Services Unleashed* (Sams Publishing). He has Java certifications from Sun Microsystems and IBM. Chad holds a BS in computer science from Carnegie Mellon University. You can visit his blog at www.luv2code.com to view his free video tutorials on Java. You can also follow him on Twitter at @darbyluvs2code.

Introduction

Smartphones and tablets are all the rage these days. Their popularity is largely due to their ability to run apps.

Android app developers are making money by selling such apps or intra-app features.

In today's challenging economic climate, you might like to try your hand at developing Android apps and generate some income. If you have good ideas, perseverance, and some artistic talent (or perhaps know some talented individuals), you are already part of the way toward achieving this goal.

Most importantly, you'll need to possess a solid understanding of the Java language and foundational application programming interfaces (APIs) before jumping into Android. After all, many Android apps are written in Java and interact with many of the standard Java APIs (such as threading and input/output APIs).

We wrote *Learn Java for Android Development* to give you a solid Java foundation that you can later extend with knowledge of Android architecture, API, and tool specifics. This book will give you a strong grasp of the Java language and the many important APIs that are fundamental to Android apps and other Java applications. It will also introduce you to key development tools.

Fourth Edition Notes

With current Android versions, the new features Java 8 introduced also entered the Android development world. For this reason, Java 8 features were added in the new book edition, namely, functional programming aspects, the streaming API, the new date and time API, and JSON handling. To streamline the book, more or less verbatim copies of parts of the official documentation were replaced by small URL references. Besides, corner cases, while valuable for certain development scenarios, were removed—after all the book is targeting Java beginners and all too advanced topics might confuse more than help to learn Java fundamentals.

Having the official Java documentation open in a browser window while reading the book certainly is a good idea.

Android meanwhile has become the most important smartphone development platform, so if you want to learn smartphone app development, starting with a decent Android version and for the very popular Java platform is a very good idea.

We hope the new book release will help you to readily acquire enough Java language programming skill, so you can explore the fun of Android application development soon.

Book Organization

The first edition of this book was organized into 10 chapters and 1 appendix. The second edition was organized into 14 chapters and 3 appendixes. The third edition was organized into 16 chapters and 2 appendixes with a bonus appendix on Android app development. For the fourth edition, a new Chapter 10 is added for functional programming and streams, and Chapter 17 contains a new introduction to the new date and time API (as of Java 8). Important topics of the former Chapter 16 were moved at appropriate places in the other chapters. Each chapter in each edition offers a set of exercises that you should complete to get the most benefit from its content. Their solutions are presented in Appendix A.

FOURTH EDITION NOTES

Chapter 1 introduces you to Java by first focusing on Java's dual nature (language and platform). It then briefly introduces you to Oracle's Java SE and Java EE editions of the Java platform. You next learn how to download and install the Java SE Development Kit (JDK), and you learn some Java basics by developing and playing with three simple Java applications. After receiving a brief introduction to the Eclipse IDE, you receive a brief introduction to Android.

Chapter 2 starts you on an in-depth journey of the Java language by focusing on language fundamentals. You first learn about simple application structure and then learn about comments, identifiers (and reserved words), types, variables, expressions (and literals), and statements.

Chapter 3 continues your journey by focusing on classes and objects. You learn how to declare a class and organize applications around multiple classes. You then learn how to construct objects from classes, declare fields in classes and access these fields, declare methods in classes and call them, initialize classes and objects, and remove objects when they're no longer needed. You also learn more about arrays, which were first introduced in Chapter 2.

Chapter 4 adds to Chapter 3's pool of object-based knowledge by introducing you to the language features that take you from object-based applications to object-oriented applications. Specifically, you learn about features related to inheritance, polymorphism, and interfaces. While exploring inheritance, you learn about Java's ultimate superclass. Also, while exploring interfaces, you discover why they were included in the Java language; interfaces are not merely a workaround for Java's lack of support for multiple implementation inheritance, but serve a higher purpose.

Chapter 5 introduces you to four categories of advanced language features: nested types, packages, static imports, and exceptions.

Chapter 6 introduces you to three additional advanced language feature categories: annotations, generics, and enums.

Chapter 7 begins a trend that focuses more on APIs than language features. This chapter first introduces you to Java's Math-oriented types. It then explores Number and its various subtypes (such as Integer, Double, and BigDecimal). Next you explore the string-oriented types (String, StringBuffer, and StringBuilder) followed by the System type. Finally, you explore the Thread class and related types for creating multithreaded applications.

Chapter 8 continues to explore Java's basic APIs by focusing on the Random class for generating random numbers; the References API, Reflection, and the StringTokenizer class for breaking a string into smaller components; and the Timer and TimerTask classes for occasionally or repeatedly executing tasks.

Chapter 9 focuses exclusively on Java's Collections Framework, which provides you with a solution for organizing objects in lists, sets, queues, and maps. You also learn about collection-oriented utility classes and review Java's legacy collection types.

Chapter 10 introduces the functional programming features which entered the Java world with Java 8. We also introduce the streaming API, which closely connects to functional programming.

Chapter 11 focuses exclusively on Java's Concurrency Utilities. After receiving an introduction to this framework, you explore executors, synchronizers (such as countdown latches), concurrent collections, the locking framework, and atomic variables (where you discover compare-and-swap).

Chapter 12 is all about classic input/output (I/O), largely from a file perspective. In this chapter, you explore classic I/O in terms of the File class, RandomAccessFile class, various stream classes, and various writer/reader classes. Our discussion of stream I/O includes coverage of Java's object serialization and deserialization mechanisms.

Chapter 13 continues to explore classic I/O by focusing on networks. You learn about the Socket, ServerSocket, DatagramSocket, and MulticastSocket classes along with related types. You also learn about the URL class for achieving networked I/O at a higher level and learn about the related URI class. After learning about the low-level NetworkInterface and InterfaceAddress classes, you explore cookie management, in terms of the CookieHandler and CookieManager classes, and the CookiePolicy and CookieStore interfaces.

Chapter 14 introduces you to new I/O. You learn about buffers, channels, selectors, regular expressions, charsets, and the Formatter and Scanner types in this chapter.

Chapter 15 focuses on databases. You first learn about the Apache Derby and SQLite database products, and then explore JDBC for communicating with databases created via these products.

Chapter 16 emphasizes Java's support for XML and JSON. We first provide a tutorial on the XML topic where you learn about the XML declaration, elements and attributes, character references and CDATA sections, namespaces, comments and processing instructions, well-formed documents, and valid documents (in terms of document

FOURTH EDITION NOTES

type definition and XML Schema). We then show you how to parse XML documents via the SAX API, parse and create XML documents via the DOM API, use the XPath API to concisely select nodes via location path expressions, and transform XML documents via XSLT. We then talk about JSON generation and parsing.

Chapter 17 completes the chapter portion of this book by covering Java's date and time APIs. We talk about the old API, and the new API which was introduced with Java 8.

Appendix A presents solutions to all of the exercises in Chapters 1 through 17.

Note You can download this book's source code by pointing your web browser to www.apress.com/us/book/9781484259429 and clicking the **Download Source Code** button.

What Comes Next?

After you complete this book, we recommend that you check out Apress's other Android-oriented books, such as *Beginning Android* by Grant Allen, and learn more about developing Android apps.

Thanks for purchasing this fourth edition of *Learn Java for Android Development*. We hope you find it a helpful preparation for, and we wish you lots of success in achieving, a satisfying and lucrative career as an Android app developer.

Jeff Friesen and Peter Späth, May 2020

CHAPTER 1

Getting Started with Java

Android apps are written in Java and use various Java application program interfaces (APIs). Because you'll want to write your own apps, but may be unfamiliar with the Java language and these APIs, this book teaches you about Java as a first step into Android app development. It provides you with Java language fundamentals and Java APIs that are useful when developing apps.

Note This book illustrates Java concepts via non-Android Java applications. It's easier for beginners to grasp these applications than corresponding Android apps. However, we also reveal a trivial Android app toward the end of this chapter for comparison purposes.

An *API* is an interface that application code uses to communicate with other code, which is typically stored in a software library. For more information on this term, check out Wikipedia's "Application programming interface" topic at http://en.wikipedia.org/wiki/Application_programming_interface.

This chapter sets the stage for teaching you the essential Java concepts that you need to understand before embarking on an Android app development career. We first answer the question: "What is Java?" Next, we show you how to install the Java SE Development Kit (JDK) and introduce you to JDK tools for compiling and running Java applications.

After presenting a few simple example applications, we show you how to install and use the open source Eclipse IDE (integrated development environment) so that you can more easily (and more quickly) develop Java applications and (eventually) Android apps. We then provide you with a brief introduction to Android and show you how Java fits into the Android development paradigm.

What Is Java?

Java is a language and a platform originated by Sun Microsystems and later acquired by Oracle. In this section, we briefly describe this language and reveal what it means for Java to be a platform. To meet various needs, Oracle organized Java into two main editions: Java SE and Java EE. A third edition, Java ME for embedded devices, plays no prominent role nowadays. This section briefly explores each of these two editions.

Java Is a Language

Java is a language in which developers express *source code* (program text). Java's *syntax* (rules for combining symbols into language features) is partly patterned after the C and C++ languages in order to shorten the learning curve for C/C++ developers.

The following list identifies a few similarities between Java and C/C++:

- Java and C/C++ share the same single-line and multiline comment styles. Comments let you document source code.
- Many of Java's reserved words are identical to their C/C++ counterparts (`for`, `if`, `switch`, and `while` are examples) and C++ counterparts (`catch`, `class`, `public`, and `try` are examples).
- Java supports character, double-precision floating-point, floating-point, integer, long integer, and short integer primitive types via the same `char`, `double`, `float`, `int`, `long`, and `short` reserved words.
- Java supports many of the same operators, including arithmetic (`+`, `-`, `*`, `/`, and `%`) and conditional (`? :`) operators.
- Java uses brace characters (`{` and `}`) to delimit blocks of statements.

The following list identifies a few of the differences between Java and C/C++:

- Java supports an additional comment style known as Javadoc.
- Java provides reserved words not found in C/C++ (`extends`, `strictfp`, `synchronized`, and `transient` are examples).

- Java doesn't require machine-specific knowledge. It supports the byte integer type (see [http://en.wikipedia.org/wiki/Integer_\(computer_science\)](http://en.wikipedia.org/wiki/Integer_(computer_science))), doesn't provide a signed version of the character type, and doesn't provide unsigned versions of integer, long integer, and short integer. Furthermore, all of Java's primitive types have guaranteed implementation sizes, which is an important part of achieving portability (discussed later). The same cannot be said of equivalent primitive types in C and C++.
- Java provides operators not found in C/C++. These operators include `instanceof` and `>>>` (unsigned right shift).

You'll learn about single-line, multiline, and Javadoc comments in Chapter 2. Also, you'll learn about reserved words, primitive types, operators, blocks, and statements in that chapter.

Java was designed to be a safer language than C/C++. It achieves safety in part by not letting you overload operators and by omitting C/C++ features such as *pointers* (storage locations containing addresses; see [http://en.wikipedia.org/wiki/Pointer_\(computer_programming\)](http://en.wikipedia.org/wiki/Pointer_(computer_programming))).

Java also achieves safety by modifying certain C/C++ features. For example, loops must be controlled by Boolean expressions instead of integer expressions where 0 is false and a nonzero value is true. (There is a discussion of loops and expressions in Chapter 2.)

These and other fundamental language features support classes, objects, inheritance, polymorphism, and interfaces. Java also provides advanced features related to nested types, packages, static imports, exceptions, assertions, annotations, generics, enums, and more. Subsequent chapters explore most of these language features.

Java Is a Platform

Java is a platform consisting of a virtual machine and an execution environment. The *virtual machine* is a software-based processor that presents an instruction set, and it is commonly referred to as the *Java virtual machine (JVM)*. The *execution environment* consists of libraries for running programs and interacting with the underlying operating system (also known as the *native platform*).

The execution environment includes a huge library of prebuilt class files that perform common tasks, such as math operations (e.g., trigonometry) and network communications. This library is commonly referred to as the *standard class library*.

A special Java program known as the *Java compiler* translates source code into *object code* consisting of instructions that are executed by the JVM and associated data. These instructions are known as *bytecode*. Figure 1-1 shows this translation process.

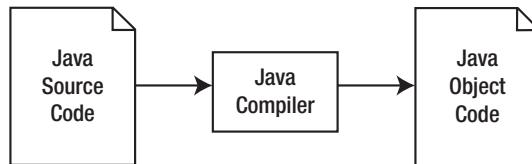


Figure 1-1. The Java compiler translates Java source code into Java object code consisting of bytecode and associated data

The compiler stores a program's bytecode and data in files having the `.class` extension. These files are known as *class files* because they typically store the compiled equivalent of classes, a language feature discussed in Chapter 3.

A Java program executes via a tool that loads and starts the JVM and passes the program's main class file to the machine. The JVM uses its *classloader* component to load the class file into memory.

After the class file has been loaded, the JVM's *bytecode verifier* component makes sure that the class file's bytecode is valid and doesn't compromise security. The verifier terminates the JVM when it finds a problem with the bytecode.

Assuming that all is well with the class file's bytecode, the JVM's *interpreter* component interprets the bytecode one instruction at a time. *Interpretation* consists of identifying bytecode instructions and executing equivalent native instructions.

Note *Native instructions* (also known as *native code*) are the instructions understood by the native platform's physical processor.

When the interpreter learns that a sequence of bytecode instructions is executed repeatedly, it informs the JVM's *just-in-time (JIT) compiler* to compile these instructions into native code.

JIT compilation is performed only once for a given sequence of bytecode instructions. Because the native instructions execute instead of the associated bytecode instruction sequence, the program executes much faster.

During execution, the interpreter might encounter a request to execute another class file's bytecode. When that happens, it asks the classloader to load the class file and the bytecode verifier to verify the bytecode before executing that bytecode.

Also during execution, bytecode instructions might request that the JVM open a file, display something on the screen, or perform another task that requires cooperation with the native platform. The JVM responds by transferring the request to the platform via its *Java Native Interface (JNI)* bridge to the native platform. Figure 1-2 shows these JVM tasks.

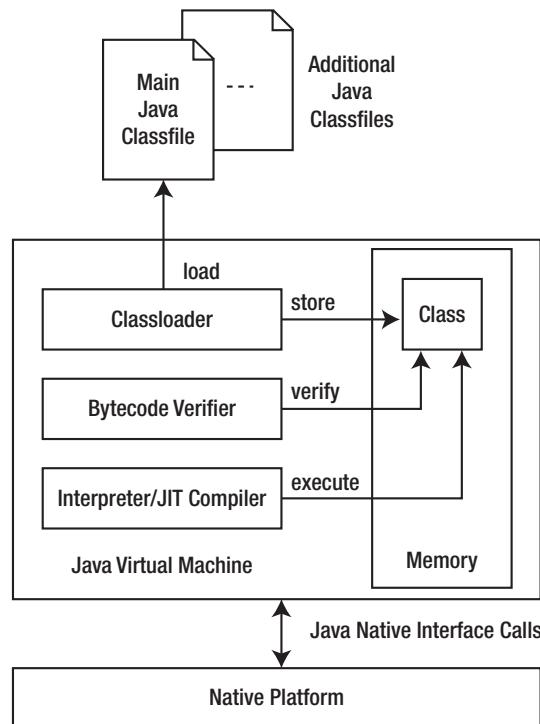


Figure 1-2. The JVM provides all of the necessary components for loading, verifying, and executing a class file

The platform side of Java promotes *portability* by providing an abstraction over the underlying platform. As a result, the same bytecode runs unchanged on Windows, Linux, Mac OS X, and other platforms.

Note Java was introduced with the slogan “write once, run anywhere.” Although Java goes to great lengths to enforce portability (such as defining an integer always to be 32 binary digits [bits] and a long integer always to be 64 bits (see <http://en.wikipedia.org/wiki/Bit> to learn about binary digits), it doesn’t always succeed. For example, despite being mostly platform independent, certain parts of Java (such as the scheduling of threads, discussed in Chapter 7) vary from underlying platform to underlying platform.

The platform side of Java also promotes *security* by doing its best to provide a secure environment (such as the bytecode verifier) in which code executes. The goal is to prevent malicious code from corrupting the underlying platform (and possibly stealing sensitive information).

Note Many security issues that have plagued Java have prompted Oracle to release various security updates.

Java SE and Java EE

Developers use different editions of the Java platform to create Java programs that run on desktop computers and web servers.

- *Java Platform, Standard Edition (Java SE)*: The Java platform for developing *applications*, which are stand-alone programs that run on desktops.
- *Java Platform, Enterprise Edition (Java EE)*: The Java platform for developing enterprise-oriented applications and *servlets*, which are server programs that conform to Java EE’s Servlet API. Java EE is built on top of Java SE.

This book largely focuses on Java SE and applications.

Note The open source variant of Java SE gets called OpenJDK; the open source variant of Java EE has the name Jakarta EE.

Installing the JDK and Exploring Example Applications

The Java SE edition can be downloaded from Oracle at www.oracle.com/technetwork/java/javase/overview/index.html. It contains everything needed to compile and run Java programs. For the corresponding open source variant, go to <https://openjdk.java.net/install/>.

Click the appropriate Download button to download the current JDK's installer application for your platform. Then run this application to install the JDK.

The installation directory contains various files, depending on the version you have chosen. For example, for JSE 13 you will find, among others, the following two important subdirectories:

- **bin:** This subdirectory contains assorted JDK tools. You'll use only a few of these tools in this book, mainly `javac` (Java compiler) and `java` (Java application launcher). However, you'll also work with `jar` (Java ARchive [JAR] creator, updater, and extractor [a *JAR file* is a ZIP file with special features]), `javadoc` (Java documentation generator), and `serialver` (serial version inspector).
- **lib:** This subdirectory contains Java and native platform library files that are used by JDK tools. Here you can also find the sources.

Note `javac` is not actually the Java compiler. It's a tool that loads and starts the JVM, identifies the compiler's main class file to the JVM, and passes the name of the source file being compiled to the compiler's main class file.

You can execute JDK tools at the *command line*, passing *command-line arguments* to a tool. For a quick refresher on the command line and command-line arguments topics, check out Wikipedia's "Command-line interface" entry (http://en.wikipedia.org/wiki/Command-line_interface).

The following command line shows you how to use `javac` to compile a source file named `App.java`:

```
javac App.java
```

The `.java` file extension is mandatory. The compiler complains when you omit this extension.

Tip You can compile multiple source files by specifying an asterisk in place of the file name, as follows:

```
javac *.java
```

Assuming success, an `App.class` file is created. If this file describes an application, which minimally consists of a single class containing a method named `main`, you can run the application as follows:

```
java App
```

You must not specify the `.class` file extension. The `java` tool complains when `.class` is specified.

In addition to downloading and installing the JDK, you'll need to access the JDK documentation, especially to explore the Java APIs. There are two sets of documentation that you can explore.

- Oracle's JDK documentation at <https://docs.oracle.com/javase/>
- Google's Java Android API documentation at
<https://developer.android.com/reference/packages.html>

Oracle's JDK documentation presents many APIs that are not supported by Android. Furthermore, it doesn't cover APIs that are specific to Android. This book focuses only on core Oracle Java APIs that are also covered in Google's documentation.

Hello, World!

It's customary to start exploring a new language and its tools by writing, compiling, and running a simple application that outputs the "Hello, World!" message. This practice dates back to Brian Kernighan's and Dennis Ritchie's seminal book, *The C Programming Language*.

Listing 1-1 presents the source code to a `HelloWorld` application that outputs this message.

Listing 1-1. Saying Hello in a Java Language Context

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

This short five-line application has a lot to say about the Java language. We'll briefly explain each feature, leaving comprehensive discussions of these features to later chapters.

This source code declares a *class*, which you can think of as a container for describing an application. The first line, `public class HelloWorld`, introduces the name of the class (`HelloWorld`), which is preceded by *reserved words* (names that have meaning to the Java compiler and which you cannot use to name other things in your programs) `public` and `class`. These reserved words respectively tell the compiler that `HelloWorld` must be stored in a file named `HelloWorld` and that a class is being declared.

The rest of the class declaration appears between a pair of brace characters (`{}`), which are familiar to C and C++ developers. Between these characters is the declaration of a single *method*, which you can think of as a named sequence of code. This method is named `main` to signify that it's the entry point into the application, and it is the analog of the `main()` function in C and C++.

The `main()` method includes a header that identifies this method and a block of code located between an open brace character (`{`) and a close brace character (`}`). Besides naming this method, the header provides the following information:

- `public`: This reserved word makes `main()` visible to the startup code that calls this method. If `public` wasn't present, the compiler would output an error message stating that it couldn't find a `main()` method.
- `static`: This reserved word causes this method to associate with the class instead of associating with any objects (discussed in Chapter 3) created from this class. Because the startup code that calls `main()` doesn't create an object from the class to call this method, it requires that the method be declared `static`. Although the compiler will not report an error when `static` is missing, it will not be possible to run `HelloWorld`, which will not be an application when the proper `main()` method doesn't exist.

- `void`: This reserved word indicates that the method doesn't return a value. If you change `void` to a type's reserved word (such as `int`) and then insert code that returns a value of this type (such as `return 0;`), the compiler will not report an error. However, you won't be able to run `HelloWorld` because the proper `main()` method wouldn't exist. (We discuss types in Chapter 2.)
- `(String[] args)`: This parameter list consists of a single parameter named `args`, which is of array type `String[]`. Startup code passes a sequence of command-line arguments to `args`, which makes these arguments available to the code that executes within `main()`. You'll learn about parameters and arguments in Chapter 3.

`main()` is called with an array of *strings* that identify the application's command-line arguments. These strings are stored in `String`-based array variable `args`. (We discuss method calling, arrays, and variables in Chapters 2 and 3.) Although the array variable is named `args`, there's nothing special about this name. You could choose another name for this variable.

`main()` presents a single line of code, `System.out.println("Hello, World!");`, which is responsible for outputting `Hello, World!` in the command window from where `HelloWorld` is run. From left to right, this *method call* accomplishes the following tasks:

- `System` identifies a standard class of system utilities.
- `out` identifies an object variable located in `System` whose methods let you output values of various types optionally followed by a newline (also known as line feed) character to the standard output stream. (In reality, a platform-dependent line terminator sequence is output. On Windows platforms, this sequence consists of a carriage return character [integer code 13] followed by a line feed character [integer code 10]. On Linux platforms, this sequence consists of a line feed character. On Mac OS X systems, this sequence consists of a carriage return character. It's convenient to refer to this sequence as a newline.)
- `println` identifies a method that prints its "Hello, World!" argument (the starting and ending double quote characters are not written; these characters delimit but are not part of the string) followed by a newline to the standard output stream.

Note The standard output stream is part of *standard I/O* (http://en.wikipedia.org/wiki/Standard_streams), which also consists of standard input and standard error streams, and which originated with the Unix operating system. Standard I/O makes it possible to read text from different sources (keyboard or file) and write text to different destinations (screen or file).

Text is read from the standard input stream, which defaults to the keyboard but can be redirected to a file. Text is written to the standard output stream, which defaults to the screen but can be redirected to a file. Error message text is written to the standard error stream, which defaults to the screen but can be redirected to a file that differs from the standard output file.

Assuming that you're familiar with your platform's command-line interface and are at the command line, make `HelloWorld` your current directory and copy Listing 1-1 to a file named `HelloWorld.java`. Then compile this source file via the following command line:

```
javac HelloWorld.java
```

Assuming that you've included the `.java` extension, which is required by `javac`, and that `HelloWorld.java` compiles, you should discover a file named `HelloWorld.class` in the current directory. Run this application via the following command line:

```
java HelloWorld
```

If all goes well, you should see the following line of output on the screen:

Hello, World!

You can redirect this output to a file by specifying the greater than angle bracket (`>`) followed by a file name. For example, the following command line stores the output in a file named `hello.txt`:

```
java HelloWorld >hello.txt
```

DumpArgs

In the previous example, we pointed out `main()`'s (`String[] args`) parameter list, which consists of a single parameter named `args`. This parameter stores an *array* (think sequence of values) of arguments passed to the application on the command line. Listing 1-2 presents the source code to a `DumpArgs` application that outputs each argument.

Listing 1-2. Dumping Command-Line Arguments Stored in `main()`'s `args` Array to the Standard Output Stream

```
public class DumpArgs {
    public static void main(String[] args) {
        System.out.println("Passed arguments:");
        for (int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

Listing 1-2's `DumpArgs` application consists of a class named `DumpArgs` that's very similar to Listing 1-1's `HelloWorld` class. The essential difference between these classes is the *for loop* (a construct for repeated execution and starting with reserved word `for`) that accesses each array item and dumps it to the standard output stream.

The `for` loop first initializes integer variable `i` to 0. This variable keeps track of how far the loop has progressed (the loop must end at some point), and it also identifies one of the entries in the `args` array. Next, `i` is compared with `args.length`, which records the number of entries in the array. The loop ends when `i`'s value equals the value of `args.length`. (We discuss `.length` in Chapter 2.)

Each loop iteration executes `System.out.println(args[i]);`. The string is stored in the `i`th entry of the `args` array. The first entry is located at *index* (location) 0. The last entry is stored at index `args.length - 1` and is accessed and then output to the standard output stream. Finally, `i` is incremented by 1 via `i++`, and `i < args.length` is reevaluated to determine whether the loop continues or ends.

Assuming that you're familiar with your platform's command-line interface and that you are at the command line, make `DumpArgs` your current directory and copy Listing 1-2 to a file named `DumpArgs.java`. Then compile this source file via the following command line:

```
javac DumpArgs.java
```

Assuming that you've included the `.java` extension, which is required by `javac`, and that `DumpArgs.java` compiles, you should discover a file named `DumpArgs.class` in the current directory. Run this application via the following command line:

```
java DumpArgs
```

If all goes well, you should see the following line of output on the screen:

Passed arguments:

For more interesting output, you'll need to pass command-line arguments to `DumpArgs`. For example, execute the following command line, which specifies Curly, Moe, and Larry as three arguments to pass to `DumpArgs`:

```
java DumpArgs Curly Moe Larry
```

This time, you should see the following expanded output on the screen:

Passed arguments:

Curly

Moe

Larry

You can redirect this output to a file. For example, the following command line stores the `DumpArgs` application's output in a file named `out.txt`:

```
java DumpArgs Curly Moe Larry >out.txt
```

EchoText

The previous two examples introduced you to a few Java language features, and they also showed outputting text to the standard output stream, which defaults to the screen but can be redirected to a file. In the final example (see Listing 1-3), we introduce more language features and demonstrate inputting text from the standard input stream and outputting text to the standard error stream.

Listing 1-3. Echoing Text Read from Standard Input to Standard Output

```

public class EchoText {
    public static void main(String[] args) {
        boolean isRedirect = false;
        if (args.length != 0)
            isRedirect = true;
        int ch;
        try {
            while ((ch = System.in.read()) != ((isRedirect) ? -1 : '\n'))
                System.out.print((char) ch);
        } catch (java.io.IOException ioe) {
            System.err.println("I/O error");
        }
        System.out.println();
    }
}

```

EchoText is a more complex application than HelloWorld or DumpArgs. Its `main()` method first declares a Boolean (true/false) variable named `isRedirect` that tells this application whether input originates from the keyboard (`isRedirect` is false) or a file (`isRedirect` is true). The application defaults to assuming that input originates from the keyboard.

There's no easy way to determine if standard input has been redirected, and so the application requires that the user tell it if this is the case by specifying one or more command-line arguments. The *if decision* (a construct for making decisions and starting with reserved word `if`) evaluates `args.length != 0`, assigning true to `isRedirect` when this Boolean expression evaluates to true (at least one command-line argument has been specified).

`main()` now introduces the `int` variable `ch` to store the integer representation of each character read from standard input. (You'll learn about `int` and integer in Chapter 2.) It then enters a sequence of code prefixed by the reserved word `try` and surrounded by brace characters. Code within this block may throw an *exception* (an object describing a problem) and the subsequent `catch` block will handle it (to address the problem). (We discuss exceptions in Chapter 5.)

The try block consists of a *while loop* (a construct for repeated execution and starting with the reserved word `while`) that reads and echoes characters. The loop first calls `System.in.read()` to read a character and assign its integer value to `ch`. The loop ends when this value equals -1 (no more input data is available from a file; standard input was redirected) or '`\n`' (the newline/line feed character) has been read, which is the case when standard input wasn't redirected. '`\n`' is an example of a character literal, which is discussed in Chapter 2.

For any other value in `ch`, this value is converted to a character via `(char)`, which is an example of Java's cast operator (discussed in Chapter 2). The character is then output via `System.out.print()`, which doesn't also terminate the current line by outputting a newline. The final `System.out.println();` call terminates the current line without outputting any content.

When standard input is redirected to a file and `System.in.read()` is unable to read text from the file (perhaps the file is stored on a removable storage device that has been removed before the read operation), `System.in.read()` fails by throwing a `java.io.IOException` object that describes this problem. The code within the catch block is then executed, which outputs an I/O error message to the standard error stream via `System.err.println("I/O error");`.

Note `System.err` provides the same families of `println()` and `print()` methods as `System.out`. You should only switch from `System.out` to `System.err` when you need to output an error message so that the error messages are displayed on the screen, even when standard output is redirected to a file.

Compile Listing 1-3 via the following command line:

```
javac EchoText.java
```

Now run the application via the following command line:

```
java EchoText
```

You should see a flashing cursor. Type the following text and press Enter:

This is a test.

You should see this text duplicated on the following line and the application should end.

Continue by redirecting the input source to a file, by specifying the less than angle bracket (<) followed by a file name:

```
java EchoText <EchoText.java x
```

Although it looks like there are two command-line arguments, there is only one: x. (Redirection symbols followed by file names don't count as command-line arguments.) You should observe the contents of EchoText.java listed on the screen.

Finally, execute the following command line:

```
java EchoText <EchoText.java
```

This time, x isn't specified, so input is assumed to originate from the keyboard. However, because the input is actually coming from the file EchoText.java, and because each line is terminated with a newline, only the first line from this file will be output.

Note If we had shortened the while loop expression to `while ((ch = System.in.read()) != -1)` and didn't redirect standard input to a file, the loop wouldn't end because -1 would never be seen. To exit this loop, you would have to press the Ctrl and C keys simultaneously on a Windows platform or the equivalent keys on a non-Windows platform.

Installing and Exploring the Eclipse IDE

Working with the JDK's tools at the command line is probably okay for small projects. However, this practice isn't recommended for large projects, which are hard to manage without the help of an IDE.

An *IDE* consists of a project manager for managing a project's files, a text editor for entering and editing source code, a debugger for locating bugs, and other features. Although Google favors the Android Studio for app development, we briefly describe another popular IDE: Eclipse. The usage scenarios of the Studio and Eclipse do not differ very much, but for Java language learning, Eclipse is the more straightforward choice.

Note For convenience, we use JDK tools throughout this book, except for this section where we discuss and demonstrate the Eclipse IDE.

Eclipse IDE is an open source IDE for developing programs in Java and other languages (such as C, COBOL, PHP, Perl, and Python). Eclipse IDE for Java Developers is one distribution of this IDE that's available for download; version 2019-12 is the current version at time of this writing.

You should download and install Eclipse IDE for Java Developers to follow along with this section's Eclipse-oriented example. Begin by pointing your browser to www.eclipse.org/downloads/ and navigate to the version applicable for your desktop operating system.

After installing Eclipse, run this application. You should discover a splash screen identifying this IDE and a dialog box that lets you choose the location of a workspace for storing projects followed by a main window like the one shown in Figure 1-3.

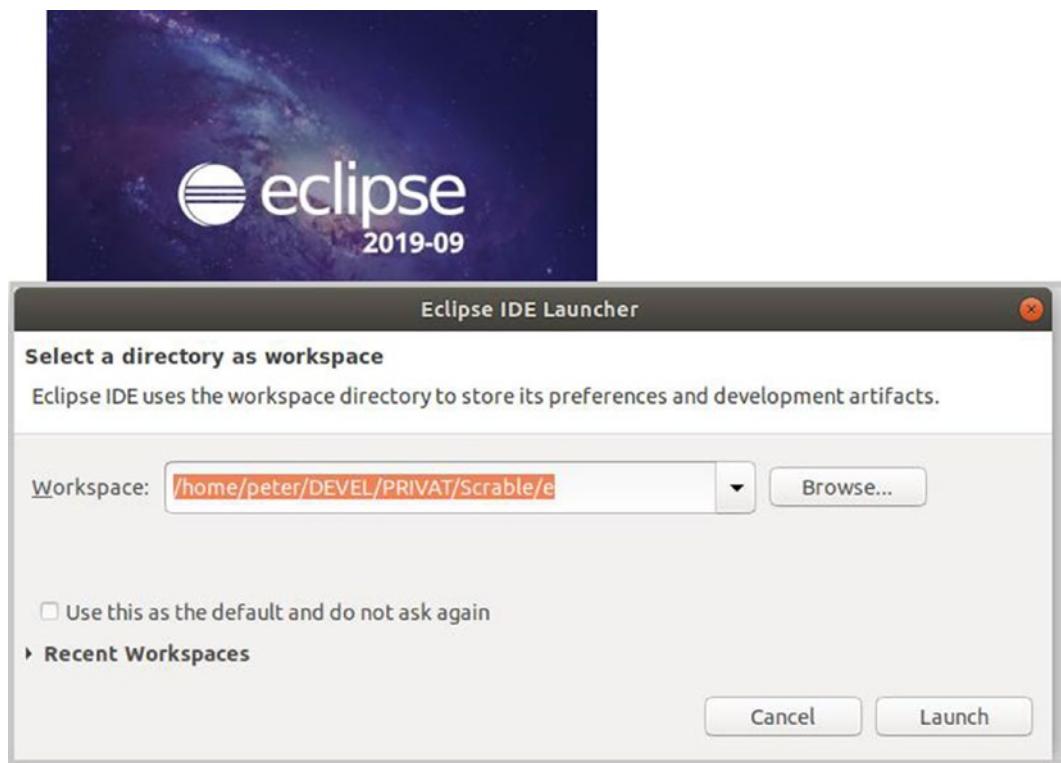


Figure 1-3. Keep the default workspace or choose another workspace

Click the OK button, and you're taken to Eclipse's main window. See Figure 1-4.

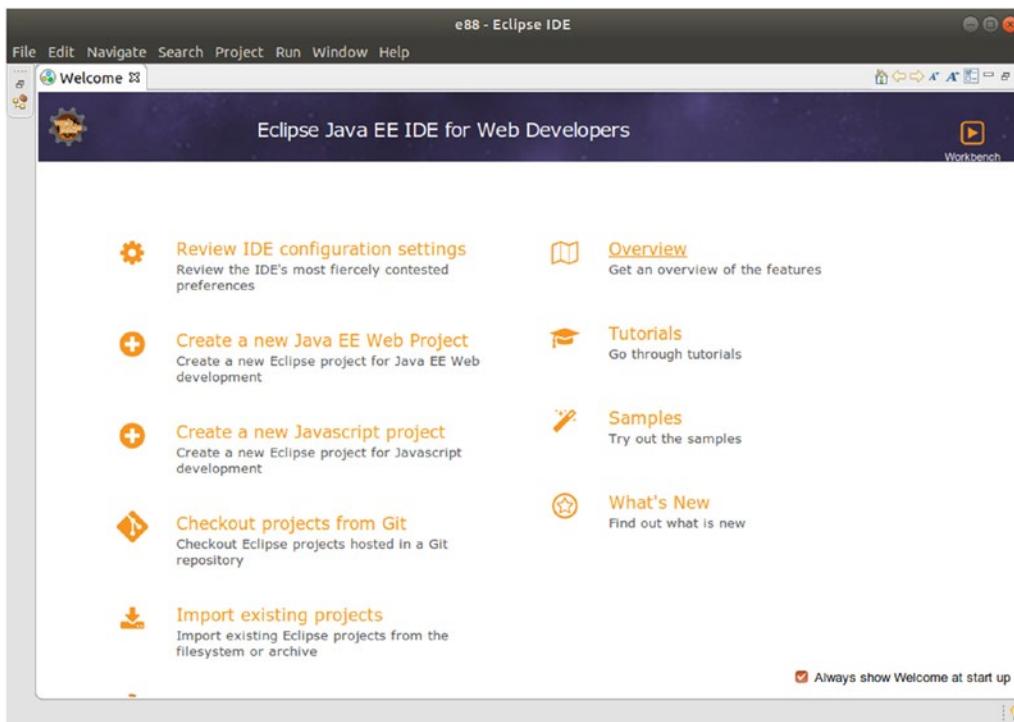


Figure 1-4. The main window initially presents a Welcome tab

The main window initially presents a Welcome tab from which you can learn more about Eclipse. Click this tab's X icon to close this tab; you can restore the Welcome tab by selecting Welcome from the menu bar's Help menu.

The Eclipse user interface is based on a main window that consists of a menu bar, a toolbar, a workbench area, and a status bar. The *workbench* presents windows for organizing Eclipse projects, editing source files, viewing messages, and more.

To help you get comfortable with the Eclipse user interface, we'll show you how to create a `DumpArgs` project containing a single `DumpArgs.java` source file with Listing 1-2's source code. You'll also learn how to compile and run this application.

Complete the following steps to create the `DumpArgs` project:

1. Select New from the File menu and then Project... and Java Project from the resulting pop-up menu.
2. In the resulting New Java Project dialog box, enter **DumpArgs** into the Project name text field. Keep all of the other defaults, and click the Finish button.

After the second step (and after closing the Welcome tab), you'll see a workbench similar to the one shown in Figure 1-5.

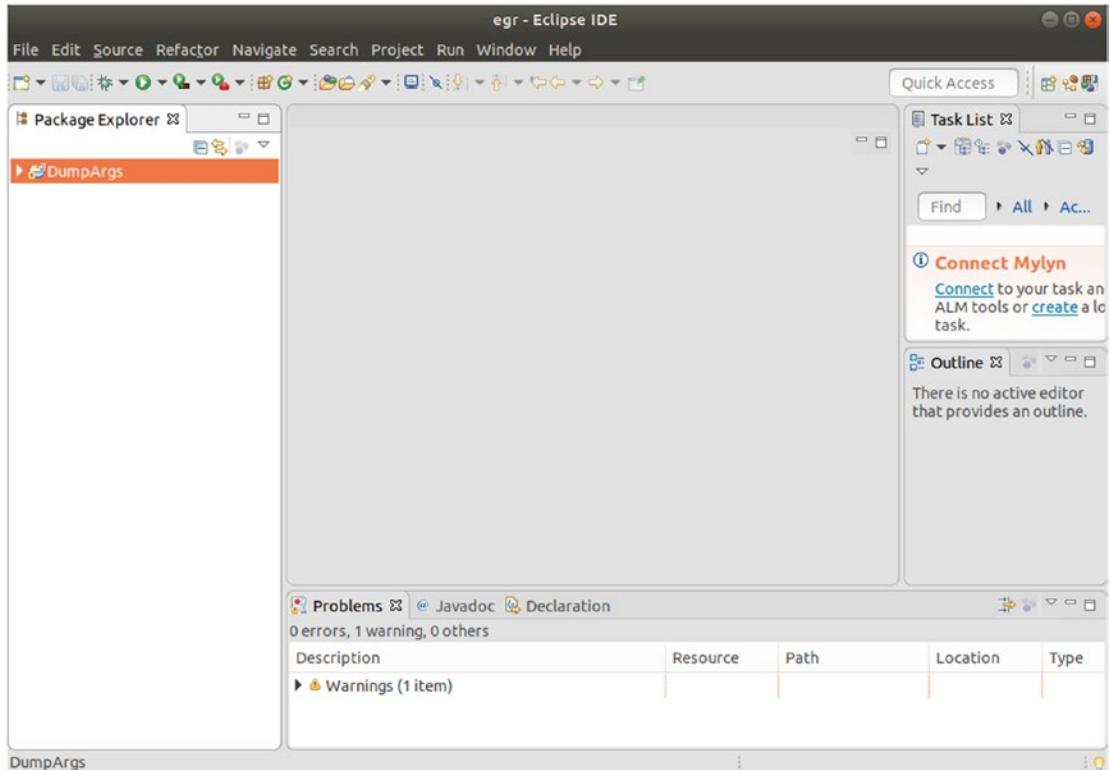


Figure 1-5. A DumpArgs entry appears in the workbench's Package Explorer

On the left side of the workbench, you'll see a window titled Package Explorer. This window identifies the workspace's projects in terms of packages (discussed in Chapter 5). At the moment, only a single DumpArgs entry appears in this window.

Clicking the triangle icon to the left of DumpArgs expands this entry to reveal src and JRE System Library items. The src item stores the DumpArgs project's source files, and the JRE System Library item identifies various JRE files that are used to run this application.

You'll now add a new file named `DumpArgs.java` to src.

1. Highlight src, and select New and Class from the resulting pop-up menu.
2. In the resulting dialog box, enter **DumpArgs** into the name text field, and click the Finish button.

CHAPTER 1 GETTING STARTED WITH JAVA

Eclipse responds by displaying an editor window titled DumpArgs.java. Copy Listing 1-2 content to this window. Then compile and run this application by selecting Run and Run As Java Application from the Run menu. (If you see a Save and Launch dialog box, click OK to close this dialog box.) The output will be shown in the Console view.

You must pass command-line arguments to DumpArgs to see additional output from this application. You can accomplish this task via these steps:

1. Select Run Configurations... from the Run menu.
2. In the resulting Run Configurations dialog box, select the Arguments tab.
3. Enter **Curly Moe Larry** into the Program arguments text area, and click the Close button. See Figure 1-6.

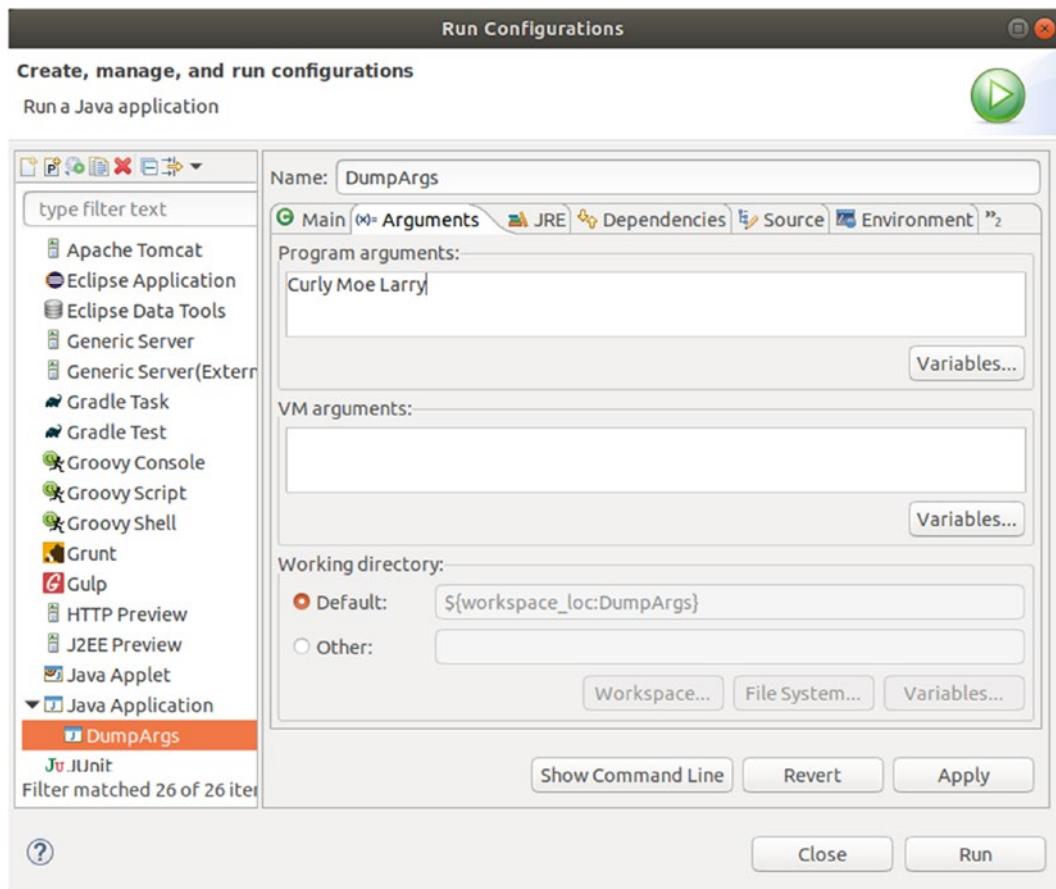


Figure 1-6. Arguments are entered into the Program arguments text area

Once again, select Run As Java Application from the Run menu to run the DumpArgs application. This time, the Console tab reveals Curly, Moe, and Larry on separate lines below "Passed arguments:".

This is all we have to say about the Eclipse IDE. For more information, study the tutorials via the Welcome tab, access IDE help via the Help menu, and explore the Eclipse documentation at www.eclipse.org/documentation/.

Java Meets Android

In the previous editions of this book, we provided an introduction to Java language features and assorted APIs that are helpful when developing Android apps. Apart from a few small references to various Android items, we didn't delve into Android. This edition still explores Java language features and APIs that are useful in Android app development. However, it also introduces you to Android.

In this section, we first answer the "What is Android?" question. We next review Android's history and talk about its various releases. After exploring Android's architecture, we present the Android version of HelloWorld.

What Is Android?

Android is Google's software stack for mobile devices. This stack consists of apps (such as Browser and Contacts), a virtual machine in which apps run, *middleware* (software that sits on top of the operating system and provides various services to the virtual machine and its apps), and a Linux-based operating system.

Android offers the following features:

- An application framework enabling reuse and replacement of app components
- Bluetooth, mobile network and Wi-Fi support (hardware dependent)
- Camera, GPS, compass, and accelerometer support (hardware dependent)
- Android RunTime (ART) virtual machine optimized for mobile devices
- GSM telephony support (hardware dependent)
- Integrated web browser

- Media support for common audio, video, and still image formats (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- Optimized graphics powered by a custom 2D graphics library; 3D graphics based on OpenGL ES 1.0, 1.1, 2.0, or 3.0 (hardware acceleration optional)
- SQLite for structured data storage (We introduce SQLite in Chapter 14.)

Although not part of the software stack, Android's rich development environment provided by Android Studio could also be considered an Android feature.

History of Android

Contrary to what you might expect, Android didn't originate with Google. Instead, Android, Inc., a small Palo Alto, California-based startup company, initially developed Android. Google bought this company in the summer of 2005 and released a beta version of the Android SDK in November 2007.

On September 23, 2008, Google released Android 1.0, whose core features included a web browser, camera support, Google Search, Wi-Fi and Bluetooth support, and more. This release corresponds to API Level 1. (An *API level* is a 1-based integer that uniquely identifies the API revision offered by an Android version; it's a way of distinguishing one significant Android release from another.)

Note The Wikipedia entry for Android gives you quite a good overview about various Android releases. Or enter “android version history” in your favorite search engine.

Android Architecture

The Android software stack consists of apps at the top, a Linux kernel with various drivers at the bottom, and middleware (an application framework, libraries, and the Android runtime) in the center. Figure 1-7 shows this layered architecture.

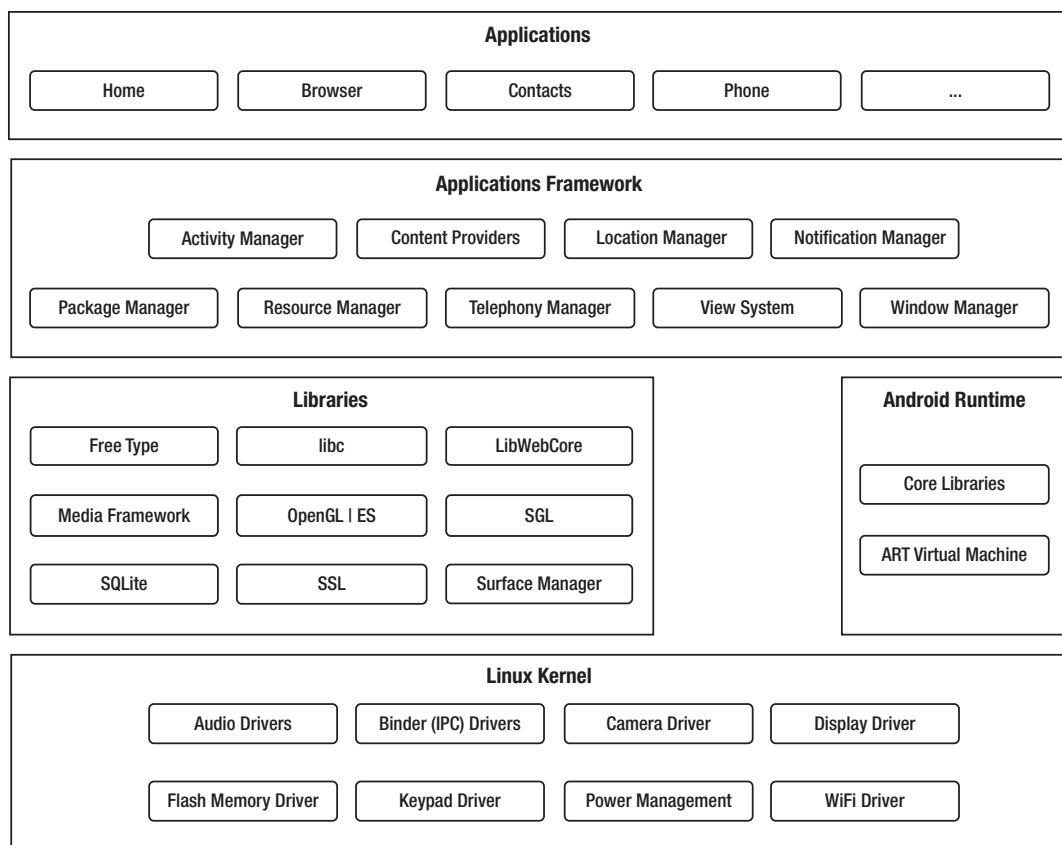


Figure 1-7. *Android's layered architecture consists of several major parts*

Users care very much about apps, and Android ships with a variety of useful core apps, which include Browser, Contacts, and Phone. All apps are written in the Java programming language. Apps form the top layer of Android's architecture.

Android doesn't officially recognize Java language features newer than Java 8, which is why we don't discuss them in this book. Regarding APIs, this platform supports many APIs from Java 8 and previous Java versions. Also, Android provides its own unique APIs.

Directly beneath the app layer is the *application framework*, a set of high-level building blocks for creating apps. The application framework is preinstalled on Android devices and consists of the following components:

- *Activity Manager:* This component provides an app's *life cycle* and maintains a shared activity stack for navigating within and among apps.

- *Content Providers*: These components encapsulate data (such as the Browser app's bookmarks) that can be shared among apps.
- *Location Manager*: This component makes it possible for an Android device to be aware of its physical location.
- *Notification Manager*: This component lets an app notify the user of a significant event (such as a message's arrival) without interrupting what the user is currently doing.
- *Package Manager*: This component lets an app learn about other app packages that are currently installed on the device.
- *Resource Manager*: This component lets an app access its resources.
- *Telephony Manager*: This component lets an app learn about a device's telephony services. It also handles making and receiving phone calls.
- *View System*: This component manages user interface elements and user interface-oriented event generation.
- *Window Manager*: This component organizes the screen's real estate into windows, allocates drawing surfaces, and performs other window-related jobs.

The components of the application framework rely on a set of C/C++ libraries to perform their functions. Developers interact with the following libraries by way of framework APIs:

- *FreeType*: This library supports bitmap and vector font rendering.
- *libc*: This library is a BSD-derived implementation of the standard C system library, tuned for embedded Linux-based devices.
- *LibWebCore*: This library offers a modern and fast web browser engine that powers the Android browser and an embeddable web view. It's based on WebKit (<http://en.wikipedia.org/wiki/WebKit>), and the Google Chrome and Apple Safari browsers also use it.

- *Media Framework*: These libraries, which are based on PacketVideo's OpenCORE, support the playback and recording of many popular audio and video formats, as well as working with static image files. Supported formats include MPEG4, H.264, MP3, AAC, AMR, JPEG, PNG, and GIF.
- *OpenGL / ES*: These 3D graphics libraries provide an OpenGL implementation based on OpenGL ES 1.0/1.1/2.0/3.0 APIs. They use hardware 3D acceleration (where available) or the included (and highly optimized) 3D software rasterizer.
- *SGL*: This library provides the underlying 2D graphics engine.
- *SQLite*: This library provides a powerful and lightweight relational database engine that's available to all apps and that's also used by Mozilla Firefox and Apple's iPhone for persistent storage.
- *SSL*: This library provides secure sockets layer-based security for network communication.
- *Surface Manager*: This library manages access to the display subsystem and seamlessly composites 2D and 3D graphic layers from multiple apps.

Android provides a runtime environment that consists of core libraries (implementing a subset of the Apache Harmony Java version 5 implementation) and the ART *virtual machine* (a non-JVM).

Each Android app defaults to running in its own Linux *process* (executing application), which hosts an instance of ART. This virtual machine has been designed so that devices can run multiple virtual machines efficiently.

Note Android starts a process when any part of the app needs to execute, and it shuts down the process when it's no longer needed and system resources are required by other apps.

Perhaps you're wondering how it's possible to have a non-JVM run Java code. The answer is that ART doesn't run Java code. Instead, Android transforms compiled Java class files into a special ART binary format, and it's this resulting code that gets executed by ART.

Finally, the libraries and Android runtime rely on the Linux kernel for underlying core services, such as threading, low-level memory management, a network stack, process management, and a driver model. Furthermore, the kernel acts as an abstraction layer between the hardware and the rest of the software stack.

ANDROID SECURITY MODEL

Android's architecture includes a security model that prevents apps from performing operations that are considered harmful to other apps, Linux, or users. This security model is mostly based on process-level enforcement via standard Linux features (such as user and group IDs) and places processes in a security *sandbox*.

By default, the sandbox prevents apps from reading or writing the user's private data (such as contacts or emails), reading or writing another app's files, performing network access, keeping the device awake, accessing the camera, and so on. Apps that need to access the network or perform other sensitive operations must first obtain permission to do so.

Android handles permission requests in various ways, typically by automatically allowing or disallowing the request based upon a certificate or by prompting the user to grant or revoke the permission. Permissions required by an app are declared in the app's manifest file so that they are known to Android when the app is installed. These permissions won't subsequently change.

Android Says Hello

Earlier in this chapter, we introduced you to `HelloWorld`, a Java application that outputs "Hello, World!" Because you might be curious about its Android equivalent, check out Listing 1-4.

Listing 1-4. The Android Equivalent of `HelloWorld`

```
public class HelloWorld extends android.app.Activity {  
    public void onCreate(android.os.Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        System.out.println("Hello, World!");  
    }  
}
```

Listing 1-4 isn't too different from Listing 1-1, but there are some significant changes. For one thing, `HelloWorld` *extends* another class named `Activity` (stored in a package named `android.app`—see Chapter 5 for a discussion of packages). By extending `Activity`, `HelloWorld` proclaims itself as an *activity*, which you can think of as a user interface screen. (We discuss extension in Chapter 4.)

By extending `Activity`, `HelloWorld` *inherits* that class's `create()` method, which Android calls when creating the activity. `HelloWorld` *overrides* this method with its own implementation so that it can output the “Hello, World!” message. However, `create()` first needs to execute `Activity`'s version of this method (via `super.onCreate()`), so that the activity is properly initialized.

Note “Hello, World!” isn't displayed on an Android device's screen. Instead, it's written to a log file that can be examined by Android's `adb` tool.

`HelloWorld`, `DumpArgs`, and `EchoText` demonstrate `public static void main(String[] args)` as a Java application's entry point. This is where the application's execution begins. In contrast, as you've just seen, an Android app doesn't require this method for its entry point because the app's architecture is very different.

EXERCISES

The following exercises are designed to test your understanding of Chapter 1's content:

1. What is Java?
2. What is a virtual machine?
3. What is the purpose of the Java compiler?
4. True or false: A class file's instructions are commonly referred to as *bytecode*.

5. What does the JVM's interpreter do when it learns that a sequence of bytecode instructions is being executed repeatedly?
 6. How does the Java platform promote portability?
 7. How does the Java platform promote security?
 8. True or false: Java SE is the Java platform for developing servlets.
 9. Which JDK tool is used to compile Java source code?
 10. Which JDK tool is used to run Java applications?
 11. What is standard I/O?
 12. How do you specify the `main()` method's header?
 13. What is an IDE? Identify the IDE that Google supports for developing Android apps.
 14. What is Android?
-

Summary

Java is a language and a platform. The language is partly patterned after the C and C++ languages to shorten the learning curve for C/C++ developers. The platform consists of a virtual machine and associated execution environment.

The Java language shares several similarities with C/C++, such as presenting the same single-line and multiline comments and offering various reserved words that are also found in C/C++. However, there are differences, such as providing `>>>` and other operators not found in C/C++.

The Java platform includes a huge library of prebuilt class files that perform common tasks, such as math operations (e.g., trigonometry) and network communications. This library is commonly referred to as the standard class library.

A special Java program known as the Java compiler translates source code into object code consisting of instructions that are executed by the JVM and associated data. These instructions are known as bytecode.

Developers use different editions of the Java platform to create Java programs that run on desktop computers and web servers. These editions are known as Java SE and Java EE.

The JDK provides tools (including the Java compiler) for developing and running Java programs.

Working with the JDK's tools at the command line isn't recommended for large projects, which are hard to manage without the help of an integrated development environment. Eclipse is a popular IDE used for that purpose. Another IDE, Android Studio, is specially tailored for Android development.

Android is Google's software stack for mobile devices. This stack consists of apps, a virtual machine in which apps run, middleware that sits on top of the operating system and provides various services to the virtual machine and its apps, and a Linux-based operating system.

Android didn't originate with Google. Instead, Android, Inc., a small Palo Alto, California-based startup company, initially developed Android. Google bought this company in the summer of 2005, and it released Android 1.0 on September 23, 2008.

Android's architecture is based on an application layer, an application framework, libraries, an Android runtime (consisting of core libraries implementing a subset of the Apache Harmony Java version 5 implementation and the ART virtual machine), and a Linux kernel.

Android doesn't officially recognize Java language features newer than Java 8, which is why we don't discuss them in this book. Regarding APIs, this platform supports many APIs from Java 8 and previous Java versions. Also, Android provides its own unique APIs.

Chapter 2 introduces you to the Java language by focusing on this language's fundamentals. You'll learn about comments, identifiers, types, variables, expressions, statements, and more.

CHAPTER 2

Learning Language Fundamentals

Aspiring Android app developers need to understand the Java language in which an app's source code is written. This chapter introduces you to this language by focusing on its fundamentals. Specifically, you'll learn about application structure, comments, identifiers (and reserved words), types, variables, expressions (and literals), and statements.

Note The American Standard Code for Information Interchange (ASCII) has traditionally been used to encode a program's source code. Because ASCII is limited to the English language, Unicode (<http://unicode.org/>) was developed as a replacement. *Unicode* is a computing industry standard for consistently encoding, representing, and handling text that's expressed in most of the world's writing systems. Because Java supports Unicode, non-English-oriented symbols can be integrated into or accessed from Java source code.

Learning Application Structure

Chapter 1 introduced you to three small Java applications. Each application exhibited a similar structure that we employ throughout this book. Before developing Java applications, you need to understand this structure, which Listing 2-1 presents. Throughout this chapter, we present code fragments that you can paste into this structure to create working applications.

Listing 2-1. Structuring a Java Application

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

An application is based on a class declaration. (We discuss classes in Chapter 3.) The declaration begins with a header consisting of `public`, followed by `class`, followed by `X`, where `X` is a placeholder for the actual name, for example, `HelloWorld`. The header is followed by a pair of braces (`{` and `}`) that denote the class's body.

Between these braces is a special method declaration (we discuss methods in Chapter 3), which defines the application's entry point. It starts with a header that consists of `public`, followed by `static`, followed by `void`, followed by `main`, followed by `(String[] args)`. A pair of braces follows this header and denotes the method's body. The `...` represents code that you specify to execute.

You can pass a sequence of arguments to the application when executing it at the command line. These string-based arguments are stored in the `args` array (a *string* is a character sequence delimited by double quote `"`) characters). We introduce arrays later in this chapter and further discuss them in Chapter 3. There's nothing special about `args`: we could choose another name for it, for example, `arguments`.

You must store this class declaration in a file whose name matches `X` and has a `.java` file extension. You would then compile the source code as follows:

```
javac X.java
```

`X` is a placeholder for the actual class name. Also, the “`.java`” file extension is mandatory.

Assuming that compilation succeeds, which results in a class file named `X.class` being created, you would subsequently run the application as follows:

```
java X
```

Replace `X` with the actual class name. Don't specify the “`.class`” file extension.

If you need to pass command-line arguments to the application, specify them after the class name according to the following pattern:

```
java X arg1 arg2 arg3 ...
```

Here, *arg1*, *arg2*, and *arg3* are placeholders for three command-line arguments. The trailing ... signifies additional arguments (if any).

Finally, if you need to specify a sequence of words as a single argument, place these words between double quotes to prevent java from treating them as separate arguments, like so:

```
java X "These words constitute a single argument."
```

Learning Comments

Source code needs to be documented so that you (and any others who have to maintain it) can understand it, now and later. Source code should be documented while being written and whenever it's modified. If these modifications impact existing documentation, the documentation must be updated so that it accurately explains the code.

Java provides the *comment* feature for embedding documentation in source code. When the source code is compiled, the Java compiler ignores all comments; no bytecodes are generated. Single-line, multiline, and Javadoc comments are supported.

Single-Line Comments

A *single-line comment* occupies all or part of a single line of source code. This comment begins with the // character sequence and continues with explanatory text. The compiler ignores everything from // to the end of the line in which // appears.

The following example presents a single-line comment:

```
double x = Math.sqrt(10 * 10 + 20 * 20); // Distance from (0, 0) to (10, 20).
```

This example calculates the distance between the (0, 0) origin and the point (10, 20) in the Cartesian x/y plane (sqrt() calculates the square root).

Note Single-line comments are useful for inserting short but meaningful explanations of source code into this code. Don't use them to insert redundant documentation. For example, when declaring a variable, don't insert a meaningless comment such as // This variable stores integer values.

Multiline Comments

A *multiline comment* occupies one or more lines of source code. This comment begins with the `/*` character sequence, continues with explanatory text, and ends with the `*/` character sequence. Everything from `/*` through `*/` is ignored by the compiler.

The following example demonstrates a multiline comment:

```
/*
 A year is a leap year when it's divisible by 400, or divisible by 4 and
 not also divisible by 100.
*/
System.out.println(year % 400 == 0 || (year % 4 == 0 && year % 100 != 0));
```

This example assumes the existence of an integer variable (discussed later) named `year` that stores an arbitrary four-digit year. It evaluates a complex expression (discussed later) that determines whether the year is leap or not. The expression returns true (leap year) or false (not leap year), which is output by `System.out.println()`. The multiline comment explains what constitutes a leap year.

Caution You cannot place one multiline comment inside another. For example,
`/*/* Nesting multiline comments is illegal! */*/` isn't a valid
multiline comment.

Javadoc Comments

Java supports a third kind of comment that simplifies the specification of external documentation (e.g., an HTML export). You'll find this Javadoc comment feature helpful in the preparation of technical documentation for other developers who rely on your Java applications, libraries, and other Java-based software products.

A *Javadoc comment* occupies one or more lines of source code. This comment begins with the `/**` character sequence, continues with explanatory text, and ends with the `*/` character sequence. Everything from `/**` through `*/` is ignored by the compiler. You usually let any line inside the comment start with an `*` (asterisk)—this is not a must, but a usual convention.

The following example demonstrates a Javadoc comment:

```
/**
 * Application entry point
 *
 * @param args array of command-line arguments passed to this method
 */
public static void main(String[] args) {
    // TODO code application logic here
}
```

This example begins with a Javadoc comment that describes the `main()` method. Sandwiched between `/**` and `*/` is a description of the method and the `@param` *Javadoc tag* (an `@`-prefixed instruction to the `javadoc` tool).

The following list identifies several commonly used tags:

- `@author` identifies the source code's author.
- `@deprecated` identifies a source code entity (such as a method) that should no longer be used.
- `@param` identifies one of a method's parameters.
- `@see` provides a see-also reference.
- `@since` identifies the software release where the entity first originated.
- `@return` identifies the kind of value that the method returns.
- `@throws` documents an exception thrown from a method. (We discuss exceptions in Chapter 5.)

[Listing 2-2](#) presents Chapter 1's `DumpArgs` application source code with Javadoc comments that describe the `DumpArgs` class and its `main()` method.

Listing 2-2. Documenting an Application Class and Its `main()` Method

```
/**
 * Dump all command-line arguments to standard output.
 *
 * @author Jeff Friesen
 */
public class DumpArgs {
```

```
/**  
 * Application entry point.  
 *  
 * @param args array of command-line arguments.  
 */  
public static void main(String[] args) {  
    System.out.println("Passed arguments:");  
    for (int i = 0; i < args.length; i++)  
        System.out.println(args[i]);  
}  
}
```

You can extract these documentation comments into a set of HTML files by using the JDK's `javadoc` tool as follows:

```
javadoc DumpArgs.java
```

It also generates several files, including the `index.html` documentation entry-point file. Point your browser to this file, and you should see a page similar to that shown in Figure 2-1.

The screenshot shows a Mozilla Firefox browser window titled "DumpArgs - Mozilla Firefox". The address bar displays "file:///home/peter/tmp/egr/Dump ***". The page content is the documentation for the `DumpArgs` class. The top navigation bar includes links for PACKAGE, CLASS (which is selected), USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar are links for SUMMARY, NESTED, FIELD, CONSTR, and METHOD, and DETAIL, FIELD, CONSTR, and METHOD. A search bar is present at the top right. The main content starts with the package declaration `java.lang.Object` followed by the class definition `DumpArgs`. It then describes the class as "Dump all command-line arguments to standard output." An "Author:" section lists Jeff Friesen. Below this is a "Constructor Summary" table with one entry: `DumpArgs()`. At the bottom is a "Method Summary" table with tabs for All Methods, Static Methods, and Concrete Methods.

Constructor	Description
<code>DumpArgs()</code>	

All Methods	Static Methods	Concrete Methods
-------------	----------------	------------------

Figure 2-1. The entry-point page into `DumpArgs`'s documentation describes this class

Learning Identifiers

Source code entities such as classes and methods need to be named so that they can be referenced from elsewhere in the code. Java provides the identifiers feature for this purpose.

An *identifier* consists of letters (A-Z, a-z), digits (0-9), and an underscore. This name must begin with a letter or an underscore.

Examples of valid identifiers include the following:

- i
- counter
- j2
- _for

Examples of invalid identifiers include the following:

- 1name (starts with a digit)
- first#name (# isn't a valid identifier symbol)

Note Java is a *case-sensitive language*, which means that identifiers differing in case are considered separate identifiers. For example, temperature and Temperature are separate identifiers.

Almost any valid identifier can be chosen to name a class, method, or other source code entity. However, some identifiers are reserved for special purposes; they are known as *reserved words*. Java reserves the following identifiers:

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static

strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		

The compiler outputs an error message when you attempt to use any of these reserved words outside of their usage contexts. Also, `const` and `goto` are not used by the Java language.

Listing 2-1 revealed several identifiers: `public`, `class`, `X`(a placeholder for an identifier), `static`, `void`, `main`, `String`, and `args`. Identifiers `public`, `class`, `static`, and `void` are also reserved words.

Note Most of Java's reserved words are also known as *keywords*. The three exceptions are `false`, `null`, and `true`, which are examples of *literals* (values specified verbatim).

Learning Types

Applications process data items, such as integers, floating-point values, characters, and strings. Data items are classified according to various characteristics. For example, integers are whole numbers without fractions. Also, a string is a sequence of characters that's treated as a unit and possesses a length that identifies the number of characters in the sequence.

Java uses the term *type* to describe classifications of data items. A *type* identifies a set of data items (and their representation in memory) and a set of operations that transform these data items into other data items of that set. For example, the integer type identifies numeric values with no fractional parts and integer-oriented math operations, such as adding two integers to yield another integer.

Note Java is a *strongly typed language*, which means that every expression, variable, and so on has a type known to the compiler. This capability helps the compiler detect type-related errors at compile time rather than having these errors manifest themselves at runtime. Expressions and variables are discussed later in this chapter.

Java recognizes primitive types, user-defined types, and array types. These types are defined in the following sections.

Primitive Types

A *primitive type* is a type that's defined by the language and whose values are not objects. Java supports the Boolean, character, byte integer, short integer, integer, long integer, floating-point, and double-precision floating-point primitive types. They are described in Table 2-1.

Table 2-1. Primitive Types

Primitive Type	Reserved Word	Size	Min Value	Max Value
Boolean	Boolean	--	--	--
Character	Char	16-bit	Unicode 0	Unicode 65,535
Byte integer	Byte	8-bit	-128	+127
Short integer	Short	16-bit	-32,768	+32,767
Integer	Int	32-bit	-2,147,483,648	+2,147,483,647
Long integer	Long	64-bit	-9,223,372,036, 854,775,808	+9,223,372,036,854, 775,807
Floating-point	Float	32-bit	IEEE 754	IEEE 754
Double-precision floating-point	Double	64-bit	IEEE 754	IEEE 754

Table 2-1 describes each primitive type in terms of its reserved word, size, minimum value, and maximum value. A “--” entry indicates that the column in which it appears isn't applicable to the primitive type described in that entry's row.

The size column identifies the size of each primitive type in terms of the number of *bits* (binary digits; each digit is either 0 or 1) that a value of that type occupies in memory. Except for Boolean (whose size is implementation dependent; one Java implementation might store a Boolean value in a single bit, whereas another implementation might require an 8-bit *byte* for performance efficiency), each primitive type's implementation has a specific size.

BINARY VS. DECIMAL

Computers process numbers encoded via the *binary number system*, which is a base-2 number system in which there are only two digits: 0 and 1. In contrast, people process numbers according to the *decimal number system*, which is a base-10 number system in which there are ten digits: 0 through 9.

The minimum value and maximum value columns identify the smallest and largest values that can be represented by each type. Except for Boolean (whose only values are true and false), each primitive type has a minimum value and a maximum value.

The minimum and maximum values of the character type refer to Unicode. **Unicode 0** is shorthand for “the first Unicode code point”; a *code point* is an integer that represents a symbol (such as A) or a control character (such as newline or tab) or that combines with other code points to form a symbol.

Note The character type’s limits imply that this type is *unsigned* (all character values are positive). In contrast, each numeric type is *signed* (it supports positive and negative values).

The minimum and maximum values of the byte integer, short integer, integer, and long integer types reveal that there is one more negative value than positive value (0 is typically not regarded as a positive value). The reason for this imbalance has to do with how integers are represented (*two’s complement representation*).

Java represents an integer value as a combination of a *sign bit* (the leftmost bit; 0 for a positive value and 1 for a negative value) and *magnitude bits* (all remaining bits to the right of the sign bit). When the sign bit is 0, the magnitude is stored directly. However, when the sign bit is 1, the magnitude is stored using *two’s complement representation* in which all 1s are flipped to 0s, all 0s are flipped to 1s, and 1 is added to the number behind the minus sign.

The minimum and maximum values of the floating-point and double-precision floating-point types refer to *Institute of Electrical and Electronics Engineers (IEEE) 754*, which is a standard for representing floating-point values in memory. Check out Wikipedia’s “IEEE 754-2008” entry (http://en.wikipedia.org/wiki/IEEE_754) to learn more about this standard.

Note Developers who argue that Java should support objects only aren't happy about the inclusion of primitive types in the language. However, Java was designed to include primitive types to overcome the speed and memory limitations of early 1990s-era devices, to which Java was originally targeted.

Object Types

An object *type* is a type that's often used to model a real-world concept (such as a color or a bank account). The developer defines it using a class, an interface, an enum, or an annotation type; and its values are objects. (We discuss classes in Chapter 3, interfaces in Chapter 4, and enums and annotation types in Chapter 6.)

For example, you could create a `Color` class to model colors; its values could describe colors as red/green/blue components and its methods (see Chapter 3) could return these components.

Note You can think of Chapter 1's `HelloWorld`, `DumpArgs`, and `EchoText` classes as examples of object types. However, these classes aren't used to create objects but describe applications instead.

Java's `String` class defines the string type and is a member of the standard class library. Its values describe character sequences, and its methods perform string operations such as joining two strings. Unlike user-defined types, `String` enjoys language support for initializing `String` variables and joining strings into a single string. You'll see examples of this later in the chapter.

Object types are also known as *reference types* because a variable of that type stores a *reference* (a memory address or some other identifier) to a region of memory that stores an object of that type. In contrast, variables of primitive types store the values directly; they don't store references to these values.

Array Types

An *array type* is a special reference type that signifies an *array*, a region of memory that stores values in equal-size and contiguous slots, which are commonly referred to as *elements*. This type consists of the *element type* (a primitive type, object type, or array type) and one or more pairs of square brackets that indicate the number of *dimensions* (extents). A single pair of brackets signifies a *one-dimensional array*, two pairs of brackets signify a *two-dimensional array*, three pairs of brackets signify a one-dimensional array of two-dimensional arrays, and so on. For example, `int[]` signifies a one-dimensional array (with `int` as the element type), and `double[][]` signifies a two-dimensional array (with `double` as the element type).

Learning Variables

Applications manipulate values that are stored in memory, which is symbolically represented in source code through the use of the variables feature. A *variable* is a named memory location that stores some type of value. A variable that stores a reference is often referred to as a *reference variable*.

Variables must be declared before they're used. A declaration minimally consists of a type name, optionally followed by a sequence of square bracket pairs, followed by a name, optionally followed by a sequence of square bracket pairs, and terminated with a semicolon character (`;`). Consider the following examples:

```
int counter;           // Declare integer variable counter.
double temperature;  // Declare double-precision floating-point variable
                     temperature.
String firstName;    // Declare String variable firstName.
int[] ages;          // Declare one-dimensional integer array variable ages.
char gradeLetters[]; // Declare one-dimensional character array variable
                     gradeLetters.
float[][] matrix;    // Declare two-dimensional floating-point array
                     variable matrix.
```

No string is yet associated with `firstName`, and no arrays are yet associated with `ages`, `gradeLetters`, and `matrix`.

Note Square brackets can appear after the type name or after the variable name, but not in both places. For example, the compiler reports an error when it encounters `int[] x[];`. It is common practice to place the square brackets after the type name (as in `int[] ages;`) instead of after the variable name (as in `char gradeLetters[];`), unless the array is being declared in a context such as `int x, y[], z;`.

You can declare multiple variables on one line by separating each variable from its predecessor with a comma, as demonstrated by the following example:

```
int x, y[], z;
```

This example declares three variables named `x`, `y`, and `z`. Each variable shares the same type, which happens to be integer. Unlike `x` and `z`, which store single integer values, `y[]` signifies a one-dimensional array whose element type is integer; each element stores an integer value. No array is yet associated with `y`.

The square brackets must appear after the variable name when the array is declared on the same line as the other variables. If you place the square brackets after the type name, as in `int[] x, y, z;`, all three variables signify one-dimensional arrays of integers.

Learning Expressions

The previously declared variables were not explicitly initialized to any values. As a result, they are either initialized to default values (such as 0 for `int` and 0.0 for `double`) or remain uninitialized, depending on the contexts in which they appear (declared within classes or declared within methods). In Chapter 3, we discuss variable contexts in terms of local variables, parameters, and fields.

Java provides the expressions feature for initializing variables and for other purposes. An *expression* is a combination of literals, variable names, method calls, and operators. At runtime, it evaluates to a value whose type is referred to as the expression's type. If the expression is being assigned to a variable, this type must agree with the variable's type; otherwise, the compiler reports an error.

Java recognizes simple expressions and compound expressions. These types are defined in the following sections.

Simple Expressions

A *simple expression* is a *literal* (a value expressed verbatim), the name of a variable (containing a value), or a method call (returning a value). Java supports several kinds of literals: string, Boolean true and false, character, integer, floating-point, and null.

Note A method call that doesn't return a value—the called method is known as a *void method*—is a special kind of simple expression, for example, `System.out.println("Hello, World!");`. This stand-alone expression cannot be assigned to a variable. Attempting to do so (as in `int i = System.out.println("X");`) causes the compiler to report an error.

A *string literal* consists of a sequence of Unicode characters surrounded by a pair of double quotes, for example, "The quick brown fox jumps over the lazy dog." It might also contain *escape sequences*, which are special syntax for representing certain printable and nonprintable characters that cannot otherwise appear in the literal. For example, "The quick brown \"fox\" jumps over the lazy dog." uses the \" escape sequence to surround fox with double quotes.

Table 2-2 describes all supported escape sequences.

Table 2-2. Escape Sequences

Escape Syntax	Description
\\"	Backslash
\\"	Double quote
\'	Single quote
\b	Backspace
\f	Form feed
\n	Newline (also referred to as line feed)
\r	Carriage return
\t	Horizontal tab

Finally, a string literal might contain *Unicode escape sequences*, which are special syntax for representing Unicode characters. A Unicode escape sequence begins with \u and continues with four hexadecimal digits (0–9, A–F, a–f) with no intervening space. For example, \u0041 represents capital letter A, and \u20ac represents the European Union's euro currency symbol.

A *Boolean literal* consists of reserved word true or reserved word false.

A *character literal* consists of a single Unicode character surrounded by a pair of single quotes ('A' is an example). You can also use, as a character literal, an escape sequence (e.g., '\') or a Unicode escape sequence (such as '\u0041').

An *integer literal* consists of a sequence of digits. If the literal is to represent a long integer value, it must be suffixed with an uppercase L or lowercase l (L is easier to read). If there is no suffix, the literal represents a 32-bit integer (an int).

Integer literals can be specified in the decimal, hexadecimal, and octal formats:

- The decimal format is the default format, for example, 127.
- The hexadecimal format requires that the literal begin with 0x or 0X and continue with hexadecimal digits (0–9, A–F, a–f), for example, 0x7F.
- The octal format requires that the literal be prefixed with 0 and continue with octal digits (0–7), for example, 0177.

For big numbers you can use an underscore as a thousands separator, as in 1_456_417.

A *floating-point literal* consists of an integer part, a decimal point (represented by the period [.]), a fractional part, an exponent (starting with letter E or e), and a type suffix (letter D, d, F, or f). Most parts are optional, but enough information must be present to differentiate the floating-point literal from an integer literal. Examples include 0.1 (double-precision floating-point), 89F (floating-point), 600D (double-precision floating-point), and 13.08E+23 (double-precision floating-point).

Finally, the null literal is assigned to a reference variable to indicate that the variable doesn't refer to an object.

Listing 2-3 presents a SimpleExpressions application that uses literals to initialize the previously presented variables.

Listing 2-3. Using Literals to Initialize Variables

```

public class SimpleExpressions {
    public static void main(String[] args) {
        int counter = 10;
        double temperature = 98.6; // Assume Fahrenheit scale.
        String firstName = "Mark";
        int[] ages = { 52, 28, 93, 16 };
        char gradeLetters[] = { 'A', 'B', 'C', 'D', 'F' };
        float[][] matrix = { { 1.0F, 2.0F, 3.0F }, { 4.0F, 5.0F, 6.0F } };
        int x = 1, y[] = { 1, 2, 3 }, z = 3;
        double pi = 3.14159;
        System.out.println(counter);
        System.out.println(temperature);
        System.out.println(ages.length);
        System.out.println(gradeLetters.length);
        System.out.println(matrix.length);
        System.out.println(x);
        System.out.println(y.length);
        System.out.println(z);
        System.out.println(pi);
    }
}

```

The first example assigns 32-bit integer literal 10 to 32-bit integer variable `counter`. The second example assigns double-precision floating-point literal 98.6 to double-precision floating-point variable `temperature`. The third example assigns string literal "Mark" to `String` variable `firstName`.

The fourth through seventh examples use array initializers (such as { 52, 28, 93, 16 }) to initialize arrays that are assigned to the `ages`, `gradeLetters`, `matrix`, and `y` array variables. An *array initializer* consists of a brace-and-comma-delimited list of expressions, which (as the `matrix` example shows) may be array initializers. The `matrix` example results in a table that looks like the following:

```

1.0F 2.0F 3.0F
4.0F 5.0F 6.0F

```

Each array variable is associated with a `.length` property that returns the number of elements in the array. For example, because `ages` contains 4 elements, `ages.length` returns 4. Similarly, because `matrix` contains 2 rows, `matrix.length` returns 2. We'll have more to say about this property and also show you how to access array elements later in this chapter.

You can then run this application via the following command line:

```
java SimpleExpressions
```

You should observe the following output:

```
10
98.6
4
5
2
1
3
3
3.14159
```

ORGANIZING VARIABLES IN MEMORY

Perhaps you're curious about how variables are organized in memory. Figure 2-2 presents one possible high-level organization for the `counter`, `ages`, and `matrix` variables, along with the arrays assigned to `ages` and `matrix`.

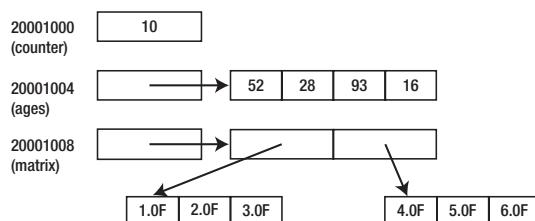


Figure 2-2. The `counter` variable stores a 4-byte integer value, whereas `ages` and `matrix` store 4-byte references to their respective arrays

Figure 2-2 reveals that each of counter, ages, and matrix is stored at a memory address (starting at a fictitious 20001000 value in this example) that's divisible by 4 (each variable stores a 4-byte value); that counter's 4-byte value is stored at this address; and that each of the ages and matrix 4-byte memory locations stores the 32-bit address of its respective array (64-bit addresses would most likely be used on 64-bit virtual machines). Also, a one-dimensional array is stored as a list of values, whereas a two-dimensional array is stored as a one-dimensional row array of addresses, where each address identifies a one-dimensional column array of values for that row.

In addition to assigning literals to variables, you can also assign variables and the results of method calls to variables, like so:

```
int counter2 = counter;
    // Assign previous counter variable value to counter2.
boolean isLeap = isLeapYear(2012);
    // Assign Boolean result of calling isLeapYear(2012) to isLeap.
```

These examples have assumed that only those expressions whose types are the same as the types of the variables that they are initializing can be assigned to those variables. However, under certain circumstances, it's possible to assign an expression having a different type. For example, Java permits you to assign certain integer literals to short integer variables, as in `short s = 20;`, and assign a short integer expression to an integer variable, as in `int i = s;`.

Java permits the former assignment because 20 can be represented as a short integer (no information is lost). In contrast, Java would complain about `short s = 40000;` because integer literal 40000 cannot be represented as a short integer (32767 is the maximum positive integer that can be stored in a short integer variable). Java permits the latter assignment because no information is lost when Java converts from a type with a smaller set of values to a type with a wider set of values.

Java supports the following primitive-type conversions via widening conversion rules:

- Byte integer to short integer, integer, long integer, floating-point, or double-precision floating-point
- Short integer to integer, long integer, floating-point, or double-precision floating-point

- Character to integer, long integer, floating-point, or double-precision floating-point
- Integer to long integer, floating-point, or double-precision floating-point
- Long integer to floating-point or double-precision floating-point
- Floating-point to double-precision floating-point

Listing 2-4 presents a `SimpleExpressions` application that demonstrates these additional insights into simple expressions.

Listing 2-4. Learning More About Simple Expressions

```
public class SimpleExpressions {  
    public static void main(String[] args) {  
        int counter = 30;  
        int counter2 = counter;  
        System.out.println(counter);  
  
        short s = 20;  
        System.out.println(s);  
        int i = s;  
        System.out.println(i);  
  
        // short s2 = 40000; // possible loss of precision error  
  
        int i2 = -1;  
        double d = i2;  
        System.out.println(d);  
    }  
}
```

This application demonstrates assigning one variable to another and assigning literal values to variables where the types don't match. For example, in the `double d = i2;` example, a widening conversion rule converts the 32-bit integer value stored in variable `i2` to a double-precision floating-point value that's assigned to variable `d`.

Compile Listing 2-4 as follows:

```
javac SimpleExpressions.java
```

Run this application via the following command line:

```
java SimpleExpressions
```

You should observe the following output:

```
30
20
20
-1.0
```

Note When converting from a smaller integer to a larger integer, Java copies the smaller integer's sign bit into the extra bits of the larger integer.

In Chapter 4, we discuss the widening conversion rules for performing type conversions in the contexts of user-defined and array types.

Compound Expressions

A *compound expression* is a sequence of simple expressions and operators, where an *operator* (a sequence of instructions symbolically represented in source code) transforms its *operand* expression value(s) into another value. For example, `-6` is a compound expression consisting of operator `-` and integer literal `6` as its operand. This expression transforms `6` into its negative equivalent. Similarly, `x + 5` is a compound expression consisting of variable name `x`, integer literal `5`, and operator `+` sandwiched between these operands. Variable `x`'s value is fetched and added to `5` when this expression is evaluated. The sum becomes the value of the expression.

Note When *x*'s type is byte integer or short integer, this variable's value is widened to an integer. However, when *x*'s type is long integer, floating-point, or double-precision floating-point, 5 is widened to the appropriate type. The addition operation is performed after the widening conversion takes place.

Java supplies many operators, which are classified by the number of operands that they take. A *unary operator* takes only one operand (unary minus [-] is an example), a *binary operator* takes two operands (addition [+] is an example), and Java's single *ternary operator* (conditional [?::]) takes three operands.

Operators are also classified as prefix, postfix, and infix. A *prefix operator* is a unary operator that precedes its operand (as in -6), a *postfix operator* is a unary operator that trails its operand (as in *x*++), and an *infix operator* is a binary or ternary operator sandwiched between the binary operator's two or the ternary operator's three operands (as in *x* + 5).

Table 2-3 presents all supported operators in terms of their symbols, descriptions, and precedence levels; we discuss the concept of precedence at the end of this section. Various operator descriptions refer to “integer type,” which is shorthand for specifying any of byte integer, short integer, integer, or long integer unless “integer type” is qualified as a 32-bit integer. Also, “numeric type” refers to any of these integer types along with floating-point and double-precision floating-point.

Table 2-3. Operators

Operator	Symbol	Description	Precedence
Addition	+	Given <i>operand1</i> + <i>operand2</i> , where each operand must be of character or numeric type, add <i>operand2</i> to <i>operand1</i> and return the sum.	10
Array index	[]	Given <i>variable</i> [<i>index</i>], where <i>index</i> must be of integer type, read value from or store value into <i>variable</i> 's storage element at location <i>index</i> .	13
Assignment	=	Given <i>variable</i> = <i>operand</i> , which must be <i>assignment-compatible</i> (their types must agree), store <i>operand</i> in <i>variable</i> .	0

(continued)

Table 2-3. (continued)

Operator	Symbol	Description	Precedence
Bitwise AND	&	Given <i>operand1</i> & <i>operand2</i> , where each operand must be of character or integer type, bitwise AND their corresponding bits and return the result. A result bit is set to 1 when each operand's corresponding bit is 1. Otherwise, the result bit is set to 0.	6
Bitwise complement	~	Given ~ <i>operand</i> , where <i>operand</i> must be of character or integer type, flip <i>operand</i> 's bits (1s to 0s and 0s to 1s) and return the result.	12
Bitwise exclusive OR	^	Given <i>operand1</i> ^ <i>operand2</i> , where each operand must be of character or integer type, bitwise exclusive OR their corresponding bits and return the result. A result bit is set to 1 when one operand's corresponding bit is 1 and the other operand's corresponding bit is 0. Otherwise, the result bit is set to 0.	5
Bitwise inclusive OR		Given <i>operand1</i> <i>operand2</i> , which must be of character or integer type, bitwise inclusive OR their corresponding bits and return the result. A result bit is set to 1 when either (or both) of the operand's corresponding bit is 1. Otherwise, the result bit is set to 0.	4
Cast	(<i>type</i>)	Given (<i>type</i>) <i>operand</i> , convert <i>operand</i> to an equivalent value that can be represented by <i>type</i> . For example, you could use this operator to convert a floating-point value to a 32-bit integer value.	12
Compound assignment	+ =, - =, * =, / =, % =, & =, =, ^ =, << =, >> =, >>> =	Given <i>variable operator operand</i> , where <i>operator</i> is one of the listed compound operator symbols and where <i>operand</i> is assignment-compatible with <i>variable</i> , perform the indicated operation using <i>variable</i> 's value as <i>operator</i> 's left operand value and store the resulting value in <i>variable</i> .	0

(continued)

Table 2-3. (continued)

Operator	Symbol	Description	Precedence
Conditional AND	? :	Given <i>operand1</i> ? <i>operand2</i> : <i>operand3</i> , where <i>operand1</i> must be of Boolean type, return <i>operand2</i> when <i>operand1</i> is true or <i>operand3</i> when <i>operand1</i> is false. The types of <i>operand2</i> and <i>operand3</i> must agree.	1
Conditional OR	&&	Given <i>operand1</i> && <i>operand2</i> , where each operand must be of Boolean type, return true when both operands are true. Otherwise, return false. When <i>operand1</i> is false, <i>operand2</i> isn't examined. This is known as <i>short-circuiting</i> .	3
Division		Given <i>operand1</i> <i>operand2</i> , where each operand must be of Boolean type, return true when at least one operand is true. Otherwise, return false. When <i>operand1</i> is true, <i>operand2</i> isn't examined. This is known as <i>short-circuiting</i> .	2
Equality	/	Given <i>operand1</i> / <i>operand2</i> , where each operand must be of character or numeric type, divide <i>operand1</i> by <i>operand2</i> and return the quotient.	11
Inequality	==	Given <i>operand1</i> == <i>operand2</i> , where both operands must be comparable (e.g., you cannot compare an integer with a string literal), compare both operands for equality. Return true when these operands are equal. Otherwise, return false.	7
	!=	Given <i>operand1</i> != <i>operand2</i> , where both operands must be comparable (e.g., you cannot compare an integer with a Boolean value), compare both operands for inequality. Return true when these operands are not equal. Otherwise, return false.	7

(continued)

Table 2-3. (continued)

Operator	Symbol	Description	Precedence
Left shift	<code><<</code>	Given <i>operand1</i> <code><<</code> <i>operand2</i> , where each operand must be of character or integer type, shift <i>operand1</i> 's binary representation left by the number of bits that <i>operand2</i> specifies. For each shift, a 0 is shifted into the rightmost bit and the leftmost bit is discarded. Only the 5 low-order bits of <i>operand2</i> are used when shifting a 32-bit integer (to prevent shifting more than the number of bits in a 32-bit integer). Only the 6 low-order bits of <i>operand2</i> are used when shifting a 64-bit integer (to prevent shifting more than the number of bits in a 64-bit integer). The shift preserves negative values. Furthermore, it's equivalent to (but faster than) multiplying by a multiple of 2.	9
Logical AND	<code>&</code>	Given <i>operand1</i> <code>&</code> <i>operand2</i> , where each operand must be of Boolean type, return true when both operands are true. Otherwise, return false. In contrast to conditional AND, logical AND doesn't perform short-circuiting.	6
Logical complement	<code>!</code>	Given <code>!</code> <i>operand</i> , where <i>operand</i> must be of Boolean type, flip <i>operand</i> 's value (true to false or false to true) and return the result.	12
Logical exclusive OR	<code>^</code>	Given <i>operand1</i> <code>^</code> <i>operand2</i> , where each operand must be of Boolean type, return true when one operand is true and the other operand is false. Otherwise, return false.	5
Logical inclusive OR	<code> </code>	Given <i>operand1</i> <code> </code> <i>operand2</i> , where each operand must be of Boolean type, return true when at least one operand is true. Otherwise, return false. In contrast to conditional OR, logical inclusive OR doesn't perform short-circuiting.	4

(continued)

Table 2-3. (continued)

Operator	Symbol	Description	Precedence
Member access	.	Given <i>identifier1</i> . <i>identifier2</i> , access the <i>identifier2</i> member of <i>identifier1</i> . You'll learn about this operator in Chapter 3.	13
Method call	()	Given <i>identifier</i> (<i>argument list</i>), call the method identified by <i>identifier</i> and matching parameter list. You'll learn about method calling in Chapter 3.	13
Multiplication	*	Given <i>operand1</i> * <i>operand2</i> , where each operand must be of character or numeric type, multiply <i>operand1</i> by <i>operand2</i> and return the product.	11
Object creation	New	Given new <i>identifier</i> (<i>argument list</i>), allocate memory for object and call constructor (discussed in Chapter 3) specified as <i>identifier</i> (<i>argument list</i>). Given new <i>identifier</i> [<i>integer size</i>], allocate a one-dimensional array of values.	12
Postdecrement	--	Given <i>variable</i> --, where <i>variable</i> must be of character or numeric type, subtract 1 from <i>variable</i> 's value (storing the result in <i>variable</i>) and return the original value.	13
Postincrement	++	Given <i>variable</i> ++, where <i>variable</i> must be of character or numeric type, add 1 to <i>variable</i> 's value (storing the result in <i>variable</i>) and return the original value.	13
Predecrement	--	Given -- <i>variable</i> , where <i>variable</i> must be of character or numeric type, subtract 1 from its value, store the result in <i>variable</i> , and return the new decremented value.	12
Preincrement	++	Given ++ <i>variable</i> , where <i>variable</i> must be of character or numeric type, add 1 to its value, store the result in <i>variable</i> , and return the new incremented value.	12

(continued)

Table 2-3. (continued)

Operator	Symbol	Description	Precedence
Relational greater than	>	Given <i>operand1</i> > <i>operand2</i> , where each operand must be of character or numeric type, return true when <i>operand1</i> is greater than <i>operand2</i> . Otherwise, return false.	8
Relational greater than or equal to	>=	Given <i>operand1</i> >= <i>operand2</i> , where each operand must be of character or numeric type, return true when <i>operand1</i> is greater than or equal to <i>operand2</i> . Otherwise, return false.	8
Relational less than	<	Given <i>operand1</i> < <i>operand2</i> , where each operand must be of character or numeric type, return true when <i>operand1</i> is less than <i>operand2</i> . Otherwise, return false.	8
Relational less than or equal to	<=	Given <i>operand1</i> <= <i>operand2</i> , where each operand must be of character or numeric type, return true when <i>operand1</i> is less than or equal to <i>operand2</i> . Otherwise, return false.	8
Relational type checking	Instanceof	Given <i>operand1</i> instanceof <i>operand2</i> , where <i>operand1</i> is an object and <i>operand2</i> is a class (or other user-defined type), return true when <i>operand1</i> is an instance of <i>operand2</i> . Otherwise, return false.	8
Remainder	%	Given <i>operand1</i> % <i>operand2</i> , where each operand must be of character or numeric type, divide <i>operand1</i> by <i>operand2</i> and return the remainder. Also known as the modulus operator.	11

(continued)

Table 2-3. (continued)

Operator	Symbol	Description	Precedence
Signed right shift	>>	Given <i>operand1</i> >> <i>operand2</i> , where each operand must be of character or integer type, shift <i>operand1</i> 's binary representation right by the number of bits that <i>operand2</i> specifies. For each shift, a copy of the sign bit (the leftmost bit) is shifted to the right and the rightmost bit is discarded. Only the 5 low-order bits of <i>operand2</i> are used when shifting a 32-bit integer (to prevent shifting more than the number of bits in a 32-bit integer). Only the 6 low-order bits of <i>operand2</i> are used when shifting a 64-bit integer (to prevent shifting more than the number of bits in a 64-bit integer). The shift preserves negative values. Furthermore, it's equivalent to (but faster than) dividing by a multiple of 2.	9
String concatenation	+	Given <i>operand1</i> + <i>operand2</i> , where at least one operand is of String type, append <i>operand2</i> 's string representation to <i>operand1</i> 's string representation and return the concatenated result.	10
Subtraction	-	Given <i>operand1</i> - <i>operand2</i> , where each operand must be of character or numeric type, subtract <i>operand2</i> from <i>operand1</i> and return the difference.	10
Unary minus	-	Given - <i>operand</i> , where <i>operand</i> must be of character or numeric type, return <i>operand</i> 's arithmetic negative.	12
Unary plus	+	Like its predecessor, but return <i>operand</i> . Rarely used.	12

(continued)

Table 2-3. (continued)

Operator	Symbol	Description	Precedence
Unsigned right shift	>>>	Given <i>operand1</i> >>> <i>operand2</i> , where each operand must be of character or integer type, shift <i>operand1</i> 's binary representation right by the number of bits that <i>operand2</i> specifies. For each shift, a zero is shifted into the leftmost bit and the rightmost bit is discarded. Only the 5 low-order bits of <i>operand2</i> are used when shifting a 32-bit integer (to prevent shifting more than the number of bits in a 32-bit integer). Only the 6 low-order bits of <i>operand2</i> are used when shifting a 64-bit integer (to prevent shifting more than the number of bits in a 64-bit integer). The shift doesn't preserve negative values. Furthermore, it's equivalent to (but faster than) dividing by a multiple of 2.	9

Table 2-3's operators can be classified as additive, array index, assignment, bitwise, cast, conditional, equality, logical, member access, method call, multiplicative, object creation, relational, shift, and unary minus/plus.

Additive Operators

The additive operators consist of addition (+), subtraction (-), postdecrement (--), postincrement (++), predecrement (--), preincrement (++), and string concatenation (+). Addition returns the sum of its operands (such as 6 + 4 returns 10), subtraction returns the difference between its operands (such as 6 - 4 returns 2 and 4 - 6 returns -2), postdecrement subtracts 1 from its variable operand and returns the variable's prior value (such as x--), postincrement adds 1 to its variable operand and returns the variable's prior value (such as x++), predecrement subtracts 1 from its variable operand and returns the variable's new value (such as --x), preincrement adds 1 to its variable operand and returns the variable's new value (such as ++x), and string concatenation merges its string operands and returns the merged string (such as "A" + "B" returns "AB").

The addition, subtraction, postdecrement, postincrement, predecrement, and preincrement operators can yield values that overflow or underflow the limits of the resulting value's type. For example, adding two large positive 32-bit integer values can produce a value that cannot be represented as a 32-bit integer value. The result is said to overflow. Java doesn't detect overflows and underflows.

Java provides a special widening conversion rule for use with string operands and the string concatenation operator. When either operand isn't a string, the operand is converted to a string prior to string concatenation. For example, when presented with "A" + 5, the compiler generates code that first converts 5 to "5" and then performs the string concatenation operation, resulting in "A5".

[Listing 2-5](#) presents a `CompoundExpressions` application that lets you start experimenting with the additive operators.

Listing 2-5. Experimenting with the Additive Operators

```
public class CompoundExpressions {
    public static void main(String[] args) {
        int age = 65;
        System.out.println(age + 32);
        System.out.println(++age);
        System.out.println(age--);
        System.out.println("A" + "B");
        System.out.println("A" + 5);
        short x = 32767;
        System.out.println(++x);
    }
}
```

[Listing 2-5](#)'s `main()` method first declares a 32-bit integer `age` variable that's initialized to 32-bit integer value 65. It then outputs the result of an expression that adds `age`'s value to 32-bit integer value 32.

The preincrement and postdecrement operators are now demonstrated. First, preincrement adds 1 to `age` and the result is output. Then, `age`'s current value is output and this variable is then decremented via postdecrement. What `age` values do you think are output?

The next two expression examples demonstrate string concatenation. First, "B" is concatenated to "A" and the resulting AB is output. Then, 32-bit integer value 5 is converted to a one-character string consisting of character 5, which is then concatenated to "A". The resulting A5 is output.

At this point, overflow is demonstrated. First, a 16-bit short integer variable named x is declared and initialized to the largest positive short integer: 32767. The preincrement operator is then applied to x and the result (-32768) is output.

Compile Listing 2-5, as follows:

```
javac CompoundExpressions.java
```

Assuming successful compilation, execute the following command to run this application:

```
java CompoundExpressions
```

You should observe the following output:

```
97
66
66
AB
A5
-32768
```

Array Index Operator

The array index operator ([]) accesses an array element by presenting the location of that element as an integer index. This operator is specified after an array variable's name, such as `ages[0]`.

Indexes are relative to 0, which implies that `ages[0]` accesses the first element, whereas `ages[6]` accesses the seventh element. The index must be greater than or equal to 0 and less than the length of the array; otherwise, the virtual machine throws `ArrayIndexOutOfBoundsException` (consult Chapter 5 to learn about exceptions).

An array's length is returned by appending ".length" to the array variable. For example, `ages.length` returns the length of (the number of elements in) the array that `ages` references. Similarly, `matrix.length` returns the number of row elements in the

`matrix` two-dimensional array, whereas `matrix[0].length` returns the number of column elements assigned to the first row element of this array. (A two-dimensional array is essentially a one-dimensional row array of one-dimensional column arrays.)

Listing 2-6 presents a `CompoundExpressions` application that lets you start experimenting with the array index operator.

Listing 2-6. Experimenting with the Array Index Operator

```
public class CompoundExpressions {
    public static void main(String[] args) {
        int[] ages = { 52, 28, 93, 16 };
        char gradeLetters[] = { 'A', 'B', 'C', 'D', 'F' };
        float[][] matrix = { { 1.0F, 2.0F, 3.0F }, { 4.0F, 5.0F, 6.0F } };
        System.out.println(ages[0]);
        System.out.println(gradeLetters[2]);
        System.out.println(matrix[1][2]);
        System.out.println(ages['\u0002']);
        ages[1] = 19;
        System.out.println(ages[1]);
    }
}
```

Listing 2-6's `main()` method first declares and assigns arrays to variables `ages`, `gradeLetters`, and `matrix`. It then uses the array index operator to access the first element in the `ages` array (`ages[0]`), the third element in the `gradeLetters` array (`gradeLetters[2]`), and the third column element in the second row element of the `matrix` table array (`matrix[1][2]`).

Array indexes must be integer values. These values can be of byte integer, short integer, or integer type. However, they cannot be of long integer type because that could result in a loss of precision. The maximum number of elements that can be stored in an array is a bit less than the largest positive 32-bit integer; a long integer can be much larger than this value.

`main()` next demonstrates that you can also specify a character as an index value (`ages['\u0002']`). This is legal because Java supports a character-to-integer widening rule; it converts a character value to an integer value, which is then used as an index into the array (`ages[2]`). However, you should avoid using characters as array indexes because they're not intuitive and are potentially error prone. For example, what element

is accessed by `ages['A']`? The answer is the 66th element; A's Unicode value is 65 and the first array index is 0. Given the previous four-element `ages` array, `ages['A']` would result in `ArrayIndexOutOfBoundsException`.

Finally, `main()` demonstrates that you can also use the array index operator to assign a value to an array element. In this case, integer 19 is stored in the second array element, which is subsequently accessed and output.

Compile Listing 2-6 (`javac CompoundExpressions.java`) and run this application (`java CompoundExpressions`). You should observe the following output:

```
52
C
6.0
93
19
```

Assignment Operators

The assignment operator (`=`) assigns an expression's result to a variable (as in `int x = 4;`). The types of the variable and expression must agree; otherwise, the compiler reports an error.

Java also supports several compound assignment operators that perform a specific operation and assign its result to a variable. For example, in `pennies += 50;`, the `+=` operator evaluates the numeric expression on its right (50) and adds the result to the contents of the variable on its left (`pennies`). The other compound assignment operators behave in a similar way.

Bitwise Operators

The bitwise operators consist of bitwise AND (`&`), bitwise complement (`~`), bitwise exclusive OR (`^`), and bitwise inclusive OR (`|`). These operators are designed to work on the binary representations of their character or integral operands. Because this concept can be hard to understand if you haven't previously worked with these operators in another language, check out Listing 2-7.

Listing 2-7. Experimenting with the Bitwise Operators

```
public class CompoundExpressions {
    public static void main(String[] args) {
        System.out.println(~181);
        System.out.println(26 & 183);
        System.out.println(26 ^ 183);
        System.out.println(26 | 183);
    }
}
```

Compile Listing 2-7 (`javac CompoundExpressions.java`) and run this application (`java CompoundExpressions`). You should observe the following output:

```
-182
18
173
191
```

To make sense of these values, it helps to examine their 32-bit binary representations:

- 181 corresponds to 00000000000000000000000010110101
- 26 corresponds to 00000000000000000000000011010
- 183 corresponds to 00000000000000000000000010110111

When you specify `~181`, you end up flipping all of the bits: `~00000000000000000000000010110101` results in `1111111111111111111111101001010`. According to two's complement representation, an integer whose leading bit is 1 is regarded to be negative, which is why `~181` equates to -182.

The expression `26 & 183` can be represented in binary as follows:

```
00000000000000000000000011010
&
00000000000000000000000010110111
-----
00000000000000000000000010010
```

The resulting binary value equates to 18.

The expression `26 ^ 183` can be represented in binary as follows:

```
0000000000000000000000000000000011010
```

^

```
0000000000000000000000000000000010110111
```

```
0000000000000000000000000000000010101101
```

The resulting binary value equates to 173.

The expression `26 | 183` can be represented in binary as follows:

```
0000000000000000000000000000000011010
```

|

```
0000000000000000000000000000000010110111
```

```
0000000000000000000000000000000010111111
```

The resulting binary value equates to 191.

Cast Operator

The cast operator—(*type*)—attempts to convert the type of its operand to *type*. This operator exists because the compiler will not allow you to convert a value from one type to another in which information will be lost without specifying your intention to do so (via the cast operator). For example, when presented with `short s = 1.65 + 3;`, the compiler reports an error because attempting to convert a 64-bit double-precision floating-point value to a 16-bit signed short integer results in the loss of the fraction .65, so *s* would contain 4 instead of 4.65.

Recognizing that information loss might not always be a problem, Java permits you to state your intention explicitly by casting to the target type. For example, `short s = (short) 1.65 + 3;` tells the compiler that you want `1.65 + 3` to be converted to a short integer, and that you realize that the fraction will disappear.

The following example provides another demonstration of the need for a cast operator:

```
char c = 'A';
byte b = c;
```

The compiler reports an error about loss of precision when it encounters `byte b = c;`. The reason is that `c` can represent any unsigned integer value from 0 through 65535, whereas `b` can only represent a signed integer value from -128 through +127. Even though '`A`' equates to +65, which can fit within `b`'s range, `c` could just have easily been initialized to '`\u0323`', which wouldn't fit.

The solution to this problem is to introduce a (byte) cast operator as follows, which causes the compiler to generate code to cast `c`'s character type to byte integer:

```
byte b = (byte) c;
```

Java supports the following primitive-type conversions via cast operators:

- Byte integer to character
- Short integer to byte integer or character
- Character to byte integer or short integer
- Integer to byte integer, short integer, or character
- Long integer to byte integer, short integer, character, or integer
- Floating-point to byte integer, short integer, character, integer, or long integer
- Double-precision floating-point to byte integer, short integer, character, integer, long integer, or floating-point

A cast operator isn't always required when converting from more to fewer bits and where no data loss occurs. For example, when it encounters `byte b = 100;`, the compiler generates code that assigns integer 100 to byte integer variable `b` because 100 can easily fit into the 8-bit storage location assigned to this variable.

[Listing 2-8](#) presents a `CompoundExpressions` application that lets you start experimenting with the cast operator.

Listing 2-8. Experimenting with the Cast Operator

```
public class CompoundExpressions {
    public static void main(String[] args) {
        short s = (short) 1.65 + 3;
        System.out.println(s);
```

```

char c = 'A';
byte b = (byte) c;
System.out.println(b);

b = 100;
System.out.println(b);

s = 'A';
System.out.println(s);

s = (short) '\u0041';
System.out.println(s);
}

}

```

Listing 2-8's `main()` method first uses the `(short)` cast operator to narrow the double-precision floating-point expression `1.65 + 3` to a 16-bit short integer that's ultimately assigned to short integer variable `s`. After outputting `s`'s value (4), this method demonstrates the mandatory `(byte)` cast operator when converting from a 16-bit unsigned character type to an 8-bit signed byte integer type.

As previously mentioned, the `(byte)` cast operator isn't always required. For example, when assigning a 32-bit signed integer in the range of -128 through +127 to a byte integer variable, `(byte)` can be omitted because no information will be lost. Assigning 100 to `b` demonstrates this scenario.

In a similar way, various 16-bit unsigned character values (such as `'A'`) can be assigned to a 16-bit signed short integer variable without loss of information, and so the `(short)` cast operator can be avoided. However, other 16-bit unsigned character values don't fit into this range and must be cast to a short integer before assignment (e.g., `'\u0041'`).

Compile Listing 2-8 (`javac CompoundExpressions.java`) and run this application (`java CompoundExpressions`). You should observe the following output:

```

4
65
100
65
-21504

```

Conditional Operators

The conditional operators consist of conditional AND (`&&`), conditional OR (`||`), and conditional (`? :`). The first two operators always evaluate their left operand (a Boolean expression that evaluates to true or false) and conditionally evaluate their right operand (another Boolean expression). The third operator evaluates one of two operands based on a third Boolean operand.

Conditional AND always evaluates its left operand and evaluates its right operand only when its left operand evaluates to true. For example, `age > 64 && stillWorking` first evaluates `age > 64`. If this subexpression is true, `stillWorking` is evaluated, and its true or false value (`stillWorking` is a Boolean variable) serves as the value of the overall expression. If `age > 64` is false, `stillWorking` isn't evaluated.

Conditional OR always evaluates its left operand and evaluates its right operand only when its left operand evaluates to false. For example, `value < 20 || value > 40` first evaluates `value < 20`. If this subexpression is false, `value > 40` is evaluated, and its true or false value serves as the overall expression's value. If `value < 20` is true, `value > 40` isn't evaluated.

Conditional AND and conditional OR boost performance by preventing the unnecessary evaluation of subexpressions, which is known as *short-circuiting*. For example, if its left operand is false, there is no way that conditional AND's right operand can change the fact that the overall expression will evaluate to false.

If you aren't careful, short-circuiting can prevent *side effects* (the results of subexpressions that persist after the subexpressions have been evaluated) from executing. For example, `age > 64 && ++numEmployees > 5` increments `numEmployees` for only those employees whose ages are greater than 64. Incrementing `numEmployees` is an example of a side effect because the value in `numEmployees` persists after the subexpression `++numEmployees > 5` has evaluated.

The conditional operator is useful for making a decision by evaluating and returning one of two operands based upon the value of a third operand. The following example converts a Boolean value to its integer equivalent (1 for true and 0 for false):

```
boolean b = true;
int i = b ? 1 : 0; // 1 assigns to i
```

[Listing 2-9](#) presents a `CompoundExpressions` application that lets you start experimenting with the conditional operators.

Listing 2-9. Experimenting with the Conditional Operators

```
public class CompoundExpressions {  
    public static void main(String[] args) {  
        int age = 65;  
        boolean stillWorking = true;  
        System.out.println(age > 64 && stillWorking);  
        age--;  
        System.out.println(age > 64 && stillWorking);  
        int value = 30;  
        System.out.println(value < 20 || value > 40);  
        value = 10;  
        System.out.println(value < 20 || value > 40);  
        int numEmployees = 6;  
        age = 65;  
        System.out.println(age > 64 && ++numEmployees > 5);  
        System.out.println("numEmployees = " + numEmployees);  
        age = 63;  
        System.out.println(age > 64 && ++numEmployees > 5);  
        System.out.println("numEmployees = " + numEmployees);  
        boolean b = true;  
        int i = b ? 1 : 0; // 1 assigns to i  
        System.out.println("i = " + i);  
        b = false;  
        i = b ? 1 : 0; // 0 assigns to i  
        System.out.println("i = " + i);  
    }  
}
```

Compile Listing 2-9 () and run this application (). You should observe the following output:

```
true  
false  
false  
true
```

```
true  
numEmployees = 7  
false  
numEmployees = 7  
i = 1  
i = 0
```

Equality Operators

The equality operators consist of equality (`==`) and inequality (`!=`). These operators compare their operands to determine whether they are equal or unequal. The former operator returns true when equal and the latter operator returns true when unequal.

For example, each of `2 == 2` and `2 != 3` evaluates to true, whereas each of `2 == 4` and `4 != 4` evaluates to false.

You have to be careful when comparing floating-point expressions for equality. For example, what does `System.out.println(0.3 == 0.1 + 0.1 + 0.1);` output? If you guessed that the output is true, you would be wrong. Instead, the output is false.

The reason for this nonintuitive output is that 0.1 cannot be represented exactly in memory. The error compounds when this value is added to itself. For example, if you executed `System.out.println(0.1 + 0.1 + 0.1);`, you would observe `0.3000000000000004`, which doesn't equal 0.3.

When it comes to object operands (we discuss objects in Chapter 3), these operators don't compare their contents. Instead, object references are compared. For example, `"abc" == "xyz"` doesn't compare a with x. Because string literals are really `String` objects (Chapter 7 discusses the `String` class), `==` compares the references to these objects.

Logical Operators

The logical operators consist of logical AND (`&`), logical complement (`!`), logical exclusive OR (`^`), and logical inclusive OR (`|`). Although these operators are similar to their bitwise counterparts, whose operands must be integer/character, the operands passed to the logical operators must be Boolean. For example, `!false` returns true. Also, when confronted with `age > 64 & stillWorking`, logical AND evaluates both subexpressions; there's no short-circuiting. This same pattern holds for logical exclusive OR and logical inclusive OR.

Listing 2-10 presents a `CompoundExpressions` application that lets you start experimenting with the logical operators.

Listing 2-10. Experimenting with the Logical Operators

```
public class CompoundExpressions {  
    public static void main(String[] args) {  
        System.out.println(!false);  
        int age = 65;  
        boolean stillWorking = true;  
        System.out.println(age > 64 & stillWorking);  
        System.out.println();  
  
        boolean result = true & true;  
        System.out.println("true & true: " + result);  
        result = true & false;  
        System.out.println("true & false: " + result);  
        result = false & true;  
        System.out.println("false & true: " + result);  
        result = false & false;  
        System.out.println("false & false: " + result);  
        System.out.println();  
  
        result = true | true;  
        System.out.println("true | true: " + result);  
        result = true | false;  
        System.out.println("true | false: " + result);  
        result = false | true;  
        System.out.println("false | true: " + result);  
        result = false | false;  
        System.out.println("false | false: " + result);  
        System.out.println();  
  
        result = true ^ true;  
        System.out.println("true ^ true: " + result);  
        result = true ^ false;
```

```
System.out.println("true ^ false: " + result);
result = false ^ true;
System.out.println("false ^ true: " + result);
result = false ^ false;
System.out.println("false ^ false: " + result);
System.out.println();
int numEmployees = 1;
age = 65;
System.out.println(age > 64 & ++numEmployees > 2);
System.out.println(numEmployees);
}
}
```

After outputting the results of the `!false` and `age > 64 & stillWorking` expressions, `main()` outputs three truth tables that show how logical AND, logical inclusive OR, and logical exclusive OR behave when their operands are true or false. It then demonstrates that short-circuiting is ignored by incrementing `numEmployees` when `age > 64` returns true.

Compile Listing 2-10 (`javac CompoundExpressions.java`) and run this application (`java CompoundExpressions`). You should observe the following output:

```
true
true

true & true: true
true & false: false
false & true: false
false & false: false

true | true: true
true | false: true
false | true: true
false | false: false

true ^ true: false
true ^ false: true
```

```
false ^ true: true
false ^ false: false

false
2
```

Member Access Operator

The member access operator (`.`) is used to access a class's members or an object's members. For example, `String s = "Hello"; int len = s.length();` returns the length of the string assigned to variable `s`. It does so by calling the `length()` method member of the `String` class. In Chapter 3, we discuss member access in more detail.

Arrays are special objects that have a single `length` member. When you specify an array variable followed by the member access operator, followed by `length`, the resulting expression returns the number of elements in the array as a 32-bit integer. For example, `ages.length` returns the length of (the number of elements in) the array that `ages` references.

Method Call Operator

The method call operator `()` is used to signify that a method (discussed in Chapter 3) is being called. Also, it identifies the number, order, and types of arguments that are passed to the method to be picked up by the method's parameters. For example, in the `System.out.println("Hello");` method call, `()` signifies that a method named `println` is being called with one argument: `"Hello"`.

Multiplicative Operators

The multiplicative operators consist of multiplication `(*)`, division `(/)`, and remainder `(%)`. Multiplication returns the product of its operands (such as `6 * 4` returns 24), division returns the quotient of dividing its left operand by its right operand (such as `6 / 4` returns 1), and remainder returns the remainder of dividing its left operand by its right operand (such as `6 % 4` returns 2).

The multiplication, division, and remainder operators can yield values that overflow or underflow the limits of the resulting value's type. For example, multiplying two large positive 32-bit integer values can produce a value that cannot be represented as

a 32-bit integer value. The result is said to overflow. Java doesn't detect overflows and underflows.

Dividing a numeric value by 0 (via the division or remainder operator) also results in interesting behavior. Dividing an integer value by integer 0 causes the operator to throw an `ArithmaticException` object (Chapter 5 covers exceptions). Dividing a floating-point/double-precision floating-point value by 0 causes the operator to return `+infinity` or `-infinity`, depending on whether the dividend is positive or negative. Finally, dividing floating-point 0 by 0 causes the operator to return `NaN` (Not a Number).

[Listing 2-11](#) presents a `CompoundExpressions` application that lets you start experimenting with the multiplicative operators.

Listing 2-11. Experimenting with the Multiplicative Operators

```
public class CompoundExpressions {
    public static void main(String[] args) {
        short age = 65;
        System.out.println(age * 1000);
        System.out.println(1.0 / 0.0);
        System.out.println(10 % 4);
        System.out.println(3 / 0);
    }
}
```

Compile Listing 2-11 (`javac CompoundExpressions.java`) and run this application (`java CompoundExpressions`). You should observe the following output:

```
65000
Infinity
2
Exception in thread "main" java.lang.ArithmaticException: / by zero
    at CompoundExpressions.main(CompoundExpressions.java:9)
```

Object Creation Operator

The object creation operator (`new`) creates an object from a class, and it also creates an array from an initializer. These topics are discussed in Chapter 3.

Relational Operators

The relational operators consist of greater than ($>$), greater than or equal to (\geq), less than ($<$), less than or equal to (\leq), and type checking (`instanceof`). The former four operators compare their operands and return true when the left operand is (respectively) greater than, greater than or equal to, less than, or less than or equal to the right operand. For example, each of `5.0 > 3`, `2 >= 2`, `16.1 < 303.3`, and `54.0 <= 54.0` evaluates to true.

The type-checking operator is used to determine if an object belongs to a specific type, returning true when this is the case. For example, `"abc" instanceof String` returns true because `"abc"` is a `String` object. We discuss this operator more fully in Chapter 4.

Shift Operators

The shift operators consist of left shift (`<<`), signed right shift (`>>`), and unsigned right shift (`>>>`). Left shift shifts the binary representation of its left operand leftward by the number of positions specified by its right operand. Each shift is equivalent to multiplying by 2. For example, `2 << 3` shifts 2's binary representation left by three positions; the result is equivalent to multiplying 2 by 8.

Each of signed and unsigned right shift shifts the binary representation of its left operand rightward by the number of positions specified by its right operand. Each shift is equivalent to dividing by 2. For example, `16 >> 3` shifts 16's binary representation right by three positions; the result is equivalent to dividing 16 by 8.

The difference between signed and unsigned right shift is what happens to the sign bit during the shift. Signed right shift includes the sign bit in the shift, whereas unsigned right shift ignores the sign bit. As a result, signed right shift preserves negative numbers, but unsigned right shift doesn't. For example, `-4 >> 1` (the equivalent of `-4 / 2`) evaluates to -2, whereas `-4 >>> 1` evaluates to 2147483646.

Listing 2-12 presents a `CompoundExpressions` application that lets you start experimenting with the shift operators.

Listing 2-12. Experimenting with the Shift Operators

```
public class CompoundExpressions {
    public static void main(String[] args) {
        System.out.println(2 << 3);
        System.out.println(16 >> 3);
```

```
        System.out.println(-4 >> 1);
        System.out.println(-4 >>> 1);
    }
}
```

Compile Listing 2-12 (`javac CompoundExpressions.java`) and run this application (`java CompoundExpressions`). You should observe the following output:

```
16
2
-2
2147483646
```

Tip The shift operators are faster than multiplying or dividing by powers of 2.

Unary Minus/Plus Operators

Unary minus (-) and unary plus (+) are the simplest of all operators. Unary minus returns the negative of its operand (such as `-5` returns `-5` and `--5` returns `5`), whereas unary plus returns its operand verbatim (such as `+5` returns `5` and `+-5` returns `-5`). Unary plus is not commonly used, but it is presented for completeness.

Precedence and Associativity

When evaluating a compound expression, Java takes each operator's *precedence* (level of importance) into account to ensure that the expression evaluates as expected. For example, when presented with the expression `60 + 3 * 6`, you expect multiplication to be performed before addition (multiplication has higher precedence than addition) and the final result to be `78`. You don't expect addition to occur first, yielding a result of `378`.

Note Table 2-3's rightmost column presents a value that indicates an operator's precedence: the higher the number, the higher the precedence. For example, addition's precedence level is 10 and multiplication's precedence level is 11, which means that multiplication is performed before addition.

Precedence can be circumvented by introducing open and close parentheses, (and), into the expression, where the innermost pair of nested parentheses is evaluated first. For example, evaluating $2 * ((60 + 3) * 6)$ results in $(60 + 3)$ being evaluated first, $(60 + 3) * 6$ being evaluated next, and the overall expression being evaluated last. Similarly, in the expression $60 / (3 - 6)$, subtraction is performed before division.

During evaluation, operators with the same precedence level (such as addition and subtraction, which both have level 10) are processed according to their *associativity* (a property that determines how operators having the same precedence are grouped when parentheses are missing).

For example, expression $9 * 4 / 3$ is evaluated as if it was $(9 * 4) / 3$ because * and / are left-to-right associative operators. In contrast, expression $x = y = z = 100$ is evaluated as if it was $x = (y = (z = 100))$, where 100 is assigned to z, z's new value (100) is assigned to y, and y's new value (100) is assigned to x because = is a right-to-left associative operator.

Most of Java's operators are left-to-right associative. Right-to-left associative operators include assignment, bitwise complement, cast, compound assignment, conditional, logical complement, object creation, predecrement, preincrement, unary minus, and unary plus.

Listing 2-13 presents a CompoundExpressions application that lets you start experimenting with precedence and associativity.

Listing 2-13. Experimenting with Precedence and Associativity

```
public class CompoundExpressions {
    public static void main(String[] args) {
        System.out.println(60 + 3 * 6);
        System.out.println(2 * ((60 + 3) * 6));
        System.out.println(9 * 4 / 3);

        int x, y, z;
        x = y = z = 100;
        System.out.println(x);
        System.out.println(y);
        System.out.println(z);

        int i = 0x12345678;
        byte b = (byte) (i & 255);
        System.out.println(b);
    }
}
```

```

System.out.println("b == 0x78: " + (b == 0x78));
b = (byte) ((i >> 8) & 255);
System.out.println(b);
System.out.println("b == 0x56: " + (b == 0x56));
b = (byte) ((i >> 16) & 255);
System.out.println(b);
System.out.println("b == 0x34: " + (b == 0x34));
b = (byte) ((i >> 24) & 255);
System.out.println(b);
System.out.println("b == 0x12: " + (b == 0x12));
}
}

```

You'll often find yourself needing to use open and close parentheses to change an expression's evaluation order. For example, consider the second part of the `main()` method, which extracts each of the 4 bytes in the 32-bit value assigned to integer variable `i` and outputs this byte.

After processing the declaration and initialization of 32-bit integer variable `i` (`int i = 0x12345678;`), the compiler encounters `byte b = (byte) (i & 255);`. It generates bytecode that first evaluates expression `i & 255`, which returns a 32-bit result, passes this result to the `(byte)` cast operator to convert it to an 8-bit result, and assigns the 8-bit result to 8-bit byte integer variable `b`.

Suppose the parentheses were absent, resulting in `byte b = (byte) i & 255;`. The compiler would then report an error about loss of precision because it interprets this expression as follows:

1. Cast variable `i` to an 8-bit byte integer. The cast operator is a unary operator that takes only one operand. Also, cast has higher precedence (12) than bitwise AND (6) so cast is evaluated first.
2. Widen `i` to a 32-bit integer as the left operand of bitwise AND (&). Operand 255 is already a 32-bit integer.
3. Apply bitwise AND to these operands. The result is a 32-bit integer.
4. Attempt to assign the 32-bit integer result to 8-bit byte integer variable `b`.

The 32-bit integer result can vary from 0 through 255. However, the largest positive integer that `b` can store is 127. If the result ranges from 128 through 255, it will be converted to -1 through -128. This is a loss of precision and so the compiler reports an error.

Another example where `main()` uses open and close parentheses to change evaluation order is `System.out.println("b == 0x78: " + (b == 0x78));`. When the parentheses are missing, the compiler reports an “incomparable types: String and int” error. It does so because string concatenation has higher precedence (10) than equality (7). As a result, the compiler interprets the expression (without parentheses) as follows:

1. Convert `b`’s value to a string.
2. Concatenate this string to “`b == 0x78:` ”.
3. Compare the resulting string with 32-bit integer `0x78` for equality, which is illegal.

Compile Listing 2-13 (`javac CompoundExpressions.java`) and run this application (`java CompoundExpressions`). You should observe the following output:

```
78
756
12
100
100
100
120
b == 0x78: true
86
b == 0x56: true
52
b == 0x34: true
18
b == 0x12: true
```

Note Unlike languages such as C++, Java doesn't let you overload operators.

Learning Statements

Statements are the workhorses of a program. They assign values to variables, control a program's flow by making decisions and/or repeatedly executing other statements, and perform other tasks. A statement can be expressed as a simple statement or as a compound statement.

- A *simple statement* is a single stand-alone source code instruction for performing some task; it's terminated with a semicolon.
- A *compound statement* is a (possibly empty) sequence of simple and other compound statements sandwiched between open and close brace delimiters; a *delimiter* is a character that marks the beginning or end of some section. A method body (such as the `main()` method's body) is an example. Compound statements can appear wherever simple statements appear and are alternatively referred to as *blocks*.

In this section we introduce you to many of Java's statements. Additional statements are covered in later chapters. For example, in Chapter 3 we discuss the return statement.

Assignment Statements

The *assignment statement* assigns a value to a variable. This statement begins with a variable name, continues with the assignment operator (=) or a compound assignment operator (such as +=), and concludes with an assignment-compatible expression and a semicolon. The following are three examples:

```
x = 10;  
ages[0] = 25;  
counter += 10;
```

The first example assigns integer 10 to variable `x`, which is presumably of type `integer` as well. The second example assigns integer 25 to the first element of the `ages` array. The third example adds 10 to the value stored in `counter` and stores the sum in `counter`.

Note Initializing a variable in the variable's declaration (such as `int counter = 1;`) can be thought of as a special form of the assignment statement.

Decision Statements

The previously described conditional operator (`? :`) is useful for choosing between two expressions to evaluate and cannot be used to choose between two statements. For this purpose, Java supplies three decision statements: `if`, `if-else`, and `switch`.

If Statement

The *if statement* evaluates a Boolean expression and executes another statement if this expression evaluates to true. It has the following syntax:

```
if (Boolean expression)
    statement
```

This statement consists of reserved word `if`, followed by a *Boolean expression* in parentheses, followed by a *statement* to execute when *Boolean expression* evaluates to true.

The following example demonstrates the `if` statement:

```
if (numMonthlySales > 100)
    wage += bonus;
```

If the number of monthly sales exceeds 100, `numMonthlySales > 100` evaluates to true and the `wage += bonus;` assignment statement executes. Otherwise, this assignment statement doesn't execute.

Note Many people prefer to wrap a single statement in brace characters in order to prevent the possibility of error. As a result, they would typically write the previous example as follows:

```
if (numMonthlySales > 100) {  
    wage += bonus;  
}
```

We don't do this for single statements because we view the extra braces as unnecessary clutter. However, you might feel differently. Use whatever approach makes you the most comfortable.

If-Else Statement

The *if-else statement* evaluates a Boolean expression and executes one of two statements depending on whether this expression evaluates to true or false. It has the following syntax:

```
if (Boolean expression)  
    statement1  
else  
    statement2
```

This statement consists of reserved word `if`, followed by a *Boolean expression* in parentheses, followed by a *statement1* to execute when *Boolean expression* evaluates to true, followed by a *statement2* to execute when *Boolean expression* evaluates to false.

The following example demonstrates the if-else statement:

```
if ((n & 1) == 1)  
    System.out.println("odd");  
else  
    System.out.println("even");
```

This example assumes the existence of an `int` variable named `n` that's been initialized to an integer. It then proceeds to determine if the integer is odd (not divisible by 2) or even (divisible by 2).

The Boolean expression first evaluates `n & 1`, which bitwise ANDs `n`'s value with 1. It then compares the result to 1. If they're equal, a message stating that `n`'s value is odd outputs; otherwise, a message stating that `n`'s value is even outputs.

The parentheses are required because `==` has higher precedence than `&`. Without these parentheses, the expression's evaluation order would change to first evaluating `1 == 1` and then trying to bitwise AND the Boolean result with `n`'s integer value. This order results in a compiler error message because of a type mismatch: you cannot bitwise AND an integer with a Boolean value.

You could rewrite this if-else statement example to use the conditional operator, as follows:

```
System.out.println((n & 1) == 1 ? "odd" : "even");
```

However, you cannot do so with the following example:

```
if ((n & 1) == 1)
    odd();
else
    even();
```

This example assumes the existence of `odd()` and `even()` methods that don't return anything. Because the conditional operator requires that each of its second and third operands evaluates to a value, the compiler reports an error when attempting to compile `(n & 1) == 1 ? odd() : even()`.

You can chain multiple if-else statements together, resulting in the following syntax:

```
if (Boolean expression1)
    statement1
else if (Boolean expression2)
    statement2
else if (...)

...
else
    statementN
```

If `Boolean expression1` evaluates to true, `statement1` executes. Otherwise, if `Boolean expression2` evaluates to true, `statement2` executes. This pattern continues until one of

these expressions evaluates to true and its corresponding statement executes, or the final `else` is reached and `statementN` (the default statement) executes.

Listing 2-14 presents a `GradeLetters` application that demonstrates chaining together multiple if-else statements.

Listing 2-14. Experimenting with If-Else Chaining

```
public class GradeLetters {
    public static void main(String[] args) {
        int testMark = 69;
        char gradeLetter;

        if (testMark >= 90) {
            gradeLetter = 'A';
            System.out.println("You aced the test.");
        } else if (testMark >= 80) {
            gradeLetter = 'B';
            System.out.println("You did very well on this test.");
        } else if (testMark >= 70) {
            gradeLetter = 'C';
            System.out.println("You'll need to study more for future tests.");
        } else if (testMark >= 60) {
            gradeLetter = 'D';
            System.out.println("Your test result suggests that you need a
tutor.");
        } else {
            gradeLetter = 'F';
            System.out.println("You fail and need to attend summer school.");
        }

        System.out.println("Your grade is " + gradeLetter + ".");
    }
}
```

Compile Listing 2-14 as follows:

```
javac GradeLetters.java
```

Execute the resulting application as follows:

```
java GradeLetters
```

You should observe the following output:

```
Your test result suggests that you need a tutor.  
Your grade is D.
```

DANGLING-ELSE PROBLEM

When if and if-else are used together and the source code isn't properly indented, it can be difficult to determine which if associates with the else. See the following, for example:

```
if (car.door.isOpen())  
    if (car.key.isPresent())  
        car.start();  
else car.door.open();
```

Did the developer intend for the else to match the inner if, but improperly formatted the code to make it appear otherwise? This reformatted possibility appears as follows:

```
if (car.door.isOpen())  
    if (car.key.isPresent())  
        car.start();  
    else  
        car.door.open();
```

If `car.door.isOpen()` and `car.key.isPresent()` each return true, `car.start()` executes. If `car.door.isOpen()` returns true and `car.key.isPresent()` returns false, `car.door.open();` executes. Attempting to open an open door makes no sense.

The developer must have wanted the else to match the outer if but forgot that else matches the nearest if. This problem can be fixed by surrounding the inner if with braces, as follows:

```
if (car.door.isOpen()) {  
    if (car.key.isPresent())
```

```

        car.start();
} else
    car.door.open();

```

When `car.door.isOpen()` returns true, the compound statement executes. When this method returns false, `car.door.open();` executes, which makes sense.

Forgetting that `else` matches the nearest `if` and using poor indentation to obscure this fact is known as the *dangling-else problem*.

Switch Statement

The *switch statement* lets you choose from among several execution paths in a more efficient manner than with equivalent chained if-else statements. It has the following syntax:

```

switch (selector expression) {
    case value1: statement1 [;break;]
    case value2: statement2 [;break;]
    ...
    case valueN: statementN [;break;]
    [default: statement]
}

```

This statement consists of reserved word `switch`, followed by a *selector expression* in parentheses, followed by a body of cases. The *selector expression* is any expression that evaluates to an integer, character, or string value.

Each case begins with reserved word `case`, continues with a literal value and a colon character (`:`), continues with a statement to execute, and optionally concludes with a `break` statement, which causes execution to continue after the `switch` statement.

After evaluating the *selector expression*, `switch` compares this value with each case's value until it finds a match. When there is a match, the case's statement is executed. For example, when the *selector expression*'s value matches `value1`, `statement1` executes.

The optional `break` statement (anything placed in square brackets is optional), which consists of reserved word `break` followed by a semicolon, prevents the flow of execution from continuing with the next case's statement. Instead, execution continues with the first statement following `switch`.

Note You'll usually place a break statement after a case's statement. Forgetting to include break can lead to a hard-to-find bug. However, there are situations where you want to group several cases together and have them execute common code. In this situation, you would omit the break statement from the participating cases.

If none of the cases' values match the *selector expression*'s value, and if a default case (signified by the default reserved word followed by a colon) is present, the default case's statement is executed.

The following example demonstrates this statement:

```
switch (direction) {  
    case 0: System.out.println("You are travelling north."); break;  
    case 1: System.out.println("You are travelling east."); break;  
    case 2: System.out.println("You are travelling south."); break;  
    case 3: System.out.println("You are travelling west."); break;  
    default: System.out.println("You are lost.");  
}
```

This example assumes that direction stores an integer value. When this value is in the range 0–3, an appropriate direction message is output; otherwise, a message about being lost is output.

Note This example hard-codes values 0, 1, 2, and 3, which isn't always a good idea in practice. Instead, constants should be used. Chapter 3 introduces you to constants.

Loop Statements

It's often necessary to repeatedly execute a statement; this repeated execution is called a *loop*. Java provides three kinds of loop statements: for, while, and do-while. In this section, we first discuss these statements. We then examine the topic of looping over the empty statement. Finally, we discuss the break, labeled break, continue, and labeled continue statements for prematurely ending all or part of a loop.

For Statement

The *for statement* lets you loop over a statement a specific number of times or even indefinitely. It has the following syntax:

```
for ([initialize]; [test]; [update])
    statement
```

This statement consists of reserved word `for`, followed by a header in parentheses, followed by a *statement* to execute. The header consists of an optional *initialize* section, followed by an optional *test* section, followed by an optional *update* section.

A nonoptional semicolon separates each of the first two sections from the next section.

The *initialize* section consists of a comma-separated list of variable declarations or variable assignments. Some or all of these variables are typically used to control the loop's duration and are known as *loop-control variables*.

The *test* section consists of a Boolean expression that determines how long the loop executes. Execution continues as long as this expression evaluates to true.

Finally, the *update* section consists of a comma-separated list of expressions that typically modify the loop-control variables.

The `for` statement is perfect for *iterating* (looping) over an array. Each *iteration* (loop execution) accesses one of the array's elements via an `array[index]` expression, where `array` is the array whose element is being accessed and `index` is the zero-based location of the element being accessed.

The following example uses `for` to iterate over the array of command-line arguments passed to `main()`:

```
public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
        System.out.println(args[i]);
}
```

The initialization section declares variable `i` for controlling the loop, the test section compares `i`'s current value to the length of the `args` array to ensure that this value is less than the array's length, and the update section increments `i` by 1. The `for`-based loop continues until `i`'s value equals the array's length.

Each array element is accessed via the `args[i]` expression, which returns this array's `i`th element's value (which happens to be a `String` object in this example). The first value is stored in `args[0]`.

Note Although we've named the array containing command-line arguments `args`, this name isn't mandatory. We could as easily have named it `arguments` (or even `some_other_name`).

Listing 2-15 presents a `DumpMatrix` application that uses a `for`-based loop to output the contents of a two-dimensional matrix array.

Listing 2-15. Iterating over a Two-Dimensional Array's Rows and Columns

```
public class DumpMatrix {  
    public static void main(String[] args) {  
        float[][] matrix = { { 1.0F, 2.0F, 3.0F }, { 4.0F, 5.0F, 6.0F } };  
        for (int row = 0; row < matrix.length; row++) {  
            for (int col = 0; col < matrix[row].length; col++)  
                System.out.print(matrix[row][col] + " ");  
            System.out.print("\n");  
        }  
    }  
}
```

Expression `matrix.length` returns the number of rows in this tabular array. For each row, expression `matrix[row].length` returns the number of columns for that row. This latter expression suggests that each row can have a different number of columns, although each row has the same number of columns in the example.

`System.out.print()` is closely related to `System.out.println()`. Unlike the latter method, `System.out.print()` outputs its argument without a trailing newline.

Compile Listing 2-15 as follows:

```
javac DumpMatrix.java
```

Execute the resulting application as follows:

```
java DumpMatrix
```

You should observe the following output:

```
1.0 2.0 3.0  
4.0 5.0 6.0
```

A different form of the for loop with improved readability lets you iterate through list and array elements.

Listing 2-16. Iterating over Lists and Arrays, Alternative

```
import java.util.*;
public class DumpMatrix {
    public static void main(String[] args) {
        int[] a = {1, 2, 3, 7};
        for (int i : a) {
            System.out.println(i);
        }

        List<String> l = Arrays.asList("Hello", "there");
        for (String s : l) {
            System.out.println(s);
        }
    }
}
```

Here we use `import` to import built-in library classes, namely, `Arrays` and `List` (we'll talk about imports later).

While Statement

The *while statement* repeatedly executes another statement while its Boolean expression evaluates to true. It has the following syntax:

```
while (Boolean expression)
    statement
```

This statement consists of reserved word `while`, followed by a parenthesized *Boolean expression*, followed by a *statement* to execute repeatedly. The `while` statement first evaluates the *Boolean expression*. If it's true, `while` executes the other *statement*. Once again, the *Boolean expression* is evaluated. If it's still true, `while` reexecutes the *statement*. This cyclic pattern continues.

Prompting the user to enter a specific character is one situation where while is useful. For example, suppose that you want to prompt the user to enter a specific uppercase letter or its lowercase equivalent. The following example provides a demonstration:

```
int ch = 0;
while (ch != 'C' && ch != 'c') {
    System.out.println("Press C or c to continue.");
    ch = System.in.read();
}
```

This example first initializes variable `ch`. This variable must be initialized; otherwise, the compiler will report an uninitialized variable when it tries to read `ch`'s value in the while statement's Boolean expression.

This expression uses the conditional AND operator (`&&`) to test `ch`'s value. This operator first evaluates its left operand, which happens to be expression `ch != 'C'`. (The `!=` operator converts '`C`' from 16-bit unsigned `char` type to 32-bit signed `int` type before the comparison.)

If `ch` doesn't contain `C` (it doesn't at this point; 0 was just assigned to `ch`), this expression evaluates to true.

The `&&` operator next evaluates its right operand, which happens to be expression `ch != 'c'`. Because this expression also evaluates to true, conditional AND returns true and while executes the compound statement.

The compound statement first outputs, via the `System.out.println()` method call, a message that prompts the user to press the `C` key with or without the Shift key. It next reads the entered keystroke via `System.in.read()`, saving its integer value in `ch`.

From left to write, `System` identifies a standard class of system utilities, `in` identifies an object located in `System` that provides methods for inputting one or more bytes from the standard input device, and `read()` returns the next byte (or -1 when there are no more bytes).

After this assignment, the compound statement ends and while reevaluates its Boolean expression.

Suppose `ch` contains `C`'s integer value. Conditional AND evaluates `ch != 'C'`, which evaluates to false. Detecting that the expression is already false, conditional AND short-circuits its evaluation by not evaluating its right operand and returns false. The while statement subsequently detects this value and terminates.

Suppose `ch` contains `c`'s integer value. Conditional AND evaluates `ch != 'C'`, which evaluates to true. Detecting that the expression is true, conditional AND evaluates `ch != 'c'`, which evaluates to false. Once again, the while statement terminates.

Note A for statement can be coded as a while statement. For example,

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

is equivalent to

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

Do-While Statement

The *do-while statement* repeatedly executes a statement while its Boolean expression evaluates to true. Unlike while, which evaluates the Boolean expression at the top of the loop, do-while evaluates the Boolean expression at the bottom of the loop. It has the following syntax:

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

This statement consists of the `do` reserved word, followed by a *statement* to execute repeatedly, followed by the `while` reserved word, followed by a parenthesized *Boolean expression*, followed by a semicolon.

The do-while statement first executes the other *statement*. It then evaluates the *Boolean expression*. If it's true, do-while executes the other *statement*. Once again, the *Boolean expression* is evaluated. If it's still true, do-while reexecutes the *statement*. This cyclic pattern continues.

The following example demonstrates do-while prompting the user to enter a specific uppercase letter or its lowercase equivalent:

```
int ch;
do {
    System.out.println("Press C or c to continue.");
    ch = System.in.read();
} while (ch != 'C' && ch != 'c');
```

This example is similar to its predecessor. Because the compound statement is no longer executed before the test, it's no longer necessary to initialize `ch`—`ch` is assigned `System.in.read()`'s return value before the Boolean expression's evaluation.

Looping over the Empty Statement

Java refers to a semicolon character appearing by itself as the *empty statement*. It's sometimes convenient for a loop statement to execute the empty statement repeatedly. The actual work performed by the loop statement takes place in the statement header.

Consider the following example:

```
for (String line; (line = readLine()) != null; System.out.println(line));
```

This example uses for to present a programming idiom for copying lines of text that are read from some source, via the fictitious `readLine()` method in this example, to some destination, via `System.out.println()` in this example. Copying continues until `readLine()` returns null. Note the semicolon (empty statement) at the end of the line.

Caution Be careful with the empty statement because it can introduce subtle bugs into your code. For example, the following loop is supposed to output the string Hello on ten lines. Instead, only one instance of this string is output, because it's the empty statement and not `System.out.println()` that's executed ten times:

```
for (int i = 0; i < 10; i++); // this ; represents the empty statement
    System.out.println("Hello");
```

Break Statements

What do for (;;); while (true); and do;while (true); have in common? Each of these loop statements presents an extreme example of an *infinite loop* (a loop that never ends). An infinite loop is something that you should avoid because its unending execution causes your application to hang, which isn't desirable from the point of view of your application's users.

Caution An infinite loop can also arise from a loop's Boolean expression comparing a floating-point value with a nonzero value via the equality or inequality operator, because many floating-point values have inexact internal representations. For example, the following example never ends because 0.1 doesn't have an exact internal representation:

```
for (double d = 0.0; d != 1.0; d += 0.1)
    System.out.println(d);
```

However, there are times when it's handy to code a loop as if it were infinite by using one of the aforementioned programming idioms. For example, you might code a while (true) loop that repeatedly prompts for a specific keystroke until the correct key is pressed. When the correct key is pressed, the loop must end. Java provides the break statement for this purpose.

The *break statement* transfers execution to the first statement following a switch statement (as discussed earlier) or a loop. In either scenario, this statement consists of reserved word break followed by a semicolon.

The following example uses break with an if decision statement to exit a while (true)-based infinite loop when the user presses the C or c key:

```
int ch;
while (true) {
    System.out.println("Press C or c to continue.");
    ch = System.in.read();
    if (ch == 'C' || ch == 'c')
        break;
}
```

The break statement is also useful in the context of a finite loop. For example, consider a scenario where an array of values is searched for a specific value, and you want to exit the loop when this value is found. Listing 2-17 presents an EmployeeSearch application that demonstrates this scenario.

Listing 2-17. Searching for a Specific Employee ID

```
public class EmployeeSearch {
    public static void main(String[] args) {
        int[] employeeIDs = { 123, 854, 567, 912, 224 };
        int employeeSearchID = 912;
        boolean found = false;
        for (int i = 0; i < employeeIDs.length; i++) {
            if (employeeSearchID == employeeIDs[i]) {
                found = true;
                break;
            }
        }
        System.out.println((found) ? "employee " + employeeSearchID + " exists"
            : "no employee ID matches " + employeeSearchID);
    }
}
```

Listing 2-17 uses for and if statements to search an array of employee IDs to determine if a specific employee ID exists. If this ID is found, its compound statement assigns true to found. Because there's no point in continuing the search, it then uses break to quit the loop.

Compile Listing 2-17 as follows:

```
javac EmployeeSearch.java
```

Run this application as follows:

```
java EmployeeSearch
```

You should observe the following output:

```
employee 912 exists
```

Continue Statements

The *continue statement* skips the remainder of the current loop iteration, reevaluates the loop's Boolean expression, and performs another iteration (if true) or terminates the loop (if false). Continue consists of reserved word `continue` followed by a semicolon.

Consider a while loop that reads lines from a source and processes nonblank lines in some manner. Because it shouldn't process blank lines, while skips the current iteration when a blank line is detected, as demonstrated in the following example:

```
String line;
while ((line = readLine()) != null) {
    if (isBlank(line))
        continue;
    processLine(line);
}
```

This example employs a fictitious `isBlank()` method to determine if the currently read line is blank. If this method returns true, if executes the `continue` statement to skip the rest of the current iteration and read the next line whenever a blank line is detected. Otherwise, the fictitious `processLine()` method is called to process the line's contents.

Look carefully at this example, and you should realize that the `continue` statement isn't needed. Instead, this listing can be shortened via *refactoring* (rewriting source code to improve its readability, organization, or reusability), as demonstrated in the following example:

```
String line;
while ((line = readLine()) != null) {
    if (!isBlank(line))
        processLine(line);
}
```

This example's refactoring modifies if's Boolean expression to use the logical complement operator (`!`). Whenever `isBlank()` returns false, this operator flips this value to true and if executes `processLine()`. Although `continue` isn't necessary in this example, you'll find it convenient to use this statement in more complex code where refactoring isn't as easy to perform.

EXERCISES

The following exercises are designed to test your understanding of Chapter 2's content:

1. What is Unicode?
2. What is a comment?
3. Identify the three kinds of comments that Java supports.
4. What is an identifier?
5. True or false: Java is a case-insensitive language.
6. What is a type?
7. Define primitive type.
8. Identify all of Java's primitive types.
9. Define object type.
10. Define array type.
11. What is a variable?
12. What is an expression?
13. Identify the two expression categories.
14. What is a literal?
15. Is string literal "The quick brown fox \jumps\ over the lazy dog." legal or illegal? Why?
16. What is an operator?
17. Identify the difference between a prefix operator and a postfix operator.
18. What is the purpose of the cast operator?
19. What is precedence?
20. True or false: Most of Java's operators are left-to-right associative.
21. What is a statement?
22. What is the difference between the while and do-while statements?

23. What is the difference between the break and continue statements?
24. Write a Compass application (the class is named `Compass`) whose `main()` method encapsulates the direction-oriented switch statement example presented earlier in this chapter. You'll need to declare a `direction` variable with the appropriate type and initialize this variable.
25. Create a Triangle application whose `Triangle` class's `main()` method uses a pair of nested for statements along with `System.out.print()` to output a ten-row triangle of asterisks, where each row contains an odd number of asterisks (1, 3, 5, 7, and so on), as shown here:

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

Compile and run this application.

Summary

Before developing Java applications, you need to understand the structure of a Java application. Essentially, every application needs a class that declares a `public static void main(String[] args)` method.

Source code needs to be documented so that you (and any others who have to maintain it) can understand it, now and later. Java provides the comment feature for embedding documentation in source code. Single-line, multiline, and documentation comments are supported.

A single-line comment occupies all or part of a single line of source code. This comment begins with the `//` character sequence and continues with explanatory text. The compiler ignores everything from `//` to the end of the line in which `//` appears.

A multiline comment occupies one or more lines of source code. This comment begins with the `/*` character sequence, continues with explanatory text, and ends with the `*/` character sequence. Everything from `/*` through `*/` is ignored by the compiler.

A Javadoc comment occupies one or more lines of source code. This comment begins with the `/**` character sequence, continues with explanatory text, and ends with the `*/` character sequence. Everything from `/**` through `*/` is ignored by the compiler.

Identifiers are used to name classes, methods, and other source code entities. An identifier consists of letters (A-Z, a-z), digits (0-9), and the underscore. This name must begin with a letter or an underscore. Some identifiers are reserved by Java. Examples include `abstract` and `case`.

Applications process different types of values such as integers, floating-point values, characters, and strings. A type identifies a set of values (and their representation in memory) and a set of operations that transform these values into other values of that set.

A primitive type is a type that's defined by the language and whose values are not objects. Java supports the Boolean, character, byte integer, short integer, integer, long integer, floating-point, and double-precision floating-point primitive types.

An object *type* is a type that's defined by the developer using a class, an interface, an enum, or an annotation type and whose values are objects. User-defined types are also known as reference types.

An array type is a reference type that signifies an array, a region of memory that stores values in equal-size and contiguous slots, which are commonly referred to as elements. This type consists of the element type and one or more pairs of square brackets that indicate the number of dimensions.

Applications manipulate values that are stored in memory, which is symbolically represented in source code through the use of the variables feature. A variable is a named memory location that stores some type of value.

Java provides the expressions feature for initializing variables and for other purposes. An expression combines some arrangement of literals, variable names, method calls, and operators. At runtime, it evaluates to a value whose type is referred to as the expression's type.

A simple expression is a literal, a variable name (containing a value), or a method call (returning a value). Java supports several kinds of literals: string, Boolean true and false, character, integer, floating-point, and null.

CHAPTER 2 LEARNING LANGUAGE FUNDAMENTALS

A compound expression is a sequence of simple expressions and operators, where an operator (a sequence of instructions symbolically represented in source code) transforms its operand expression value(s) into another value.

Java supplies many operators, which are classified by the number of operands that they take. A unary operator takes only one operand, a binary operator takes two operands, and Java's single ternary operator takes three operands.

Operators are also classified as prefix, postfix, and infix. A prefix operator is a unary operator that precedes its operand, a postfix operator is a unary operator that trails its operand, and an infix operator is a binary or ternary operator that's sandwiched between its operands.

Statements are the workhorses of a program. They assign values to variables, control a program's flow by making decisions and/or repeatedly executing other statements, and perform other tasks. A statement can be expressed as a simple statement or as a compound statement.

In Chapter 3, we continue to explore the Java language by examining its support for classes and objects. You also learn more about arrays.

CHAPTER 3

Discovering Classes and Objects

In Chapter 2, we introduced you to the fundamentals of the Java language. You now know how to write simple applications by inserting statements into a class's `main()` method. However, when you try to develop complex applications in this manner, you're bound to find development tedious, slow, and prone to error. Classes and objects address these problems by improving application architecture.

In this chapter, we will introduce you to Java's support for classes and objects. You will learn how to declare classes, construct objects from classes, encapsulate fields and methods in classes, restrict access to fields and methods, initialize classes and objects to appropriate startup values, and how Java takes care of removing objects that are no longer needed.

In Chapter 2, we introduced you to arrays. You learned about array types and how to declare array variables, and you discovered a simple way to create an array. However, Java also provides a more powerful and more flexible way to create arrays, which is somewhat similar to how objects are created. This chapter also extends Chapter 2's array coverage by introducing you to this capability.

Note You will soon learn that classes are the building bricks of our application. Each class typically goes to an own file, and the files can reside in a folder hierarchy. This is called packaging structure. While packages will be described later in Chapter 5, this chapter assumes all files are in the same folder.

Declaring Classes

A *class* is a container for housing an application (as demonstrated in Chapters 1 and 2), and it is also a template for manufacturing objects, which we discuss later in this chapter.

You declare a class by minimally specifying reserved word `class` followed by a name that identifies the class (so that it can be referred to from elsewhere in the source code), followed by a body. The body starts with an open brace character (`{`) and ends with a close brace (`}`). You also prepend a `public` if you want the class to have a `main()` entry point or want the class to be accessible from other packages (we'll talk about packages later in Chapter 5). Sandwiched between these delimiters are various kinds of member declarations. Consider Listing 3-1.

Listing 3-1. Declaring a Skeletal Image Class

```
public class Image {  
    // various member declarations  
}
```

Listing 3-1 declares a class named `Image`, which presumably describes some kind of image for displaying on the screen. By convention, a class's name, which must be a valid Java identifier, begins with an uppercase letter. Furthermore, the first letter of each subsequent word in a multiword class name is capitalized. This is known as *camel casing*.

As file name for a public class you choose the same name as the class name.

Nonpublic classes and public classes can be mixed in one file. However only one class can be marked public then and the file name must match this public class's name.

Classes and Applications

In Chapter 2, we discussed application structure in terms of a `public static void main(String[] args)` method declared in a public class. See Listing 3-2.

Listing 3-2. Declaring Application Entry Points

```
public class App {
    public static void main(String[] args) {
        // statements to execute
    }
}
```

Listing 3-2 must be stored in `App.java`. You would compile this source file as follows:

```
javac App.java
```

You could then run the `main()` method by executing the following command:

```
java App
```

Constructing Objects

`Image`, `SavingsAccount`, and `App` are examples of user-defined types from which *objects* (class instances) can be created. You create these objects by using the `new` operator with a *constructor*, which is a block of code that's declared in a class for constructing an object from that class by initializing it in some manner.

Object creation has the following abstract syntax:

```
new constructor
```

The `new` operator allocates memory to store the object whose type is specified by *constructor* and then *invokes* (calls) *constructor* to initialize the object, which is stored in the *heap* (a region of memory for storing objects). When *constructor* ends, `new` returns a *reference* (a memory address or other identifier) to the object so that it can be accessed elsewhere in the application.

The constructor declaration has the following syntax:

```
public class_name(parameter_list) {
    // statements to execute
}
```

The `public` is optional, but you have to add it for the constructor to be invocable from other packages (see Chapter 5). Unlike in a method declaration (discussed later in this chapter), a constructor doesn't begin with a return type because it cannot return a value to `new`, which calls the constructor. If a constructor could return an arbitrary value, how would Java return that value? After all, the `new` operator returns a reference to an object, and how could `new` also return a constructor value?

A constructor doesn't have an own name. Instead, you must specify the name of the class that declares the constructor. This name is followed by a round bracket-delimited *parameter list*, which is a comma-separated list of zero or more parameter declarations. A *parameter* is a constructor or method variable that receives an expression value passed to the constructor or method when it's called. This expression value is known as an *argument*.

Note The number of arguments passed to a constructor or method, or the number of operator operands, is known as the constructor's, method's, or operator's *arity*.

Consider the following example:

```
Image image = new Image();
```

The `new` operator allocates memory to store an `Image` object in the heap. It then invokes a constructor with no parameters—a *noargument constructor*—to initialize this object. Following initialization, `new` returns a reference to the newly initialized `Image` object. The reference is stored in a variable named `image` whose type is specified as `Image`. (It's common to refer to the variable as an object, as in the `image` object, although it stores only an object's reference and not the object itself.)

Note `new`'s returned reference is represented in source code by reserved word `this`. Wherever `this` appears, it represents the current object. Also, variables that store references are called *reference variables*.

Default Constructor

`Image` doesn't explicitly declare a constructor. When a class doesn't declare a constructor, Java implicitly creates a constructor for that class. The created constructor is known as the *default noargument constructor* because no arguments (demonstrated shortly) appear between its (and) characters when the constructor is invoked.

Note Java doesn't create a default noargument constructor when at least one constructor is declared.

Explicit Constructors

You can explicitly declare a constructor within a class's body. For example, Listing 3-3 enhances Listing 3-1's `Image` class by declaring a single constructor with an empty parameter list.

Listing 3-3. Declaring an `Image` Class with a Single Constructor

```
public class Image {
    public Image() {
        System.out.println("Image() called");
    }
}
```

Listing 3-3's `Image` class declares a noargument constructor for initializing an `Image` object. The declaration consists of a class named `Image` followed by round brackets. This constructor simulates default initialization. It does so by invoking `System.out.println()` to output a message signifying that it's been called.

As previously shown with the default noargument constructor, you would create an object from this class by executing a statement such as the following:

```
Image image = new Image();
```

The constructor would execute the `System.out.println()` method call, which results in the following output:

```
Image() called
```

You will often declare constructors with nonempty parameter lists. Listing 3-4 demonstrates this scenario by declaring an `Image` class with a pair of constructors.

Listing 3-4. Declaring an `Image` Class with Two Constructors

```
public class Image {  
    public Image(String filename) {  
        this(filename, null);  
        System.out.println("Image(String filename) called");  
    }  
  
    public Image(String filename, String imageType) {  
        System.out.println("Image(String filename, String imageType)  
                           called");  
        if (filename != null) {  
            System.out.println("reading " + filename);  
            if (imageType != null)  
                System.out.println("interpreting " + filename + " as storing a " +  
                                   imageType + " image");  
        }  
        // Perform other initialization here.  
    }  
}
```

Listing 3-4's `Image` class first declares an `Image(String filename)` constructor whose parameter list consists of a single *parameter declaration*—a variable's type followed by the variable's name. The `java.lang.String` parameter is named `filename`, signifying that this constructor obtains image content from a file.

Note Throughout this and the remaining chapters, we typically prefix the first use of a predefined type (such as `String`) with the package hierarchy in which the type is stored. For example, `String` is stored in the `lang` subpackage of the `java` package. We do so to help you learn where types are stored in the standard class library. We will have more to say about packages in Chapter 5.

`Image(String filename)` demonstrates that some constructors rely on other constructors to help them initialize their objects. This is done to avoid redundant code, which increases the size of an object and unnecessarily takes memory away from the heap that could be used for other purposes. For example, `Image(String filename)` relies on `Image(String filename, String imageType)` to read the file's image content into memory.

Although it appears otherwise, constructors don't have names (however, it's common to refer to a constructor by specifying the class name and parameter list). A constructor calls another constructor by using keyword `this` and a round bracket-delimited and comma-separated list of arguments. For example, `Image(String filename)` executes `this(filename, null);` to execute `Image(String filename, String imageType)`.

Caution You must use keyword `this` to call another constructor; you cannot use the class's name, as in `Image()`. The `this()` constructor call (when present) must be the first code that's executed within the constructor. This rule prevents you from specifying multiple `this()` constructor calls in the same constructor. Finally, you cannot specify `this()` in a method; constructors can be called only by other constructors and during object creation. (We discuss methods later in this chapter.)

When present, the constructor call must be the first code that's specified within a constructor; otherwise, the compiler reports an error. For this reason, a constructor that calls another constructor can perform additional work only after the other constructor has finished. For example, `Image(String filename)` executes `System.out.println("Image(String filename) called");` after the invoked `Image(String filename, String imageType)` constructor finishes.

The `Image(String filename, String imageType)` constructor declares an `imageType` parameter that signifies the kind of image stored in the file—a Portable Network Graphics (PNG) image, for example. Presumably, the constructor uses `imageType` to speed up processing by not examining the file's contents to learn the image format. When `null` is passed to `imageType`, as happens with the `Image(String filename)` constructor, `Image(String filename, String imageType)` examines file content to learn the format. If `null` was also passed to `filename`, `Image(String filename, String imageType)` wouldn't read the file but would presumably notify the code attempting to create the `Image` object of an error condition.

The following example shows you how to create two `Image` objects, calling the first constructor with argument "image.png" and the second constructor with arguments "image.png" and "PNG". Each object's reference is assigned to a reference variable named `image`, replacing the previously stored reference for the second object assignment:

```
Image image = new Image("image.png");
image = new Image("image.png", "PNG");
```

These constructor calls result in the following output:

```
Image(String filename, String imageType) called
reading image.png
Image(String filename) called
Image(String filename, String imageType) called
reading image.png
interpreting image.png as storing a PNG image
```

In addition to declaring parameters, a constructor can also declare variables within its body to help it perform various tasks. For example, the previously presented `Image(String filename, String imageType)` constructor might create an object from a (currently hypothetical) `File` class that provides the means to read a file's contents. At some point, the constructor instantiates this class and assigns the instance's reference to a variable, as demonstrated by the following example:

```
public Image(String filename, String imageType) {
    System.out.println("Image(String filename, String imageType) called");
    if (filename != null) {
        System.out.println("reading " + filename);
        File file = new File(filename);
        // Read file contents into object.
        if (imageType != null)
            System.out.println("interpreting " + filename + " as storing a " +
                               imageType + " image");
        else
            // Inspect image contents to learn image type.
            ; // Empty statement is used to make if-else syntactically valid.
    }
    // Perform other initialization here.
}
```

As with the `filename` and `imageType` parameters, `file` is a variable that's local to the constructor, and it is known as a *local variable* to distinguish it from a parameter. Although all three variables are local to the constructor, there are two key differences between parameters and local variables:

- The `filename` and `imageType` parameters come into existence at the point where the constructor begins to execute and exist until execution leaves the constructor. In contrast, `file` comes into existence at its point of declaration and continues to exist until the block in which it's declared is terminated (via a closing brace character). This property of a parameter or a local variable is known as *lifetime*.
- The `filename` and `imageType` parameters can be accessed from anywhere in the constructor. In contrast, `file` can be accessed only from its point of declaration to the end of the block in which it's declared. It cannot be accessed before its declaration or after its declaring block, but nested subblocks can access the local variable. This property of a parameter or a local variable is known as *scope*.

Objects and Applications

You previously learned that you can declare a `public static void main(String[] args)` entry-point method in any public class. But, it can be confusing to declare this method in every class of a multiclass application because your application's users won't know which class is the application entry point.

Sometimes, you'll want to declare `main()` in multiple classes for testing purposes. You can then test that class independently of the other classes that make up the application. Alternatively, you might decide to perform unit testing (http://en.wikipedia.org/wiki/Unit_testing) on your classes.

Listing 3-5 presents a three-class application that demonstrates multiclass testing.

Listing 3-5. Testing an Application's Component Classes

```
// File Circle.java
public class Circle {
    public Circle() {
        System.out.println("Circle() called");
    }
}
```

CHAPTER 3 DISCOVERING CLASSES AND OBJECTS

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

// File Rectangle.java
public class Rectangle {
    public Rectangle() {
        System.out.println("Rectangle() called");
    }

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

// File Shapes.java
public class Shapes {
    public static void main(String[] args) {
        Circle c = new Circle();
        Rectangle r = new Rectangle();
    }
}
```

Listing 3-5 declares three classes: `Circle`, `Rectangle`, and `Shapes`. Each class declares a `main()` method. You would execute the following command to compile them:

```
javac Circle.java Rectangle.java Shapes.java
```

To run the `Shapes` application, you would execute the following command:

```
java Shapes
```

You can also run the `Circle` and `Rectangle` component classes to test that these components work properly:

```
java Circle
java Rectangle
```

When you’re finished testing the component classes, you should probably remove the `main()` methods from them to avoid confusion on the part of anyone studying your code. Alternatively, you could document these methods via comments and clearly document the entry-point class.

Encapsulating State and Behaviors

Classes typically combine state with behaviors. *State* refers to *attributes* (such as a counter set to 1 or an account balance storing \$20,000) that are read and/or written when an application runs, and *behaviors* refer to sequences of code (such as calculate a specific factorial or make a deposit into a savings account) that read/write attributes and perform other tasks. Combining state with corresponding behaviors is known as *encapsulation*.

Representing State via Fields

Java represents state via *fields*, which are variables declared within a class’s body. State associated with a class is described by class fields, whereas state associated with objects is described by *object fields* (also known as *instance fields*).

Note By convention, a field’s name begins with a lowercase letter, and the first letter of each subsequent word in a multiword field name is capitalized.

Declaring and Accessing Class Fields

A class field stores an attribute that’s associated with a class. All objects created from that class share this class field. When one object modifies the field’s value, the new value is visible to all current and future objects created from that class.

You declare a class field by specifying the following syntax:

```
static type_name variable_name [ = expression ] ;
```

A class field declaration begins with the `static` reserved word. This reserved word is followed by a type name and a variable name. You can optionally end the declaration by assigning a type-compatible expression, which is known as a *class field initializer*, to the

variable name. Don't forget to specify the trailing semicolon character. In the majority of cases, you also prepend a `private` modifier, which disables direct access to a field from outside the class.

For example, suppose you declare a `Car` class and want to keep track of the number of objects that are created from this class. To accomplish this task, you introduce a counter class field (initialized to 0) into this class. Check out Listing 3-6.

Listing 3-6. Adding a counter Class Field to a Car Class

```
public class Car {
    // No 'private' here, because we want to access
    // the field directly from outside. If not from the same
    // package, we have to prepend 'public'.
    static int counter = 0;

    public Car() {
        counter++;
    }
}
```

Listing 3-6 declares an `int` class field named `counter`. The `static` prefix implies that there is only one copy of this field and not one copy per object. This `counter` field is explicitly initialized to 0. Each time an object is created, the `counter++` expression in the `Car()` constructor increases `counter` by 1.

Listing 3-7 presents a `Cars` application class that demonstrates `Car` and `counter`.

Listing 3-7. Demonstrating the Car Class and Its counter Field

```
public class Cars {
    public static void main(String[] args) {
        System.out.println(Car.counter);
        Car myCar = new Car();
        System.out.println(Car.counter);
        Car yourCar = new Car();
        System.out.println(Car.counter); // direct field access
    }
}
```

Compile Listing 3-7 as follows:

```
javac Car.java Cars.java
```

Run the Cars application as follows:

```
java Cars
```

You should observe the following output, which reveals counter's initial value and that this variable is incremented for each created Car object:

```
0  
1  
2
```

It isn't necessary to explicitly initialize counter to 0. When a class is loaded into memory, class fields are initialized to default zero/false/null values. For example, counter is implicitly initialized to 0.

Note Class fields are initialized by zeroing their bits. You interpret this value as literal false, '\u0000', 0.0, 0.of, 0, 0L, or null depending on the field's type.

Within a class declaration, a class field is accessed directly, as in counter++. When accessing a class field from outside of the class, you must prepend the class name followed by the member access operator to the class field name. For example, to access counter from Cars, you must specify Car.counter.

Note We could have accessed counter via the myCar and yourCar reference variables, for example, myCar.counter and yourCar.counter. However, the preferred approach is to use the class name, as in Car.counter. This approach is preferred because it's easier to tell that a class field is being accessed.

Class fields can be modified. If you want a class field to be *constant* (an unchangeable variable), you must declare the class field to be final by prefixing its declaration with the final reserved word. Listing 3-8 presents an example.

Listing 3-8. Declaring a Constant in the Employee class

```
public class Employee {
    public final static int RETIREMENT_AGE = 65;
}
```

Listing 3-8 declares an integer constant named `RETIREMENT_AGE`. If you attempt to modify this variable subsequently, as in `RETIREMENT_AGE = 32;`, the compiler reports an error. Although reserved word `final` precedes reserved word `static` in this example, you can switch this order, which results in `static final int RETIREMENT_AGE = 65;`.

The `RETIREMENT_AGE` declaration is an example of a *compile-time constant*. Because there is only one copy of its value (because of `static`), and because this value will never change (thanks to `final`), the compiler can optimize the bytecode by inserting the constant value into all calculations where it appears. Code runs faster because it doesn't have to access a read-only class field.

A class field is created when the class that declares this field is loaded into the heap. The class field is destroyed when the class is unloaded, typically when the virtual machine ends. This property of a class field is known as *lifetime*.

A class field is visible to the entire class in which it's declared. In other words, the class field can be accessed from anywhere in its class. Unless explicitly hidden via `private` (discussed later in this chapter), the class field is visible to code outside of the class (but in the same package). This property of a class field is known as *scope*.

Declaring and Accessing Instance Fields

An instance field stores an attribute that's associated with an object. Each object maintains a separate copy of the attribute. For example, one object might have red as its color attribute, whereas a second object has green as its color attribute.

You declare an instance field by specifying the following syntax:

type_name variable_name [= expression] ;

An instance field declaration begins with a type name followed by a variable name. You can optionally end the declaration by assigning a type-compatible expression, which is known as an *instance field initializer*, to the variable name. Don't forget to specify the trailing semicolon character.

For example, you want to model a car in terms of its make, model, and number of doors. You don't use class fields to store these attributes because cars have different

makes, models, and door counts. Listing 3-9 presents a Car class with three instance field declarations for these attributes.

Listing 3-9. Declaring a Car Class with make, model, and numDoors Instance Fields

```
public class Car {
    String make;
    String model;
    int numDoors;
}
```

Listing 3-9 declares two `String` instance fields named `make` and `model`. It also declares an `int` instance field named `numDoors`. Because we prepended neither `public` nor `private` to the field declarations, we will be able to access the fields directly from inside the same package (see Chapter 5 for packages).

When an object is created, instance fields are initialized to default zero values, which you interpret at the source code level as literal value `false`, `'\u0000'`, `0`, `0L`, `0.0`, `0.0F`, or `null` (depending on field type). For example, if you executed `Car car = new Car();`, `make` and `model` would be initialized to `null` and `numDoors` would be initialized to `0`.

Listing 3-10 presents the source code to a Cars application class that directly accesses its instance fields.

Listing 3-10. Directly Accessing Instance Fields

```
public class Cars {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.make = "Toyota"; // because in-package access allowed
        myCar.model = "Camry";
        myCar.numDoors = 4;
        System.out.println("Make = " + myCar.make);
        System.out.println("Model = " + myCar.model);
        System.out.println("Number of doors = " + myCar.numDoors);
    }
}
```

Compile Listing 3-10 as follows:

```
javac Car.java Cars.java
```

Run the Cars application as follows:

```
java Cars
```

You should observe the following output, which reveals the values assigned to the Car object's make, model, and numDoors instance fields:

```
Make = Toyota
Model = Camry
Number of doors = 4
```

Inside a class declaration, an instance field is accessed directly, as in `System.out.println(numDoors);`. When accessing an instance field from outside of an object, you must prepend the desired object reference variable followed by the member access operator to the instance field name. For example, we specified `myCar.make` to access the make field of the `myCar` object in Listing 3-10.

You can explicitly initialize an instance field when declaring that field to provide a nonzero default value, which overrides the default zero value. Listing 3-11 demonstrates this point.

Listing 3-11. Initializing Car's numDoors Instance Field to a Default Nonzero Value

```
public class Car {
    String make;
    String model;
    int numDoors = 4;
}
```

Listing 3-11 explicitly initializes `numDoors` to 4 because the developer has assumed that most cars being modeled by this class have four doors. When `Car` is initialized via the default noargument `Car()` constructor, which is responsible for assigning 4 to `numDoors`, the developer only needs to initialize the `make` and `model` instance fields for those cars that have four doors.

You could remove the `myCar.numDoors = 4;` assignment from Listing 3-10, and you would still observe the same output.

It's usually not a good idea to directly initialize an object's instance fields, and you will learn why when we discuss information hiding later in this chapter. Instead, you should perform this initialization in the class's constructor(s); see Listing 3-12.

Listing 3-12. Initializing Car's Instance Fields via Constructors

```
public class Car {
    String make;
    String model;
    int numDoors;

    public Car(String make, String model) {
        this(make, model, 4);
    }

    public Car(String make, String model, int nDoors) {
        this.make = make;
        this.model = model;
        numDoors = nDoors;
    }
}
```

Listing 3-12's Car class declares `Car(String make, String model)` and `Car(String make, String model, int nDoors)` constructors. The first constructor lets you specify make and model, whereas the second constructor lets you specify values for the three instance fields.

The first constructor executes `this(make, model, 4);` to pass the values of its make and model parameters, along with a default value of 4 to the second constructor. Doing so demonstrates an alternative to initializing an instance field explicitly, and it is preferable from a code maintenance perspective.

The `Car(String make, String model, int numDoors)` constructor demonstrates another use for keyword `this`. Specifically, it demonstrates a scenario where constructor parameters have the same names as the class's instance fields. Prefixing a variable name with `this.` causes the Java compiler to create bytecode that accesses the instance field. For example, `this.make = make;` assigns the `make` parameter's `String` object reference to this (the current) `Car` object's `make` instance field. If `make = make;` was specified

instead, it would accomplish nothing by assigning `make`'s value to itself; a Java compiler might not generate code to perform the unnecessary assignment. In contrast, `this.` isn't necessary for the `numDoors = nDoors;` assignment, which initializes the `numDoors` field from the `nDoors` parameter value.

Note To minimize error (by forgetting to prefix a field name with `this.`), it's preferable to keep field names and parameter names distinct (such as `numDoors` and `nDoors`). This way, you wouldn't have to worry about the `this.` prefix (and forgetting to specify it).

Listing 3-13 demonstrates instance field initialization via these constructors.

Listing 3-13. Demonstrating Instance Field Initialization via Constructors

```
public class Cars {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Camry");
        System.out.println("Make = " + myCar.make);
        System.out.println("Model = " + myCar.model);
        System.out.println("Number of doors = " + myCar.numDoors);
        System.out.println();
        Car yourCar = new Car("Mazda", "RX-8", 2);
        System.out.println("Make = " + yourCar.make);
        System.out.println("Model = " + yourCar.model);
        System.out.println("Number of doors = " + yourCar.numDoors);
    }
}
```

Compile Listing 3-13 as follows:

```
javac Car.java Cars.java
```

Run the `Cars` application as follows:

```
java Cars
```

You should observe the following output, which reveals that each of the `myCar` and `yourCar` objects has different instance field values:

```
Make = Toyota
Model = Camry
Number of doors = 4

Make = Mazda
Model = RX-8
Number of doors = 2
```

Instance fields can be modified. If you want an instance field to be constant, you must declare the instance field to be final by prefixing its declaration with the `final` reserved word. Listing 3-14 presents an example.

Listing 3-14. Declaring a Constant in the `Employee` class

```
public class Employee {
    final int RETIREMENT_AGE = 65;
}
```

Listing 3-14 declares an integer constant named `RETIREMENT_AGE`. If you attempt to modify this variable subsequently, which you can only do in an object context (such as `emp.RETIREMENT_AGE = 32;`), the compiler reports an error.

Each object receives a copy of a read-only instance field. Regarding Listing 3-14, each `Employee` object would receive a copy of `RETIREMENT_AGE`. Because this is wasteful of memory, you would be better off also declaring `RETIREMENT_AGE` to be `static` so that there is only one copy.

A `final` instance field must be initialized, as part of the field's declaration or in the class's constructor. When initialized in the constructor, the read-only instance field is known as a *blank final* because it doesn't have a value (the field is blank) until one is assigned to it in the constructor. Because a constructor can potentially assign a different value to each object's blank final, these read-only variables are only true constants in the contexts of their objects. Check out Listing 3-15.

Listing 3-15. Declaring a Different Constant for Each Employee Class

```
public class Employee {
    final int ID;
    static int counter;

    Employee() {
        ID = counter++;
    }
}
```

Each Employee object receives a different read-only ID value.

An instance field is created when an object is created from the class that declares this field. The instance field is destroyed when the object is garbage collected. (We discuss garbage collection later in this chapter.) This property of an instance field is known as *lifetime*.

An instance field is visible to the entire class in which it's declared. In other words, the instance field can be accessed from anywhere in its class. Unless explicitly hidden via `private` (discussed later in this chapter), the instance field is visible to code outside of the class, but only in the same package and in the context of an object reference variable. This property of an instance field is known as *scope*.

Reviewing Field-Access Rules

The previous examples of field access may seem confusing because you can sometimes specify the field's name directly, whereas you need to prefix a field name with an object reference or a class name and the member access operator at other times. The following rules dispel this confusion by giving you guidance on how to access fields from the various contexts:

- Specify the name of a class field as is from anywhere within the same class as the class field declaration. For example: `counter`
- Specify the name of a class field's class, followed by the member access operator, followed by the name of the class field from outside the class. For example: `Car.counter`

- Specify the name of an instance field as is from any instance method, constructor, or instance initializer (discussed later) in the same class as the instance field declaration. For example: `numDoors`
- Specify an object reference, followed by the member access operator, followed by the name of the instance field from any class method or class initializer (discussed later) within the same class as the instance field declaration or from outside the class. For example: `Car car = new Car(); car.numDoors = 2;`

Although the final rule might seem to imply that you can access an instance field from a class context, this isn't the case. Instead, you're accessing the field from an object context.

The previous access rules aren't exhaustive because there are two more field-access scenarios to consider: declaring a parameter or local variable with the same name as an instance field or as a class field. In either scenario, the local variable/parameter is said to *shadow* (hide or mask) the field.

If you've declared a parameter/local variable that shadows a field, you can rename it, or you can use the member access operator with a class name (class field) or reserved word `this` (instance field) to identify the field explicitly. For example, Listing 3-12's `Car(String make, String model, int nDoors)` constructor demonstrated this latter solution by specifying statements such as `this.make = make;` to distinguish an instance field from a same-named parameter/local variable.

Representing Behaviors via Methods

Java represents behaviors via *methods*, which are named blocks of code declared within a class's body. Behaviors associated with a class are described by *class methods*, whereas behaviors associated with objects are described by *object methods* (also known as *instance methods*).

Note By convention, a method's name begins with a lowercase letter, and the first letter of each subsequent word in a multiword method name is capitalized.

Declaring and Invoking Class Methods

A *class method* stores a behavior that's associated with a class. All objects created from that class share this class method. The class method has no direct access to instance fields. The only way to access these fields is to access them in the context of a specific object, via an object reference variable and the member access operator.

A class method has the following syntax:

```
static return_type name(parameter_list) {  
    // statements to execute  
}
```

A class method begins with a header that starts with `static`. This reserved word is followed by a return type that specifies the type of value that the method returns. The return type is either a primitive type name (such as `int` or `double`), a reference type name (such as `String`), or reserved word `void` when the class method doesn't return any kind of value.

The class method's name follows its return type. This name must be a legal Java identifier that isn't a reserved word.

Note A method's name and the number, types, and order of its parameters are known as its *signature*.

As with a constructor, the class method header concludes with a parameter list that lets you specify the kinds of data items that are passed to the method for processing.

The header is followed by a brace-delimited body of statements that execute when the class method is invoked.

You may prepend a `public` in front of the declaration to express a method being accessible from outside the class. Also, a `private` tells a method can only be called from inside the same class. The absence of `public` and `private` means the method is callable from anywhere inside the same package.

You've already encountered the `public static void main(String[] args)` class method that serves as a class's entry point. The `java` tool creates an array of `String` objects, one object per command-line argument, and passes this array to the `args` parameter so that code within `main()` can process these arguments when it invokes `main()`.

For a second class method example, consider Listing 3-16's Utilities class and its `dumpMatrix()` class method.

Listing 3-16. A Utilities Class with a Single Class Method for Dumping a Matrix in Tabular Format

```
public class Utilities {
    private static void dumpMatrix(float[][] matrix) {
        for (int row = 0; row < matrix.length; row++) {
            for (int col = 0; col < matrix[row].length; col++)
                System.out.print(matrix[row][col] + " ");
            System.out.print("\n");
        }
    }

    public static void main(String[] args) {
        float[][] temperatures = {
            { 37.0f, 14.0f, -22.0f },
            { 0.0f, 29.0f, -5.0f }
        };
        dumpMatrix(temperatures);
        System.out.println();
        Utilities.dumpMatrix(temperatures);
    }
}
```

The `dumpMatrix()` class method dumps the contents of a two-dimensional array in tabular format to the standard output stream. The method's header tells us that the method doesn't return anything (its return type is `void`), the method is named `dumpMatrix`, and the method contains a parameter list consisting of a single parameter named `matrix`, which is of type `float[][]`.

Within the method, `row` and `col` are declared as local variables. The `row` variable is declared in the outer `for` loop. Its scope ranges from its `for` loop header through the end of the brace-delimited block that follows this header. The `col` variable is declared in the inner `for` loop. Its scope ranges from its `for` loop header through the end of the single-statement block (`System.out.print(matrix[row][col] + " ");`) that follows this header.

A third class method, `public static void main(String[] args)`, is present for testing purposes. After declaring a `temperatures` matrix, it shows you two ways to invoke `dumpMatrix()`. The invocation without a prefix is used when the class method being invoked is declared in the same class. A prefix is typically specified only when the class method is being called from a different class.

Compile Listing 3-16 as follows:

```
javac Utilities.java
```

Run the `Utilities` application as follows:

```
java Utilities
```

You should observe the following output:

```
37.0 14.0 -22.0
0.0 29.0 -5.0

37.0 14.0 -22.0
0.0 29.0 -5.0
```

Declaring and Invoking Instance Methods

An *instance method* stores a behavior that's associated with an object. Unlike a class method, an instance method can directly access an object's instance fields.

An instance method has the following syntax:

```
return_type name(parameter_list) {
    // statements to execute
}
```

Apart from the absence of the `static` reserved word, this syntax is the same as the syntax for a class method.

Listing 3-17 refactors Listing 3-12's `Car` class also to include a `printDetails()` instance method.

Listing 3-17. Extending the Car class with an Instance Method That Prints Details

```
public class Car {
    private String make; // Because of 'private' access only from inside
    private String model;
    private int numDoors;

    public Car(String make, String model) {
        this(make, model, 4);
    }

    public Car(String make, String model, int nDoors) {
        this.make = make;
        this.model = model;
        numDoors = nDoors;
    }

    public void printDetails() {
        System.out.println("Make = " + make);
        System.out.println("Model = " + model);
        System.out.println("Number of doors = " + numDoors);
        System.out.println();
    }
}
```

The `printDetails()` method has its return type set to `void` because it doesn't return a value. It prints out the car's make, model, and number of doors and then outputs a blank line separator.

Listing 3-18 presents a Cars application that creates `Car` objects and invokes each object's `printDetails()` instance method.

Listing 3-18. Demonstrating Instance Method Calls

```
public class Cars {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Camry");
        myCar.printDetails();
```

```
    Car yourCar = new Car("Mazda", "RX-8", 2);
    yourCar.printDetails();
}
}
```

The `main()` method instantiates `Car`, passing appropriate make and model strings to its two-parameter constructor; the number of doors defaults to 4. It then invokes `printDetails()` on the returned object reference to print these values. Next, `main()` creates a second `Car` object via the three-parameter constructor and prints out this object's make, model, and number of doors.

`main()`'s invocation of `printDetails()` demonstrates that an instance method is always invoked in an object reference context.

Compile Listing 3-18 as follows:

```
javac Car.java Cars.java
```

Run the `Cars` application as follows:

```
java Cars
```

You should observe the following output:

```
Make = Toyota
Model = Camry
Number of doors = 4

Make = Mazda
Model = RX-8
Number of doors = 2
```

Returning from a Method via the Return Statement

Java provides the `return` statement to terminate method execution and return control to the method's *caller* (the code sequence that called the method). This statement has the following syntax:

```
return [ expression ] ;
```

You can specify `return` without an expression and will typically use this form of the `return` statement to return prematurely from a method—or from a constructor. In other words, you don't want to execute all of the statements in the method/constructor. Check out Listing 3-19.

Listing 3-19. Returning Prematurely from an Instance Method

```
public class Employee {
    private String name;

    public Employee(String name) {
        setName(name);
    }

    public void setName(String name) {
        if (name == null) {
            System.out.println("name cannot be null");
            return;
        } else
            this.name = name;
    }

    public static void main(String[] args) {
        Employee john = new Employee(null);
    }
}
```

Listing 3-19's `Employee(String name)` constructor invokes the `setName()` instance method to initialize the `name` instance field. Providing a separate method for this purpose is a good idea because it lets you initialize the instance field at construction time and also at a later time. (Perhaps the employee changes his or her name.)

`setName()` uses an `if` statement to detect an attempt to assign the null reference to the `name` field. When such an attempt is detected, it outputs the “`name cannot be null`” error message and returns prematurely from the method so that the null value cannot be assigned (and replace a previously assigned name).

Compile Listing 3-19 as follows:

```
javac Employee.java
```

Run the Employee application as follows:

```
java Employee
```

You should observe the following output:

```
name cannot be null
```

The previous form of the return statement isn't legal in a method that returns a value. For such methods, Java lets the method return a value (whose type must be compatible with the method's return type). The following example demonstrates this version:

```
static double divide(double dividend, double divisor) {  
    if (divisor == 0.0) {  
        System.out.println("cannot divide by zero");  
        return 0.0;  
    }  
    return dividend / divisor;  
}
```

divide() uses an if statement to detect an attempt to divide its first argument by 0.0 and outputs an error message when this attempt is detected. Furthermore, it returns 0.0 to signify this attempt. If there is no problem, the division is performed and the result is returned.

Caution You cannot use this form of the return statement in a constructor because constructors don't have return types.

Method-Call Stack

Just in case you are interested, Java method invocations require a *method-call stack* (also known as a *method-invocation stack*) to keep track of the statements to which execution must return.

When a method is invoked, the virtual machine pushes its arguments and the address of the first statement to execute following the invoked method onto the method-call stack. The virtual machine also allocates stack space for the method's local variables. When the method returns, the virtual machine removes local variable space, pops the

address and arguments off of the stack, and transfers execution to the statement at this address. Two or more instance method calls can be chained together via the member access.

Chaining Together Instance Method Calls

Two or more instance method calls can be chained together via the member access operator, which results in more compact code. To accomplish instance method call chaining, you need to rearchitect your instance methods somewhat differently, which Listing 3-20 reveals.

Listing 3-20. Implementing Instance Methods So That Calls to These Methods Can Be Chained Together

```
public class SavingsAccount {  
    private int balance;  
  
    SavingsAccount deposit(int amount) {  
        balance += amount;  
        return this;  
    }  
  
    public SavingsAccount printBalance() {  
        System.out.println(balance);  
        return this;  
    }  
  
    public static void main(String[] args) {  
        new SavingsAccount().deposit(1000).printBalance();  
    }  
}
```

To achieve instance method call chaining, Listing 3-20 declares `SavingsAccount` as the return type for the `deposit()` and `printBalance()` methods. Also, each method specifies `return this;` (return current object's reference) as its last statement.

In the `main()` method, `new SavingsAccount().deposit(1000).printBalance();` performs the following tasks:

1. It creates a `SavingsAccount` object.
2. It uses the returned `SavingsAccount` reference to invoke `SavingsAccount`'s `deposit()` instance method, to add one thousand dollars to the savings account (we're ignoring cents for convenience).
3. It uses `deposit()`'s returned `SavingsAccount` reference to invoke `SavingsAccount`'s `printBalance()` instance method to output the account balance.

Compile Listing 3-20 as follows:

```
javac SavingsAccount.java
```

Run the `SavingsAccount` application as follows:

```
java SavingsAccount
```

You should observe the following output:

```
1000
```

Passing Arguments to Methods

A method call includes a list of (zero or more) arguments being passed to the method. Java passes arguments to methods via a style of argument passing called *pass-by-value*, which the following example demonstrates:

```
Employee emp = new Employee("John ");
int recommendedAnnualSalaryIncrease = 1000;
printReport(emp, recommendAnnualSalaryIncrease);
printReport(new Employee("Cuifen"), 1500);
```

Pass-by-value passes the value of a variable (e.g., the reference value stored in `emp` or the 1000 value stored in `recommendedAnnualSalaryIncrease`) or the value of some other expression (such as `new Employee("Cuifen")` or 1500) to the method.

Because of pass-by-value, you cannot assign a different `Employee` object's reference to `emp` from inside `printReport()` via the `printReport()` parameter for this argument. After all, you have only passed a copy of `emp`'s value to the method.

Many methods and constructors require you to pass a fixed number of arguments when they are called. However, Java also provides the ability to pass a variable number of arguments; such methods/constructors are often referred to as *varargs methods/constructors*. To declare a method or constructor that takes a variable number of arguments, specify three consecutive periods after the type name of the method's/constructor's rightmost parameter.

The following example presents a `sum()` method that accepts a variable number of arguments:

```
static double sum(double... values) {  
    int total = 0;  
    for (int i = 0; i < values.length; i++)  
        total += values[i];  
    return total;  
}
```

`sum()`'s implementation totals the arguments passed to this method, for example, `sum(10.0, 20.0)` or `sum(30.0, 40.0, 50.0)`. (Behind the scenes, these arguments are stored in a one-dimensional array, as evidenced by `values.length` and `values[i]`.) After these values have been totaled, this total is returned via the return statement.

Invoking Methods Recursively

A method normally executes statements that may include calls to other methods, such as `printDetails()` invoking `System.out.println()`. However, it's occasionally convenient to have a method call itself. This scenario is known as *recursion*.

For example, suppose you need to write a method that returns a *factorial* (the product of all the positive integers up to and including a specific integer). For example, $3!$ ($!$ is the mathematical symbol for factorial) equals $3 \times 2 \times 1$ or 6.

Your first approach to writing this method might consist of the code presented in the following example:

CHAPTER 3 DISCOVERING CLASSES AND OBJECTS

```
static int factorial(int n) {  
    int product = 1;  
    for (int i = 2; i <= n; i++)  
        product *= i;  
    return product;  
}
```

Although this code accomplishes its task (via iteration), `factorial()` could also be written according to the following example's recursive style:

```
static int factorial(int n) {  
    if (n == 1)  
        return 1; // base problem  
    else  
        return n * factorial(n - 1);  
}
```

The recursive approach takes advantage of being able to express a problem in simpler terms of itself. According to this example, the simplest problem, which is also known as the *base problem*, is $1!(1)$.

When an argument greater than 1 is passed to `factorial()`, this method breaks the problem into a simpler problem by calling itself with the next smaller argument value. Eventually, the base problem will be reached.

For example, calling `factorial(4)` results in the following stack of expressions:

```
4 * factorial(3)  
3 * factorial(2)  
2 * factorial(1)
```

This last expression is at the top of the stack. When `factorial(1)` returns 1, these expressions are evaluated as the stack begins to unwind:

- $2 * \text{factorial}(1)$ now becomes $2 * 1$ (2).
- $3 * \text{factorial}(2)$ now becomes $3 * 2$ (6).
- $4 * \text{factorial}(3)$ now becomes $4 * 6$ (24).

Recursion provides an elegant way to express many problems. Additional examples include searching tree-based data structures for specific values and, in a hierarchical filesystem, finding and outputting the names of all files that contain specific text.

Caution Recursion consumes stack space, so make sure that your recursion eventually ends in a base problem; otherwise, you will run out of stack space and your application will be forced to terminate.

Overloading Methods

Java lets you introduce methods with the same name but different parameter lists into the same class. This feature is known as *method overloading*. When the compiler encounters a method-invocation expression, it compares the called method's argument list with each overloaded method's parameter list as it looks for the correct method to invoke.

Two same-named methods are overloaded when their parameter lists differ in number or order of parameters. Alternatively, two same-named methods are overloaded when at least one parameter differs in type. Listing 3-21 presents an application that demonstrates these scenarios in an instance method context.

Listing 3-21. Demonstrating Method Overloading

```
public class MO {  
    public int add(int a, int b){  
        System.out.println("add(int, int) called");  
        return a + b;  
    }  
  
    public int add(int a, int b, int c) {  
        System.out.println("add(int, int, int) called");  
        return a + b + c;  
    }  
  
    public double add(double a, double b) {  
        System.out.println("add(double, double) called");  
        return a + b;  
    }  
  
    public static void main(String[] args) {  
        MO mo = new MO();  
        int result = mo.add(10, 20);  
    }  
}
```

```

        System.out.println("Result = " + result);
        result = mo.add(10, 20, 30);
        System.out.println("Result = " + result);
        double result2 = mo.add(5.0, 8.0);
        System.out.println("Result2 = " + result2);
    }
}

```

After creating an `MO` object, `main()` invokes this object's `int add(int a, int b)` instance method to add the two integer arguments together and save the result, which is subsequently output. Next, `main()` invokes `int add(int a, int b, int c)` to add the three integer arguments together and outputs the result. Finally, `main()` invokes `double add(double a, double b)` to add the two double-precision floating-point arguments together and outputs the result.

Compile Listing 3-21 as follows:

```
javac MO.java
```

Run the `MO` application as follows:

```
java MO
```

You should observe the following output:

```

add(int, int) called
Result = 30
add(int, int, int) called
Result = 60
add(double, double) called
Result2 = 13.0

```

You cannot overload a method by changing only the return type. For example, `double sum(double... values)` and `int sum(double... values)` are not overloaded. These methods are not overloaded because the compiler doesn't have enough information to choose which method to call when it encounters `sum(1.0, 2.0)` in source code.

Reviewing Method-Invocation Rules

The previous examples of method invocation may seem confusing because you can sometimes specify the method's name directly, whereas you need to prefix a method name with an object reference or a class name and the member access operator at other times. The following rules dispel this confusion by giving you guidance on how to invoke methods from the various contexts:

- Specify the name of a class method as is from anywhere within the same class as the class method. For example:
`dumpMatrix(temperatures);`
- Specify the name of the class method's class, followed by the member access operator, followed by the name of the class method from outside the class. For example: `Utilities.dumpMatrix(temperatures);` (You can also invoke a class method via an object instance, but that is considered bad form because it hides from casual observation the fact that a class method is being invoked.)
- Specify the name of an instance method as is from any instance method, constructor, or instance initializer in the same class as the instance method. For example: `setName(name);`
- Specify an object reference, followed by the member access operator, followed by the name of the instance method from any class method or class initializer within the same class as the instance method or from outside the class. For example: `Car car = new Car("Toyota", "Camry"); car.printDetails();`

Although the latter rule might seem to imply that you can call an instance method from a class context, this isn't the case. Instead, you call the method from an object context.

Also, don't forget to make sure that the number of arguments passed to a method, along with the order in which these arguments are passed, and the types of these arguments agree with their parameter counterparts in the method being invoked.

Note Field access and method call rules are combined in expression System.out.println();, where the leftmost member access operator accesses the out class field (of type java.io.PrintStream) in the java.lang.System class, and where the rightmost member access operator calls this field's println() method. You'll learn about PrintStream in Chapter 12 and System in Chapter 7.

Hiding Information

Every class *X* exposes an *interface*, which are the constructors, methods, and (possibly) fields that can be accessed from outside of *X*. For example, in Listing 3-6, class field counter can be accessed from outside of its containing Car class so counter is part of that class's interface. Also, in Listing 3-17, instance method printDetails() can be accessed from outside of its containing Car class so printDetails() is part of that class's interface.

An interface serves as a contract between a class and its *clients*, which are external classes that communicate with the class and/or its instances by accessing fields (typically public static final fields, or constants) and calling constructors and methods. The contract is such that the class promises to not change its interface, which would break dependent clients.

X also provides an *implementation* (the code within exposed methods along with optional helper methods and optional supporting fields that shouldn't be exposed) that codifies the interface. *Helper methods* are methods that assist exposed methods and shouldn't be exposed themselves.

When designing a class, your goal is to expose a useful interface while hiding details of that interface's implementation. For example, consider Listing 3-12's Car class. This class exposes make, model, and numDoors instance fields along with a pair of constructors. Many developers would regard these instance fields to belong to Car's implementation and should be hidden. After all, they could be renamed, their types could be changed, or they could be removed in a future version of the class, which would break dependent client code.

Note In contrast to nonconstant fields, you often expose constant fields. For example, Listing 3-8's Employee class exposes a RETIREMENT_AGE constant class field. However, you would probably hide constant fields that are only relevant within the context of the class in which they are declared. You would do so to prevent them from cluttering up the class's interface and exposing clients to unnecessary details.

You hide the implementation to prevent developers from accidentally accessing parts of your class that don't belong to the class's interface so that you're free to change the implementation without breaking client code. Hiding the implementation is often referred to as *information hiding*. Furthermore, many developers consider implementation hiding to be part of encapsulation.

Java supports implementation hiding by providing four levels of access control, where three of these levels are indicated via a reserved word. You can use the following access-control levels to control access to fields, methods, and constructors and two of these levels to control access to classes:

- *Public*: A field, method, or constructor that is declared public is accessible from anywhere. Classes can be declared public as well. We typically declare a class that contains the public static void main(String[] args) entry-point method public. Classes that are to be visible outside their packages (see Chapter 5) are also declared public. Public classes must be declared in files whose names match the class names.
- *Protected*: A field, method, or constructor that is declared protected is accessible from all classes in the same package as the member's class as well as subclasses of that class regardless of package.
- *Private*: A field, method, or constructor that is declared private cannot be accessed from outside the class in which it's declared.
- *Package-private*: In the absence of an access-control reserved word, a field, method, or constructor is only accessible to classes within the same package as the member's class. The same is true for non-public classes. The absence of public, protected, or private implies package-private.

You will often declare your class's instance fields to be `private` and provide special `public` instance methods for setting and getting their values. By convention, methods that set field values have names starting with `set` and are known as *setters*. Similarly, methods that get field values have names with `get` (or `is`, for Boolean fields) prefixes and are known as *getters*. Listing 3-22 demonstrates this pattern in the context of an `Employee` class declaration.

Listing 3-22. Separation of Interface from Implementation

```
public class Employee {  
    private String name;  
  
    public Employee(String name) {  
        setName(name);  
    }  
  
    public void setName(String empName) {  
        name = empName; // Assign the empName argument to the name field.  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

Listing 3-22 presents an interface consisting of the `public Employee` class, its `public` constructor, and its `public` setter/getter methods. This class and these members can be accessed from anywhere. The implementation consists of the `private name` field and constructor/method code, which is only accessible within the `Employee` class.

It might seem pointless to go to all this bother when you could simply omit `private` and access the `name` field directly. However, suppose you're told to introduce a new constructor that takes separate first and last name arguments and new methods that set/get the employee's first and last names into this class. Furthermore, suppose that it's been determined that the first and last names will be accessed more often than the entire name. Listing 3-23 reveals these changes.

Listing 3-23. Revising Implementation Without Affecting Existing Interface

```
public class Employee {  
    private String firstName;  
    private String lastName;  
  
    public Employee(String name) {  
        setName(name);  
    }  
  
    public Employee(String firstName, String lastName) {  
        setName(firstName + " " + lastName);  
    }  
  
    public void setName(String name) {  
        // Assume that the first and last names are separated by a  
        // single space character. indexOf() locates a character in a  
        // string; substring() returns a portion of a string.  
        setFirstName(name.substring(0, name.indexOf(' ')));  
        setLastName(name.substring(name.indexOf(' ') + 1));  
    }  
  
    public String getName() {  
        return getFirstName() + " " + getLastName();  
    }  
  
    public void setFirstName(String empFirstName){  
        firstName = empFirstName;  
    }  
  
    public String getFirstName(){  
        return firstName;  
    }  
  
    public void setLastName(String empLastName){  
        lastName = empLastName;  
    }
```

```
public String getLastName(){  
    return lastName;  
}  
}
```

Listing 3-23 reveals that the name field has been removed in favor of new firstName and lastName fields, which were added to improve performance. Because setFirstName() and setLastName() will be called more frequently than setName(), and because getFirstName() and getLasttName() will be called more frequently than getName(), it's more performant (in each case) to have the first two methods set/get firstName's and lastName's values rather than merging either value into/extracting this value from name's value.

Listing 3-23 also reveals setName() calling setFirstName() and setLastName(), and getName() calling getFirstName() and getLasttName(), rather than directly accessing the firstName and lastName fields. Although avoiding direct access to these fields isn't necessary in this example, imagine another implementation change that adds more code to setFirstName(), setLastName(), getFirstName(), and getLasttName(); not calling these methods will result in the new code not executing.

Client code (code that instantiates and uses a class, such as Employee) will not break when Employee's implementation changes from that shown in Listing 3-22 to that shown in Listing 3-23, because the original interface remains intact, although the interface has been extended. This lack of breakage results from hiding Listing 3-22's implementation, especially the name field.

Note setName() invokes the String class's indexOf() and substring() methods. You'll learn about these and other String methods in Chapter 7.

Java provides a little-known information hiding-related language feature that lets one object (or class method/initializer) access another object's private fields or invoke its private methods. Listing 3-24 provides a demonstration.

Listing 3-24. One Object Accessing Another Object's `private` Field

```
public class PrivateAccess {  
    private int x;  
  
    PrivateAccess(int x) {  
        this.x = x;  
    }  
  
    boolean equalTo(PrivateAccess pa){  
        return pa.x == x;  
    }  
  
    public static void main(String[] args) {  
        PrivateAccess pa1 = new PrivateAccess(10);  
        PrivateAccess pa2 = new PrivateAccess(20);  
        PrivateAccess pa3 = new PrivateAccess(10);  
        System.out.println("pa1 equal to pa2: " + pa1.equalTo(pa2));  
        System.out.println("pa2 equal to pa3: " + pa2.equalTo(pa3));  
        System.out.println("pa1 equal to pa3: " + pa1.equalTo(pa3));  
        System.out.println(pa2.x);  
    }  
}
```

Listing 3-24's `PrivateAccess` class declares a `private int` field named `x`. It also declares an `equalTo()` method that takes a `PrivateAccess` argument. The idea is to compare the argument object with the current object to determine if they are equal.

The equality determination is made by using the `==` operator to compare the value of the argument object's `x` instance field with the value of the current object's `x` instance field, returning Boolean true when they are the same. What may seem baffling is that Java lets you specify `pa.x` to access the argument object's `private` instance field. Also, `main()` is able to access `x` directly, via the `pa2` object.

We previously presented Java's four access-control levels and the following statement about private access control: "A field, method, or constructor that is declared `private` cannot be accessed from beyond the class in which it's declared." When you carefully consider this statement and examine Listing 3-24, you'll realize that `x` isn't being accessed from beyond the `PrivateAccess` class in which it's declared. Therefore, the `private` access-control level isn't being violated.

The only code that can access this `private` instance field is code located within the `PrivateAccess` class. If you attempted to access `x` via a `PrivateAccess` object that was created in the context of another class, the compiler would report an error.

Being able to access `x` directly from within `PrivateAccess` is a performance enhancement; it's faster to access this implementation detail directly than to call a method that returns its value.

Compile `PrivateAccess.java` as follows:

```
javac PrivateAccess.java
```

Run the application as follows:

```
java PrivateAccess
```

You should observe the following output:

```
pa1 equal to pa2: false
pa2 equal to pa3: false
pa1 equal to pa3: true
20
```

Tip Get into the habit of developing useful interfaces while hiding implementations because it will save you a lot of trouble when maintaining your classes.

Initializing Classes and Objects

Classes and objects need to be properly initialized before they are used. You've already learned that class fields are initialized to default zero values after a class loads and can be subsequently initialized by assigning values to them in their declarations via class field initializers, for example, `static int counter = 1;`. Similarly, instance fields are initialized to default values when an object's memory is allocated via `new` and can be subsequently initialized by assigning values to them in their declarations via instance field initializers, for example, `int numDoors = 4;`.

Another aspect of initialization that's already been discussed is the constructor, which is used to initialize an object, typically by assigning values to various instance fields, but is also capable of executing arbitrary code such as code that opens a file and reads the file's contents.

Java provides two additional initialization features: class initializers and instance initializers. These features will be discussed in the following sections.

Class Initializers

Constructors perform initialization tasks for objects. Their counterpart from a class initialization perspective is the class initializer.

A *class initializer* is a static-prefixed block that's introduced into a class body. As an example, see Listing 3-25.

Listing 3-25. Class Initialization Example

```
public class A {
    static private int invoiceNumberBase;

    static {
        invoiceNumberBase = ...; // determine somehow
        if(invoiceNumberBase < 10000) {
            System.err.println("Invalid invoice number");
        }
    }
    //...
}
```

A class can declare a mix of class initializers and class field initializers, as demonstrated in Listing 3-26.

Listing 3-26. Mixing Class Initializers with Class Field Initializers

```
public class C {
    static {
        System.out.println("class initializer 1");
    }
}
```

```

static int counter = 1;

static {
    System.out.println("class initializer 2");
    System.out.println("counter = " + counter);
}

}

```

Listing 3-26 declares a class named C that specifies two class initializers and one class field initializer.

When class C is loaded into memory, the following output gets generated:

```

class initializer 1
class initializer 2
counter = 1

```

Instance Initializers

Not all classes can have constructors, as you'll discover in Chapter 5 when we present anonymous classes. For these classes, Java offers the instance initializer to handle instance initialization tasks.

An *instance initializer* is a block that's introduced into a class body as opposed to being introduced as the body of a method or a constructor. The instance initializer is used to initialize an object via a sequence of statements, as demonstrated in Listing 3-27.

Listing 3-27. Initializing a Pair of Arrays via an Instance Initializer

```

public class Graphics {
    private double[] sines;
    private double[] cosines;

    {
        sines = new double[360];
        cosines = new double[sines.length];
    }
}

```

```

        for (int degree = 0; degree < sines.length; degree++) {
            sines[degree] = Math.sin(Math.toRadians(degree));
            cosines[degree] = Math.cos(Math.toRadians(degree));
        }
    }
}

```

Listing 3-27's `Graphics` class uses an instance initializer to create an object's `sines` and `cosines` arrays and to initialize these arrays' elements to the sines and cosines of angles ranging from 0 through 359 degrees. It does so because it's faster to read array elements than to repeatedly call `Math.sin()` and `Math.cos()` elsewhere; performance matters. (In Chapter 7 we introduce `Math`.)

A class can declare a mix of instance initializers and instance field initializers, as shown in Listing 3-28.

Listing 3-28. Mixing Instance Initializers with Instance Field Initializers

```

public class C {
{
    System.out.println("instance initializer 1");
}

int counter = 1;

{
    System.out.println("instance initializer 2");
    System.out.println("counter = " + counter);
}
}

```

Listing 3-28 declares a class named `C` that specifies two instance initializers and one instance field initializer.

When `new C()` executes, the following output gets generated:

```

instance initializer 1
instance initializer 2
counter = 1

```

Note You should rarely need to use the instance initializer, which isn't commonly used in industry. Other developers would likely miss the instance initializer while scanning the source code and might find it confusing.

Initialization Order

A class's body can contain a mixture of class field initializers, class initializers, instance field initializers, instance initializers, and constructors. (You should prefer constructors to instance field initializers, although we're guilty of not doing so consistently, and restrict your use of instance initializers to anonymous classes, discussed in Chapter 5.) Furthermore, class fields and instance fields initialize to default values. Understanding the order in which all of this initialization occurs is necessary to preventing confusion.

The general rules are:

- Class fields initialize to default or explicit values just after a class is loaded. Immediately after a class loads, all class fields are zeroed to default values.
- Instance fields initialize to default or explicit values during object creation. When new allocates memory for an object, it zeros all instance fields to default values.

Additionally, because initialization occurs in a top-down manner, attempting to access the contents of a class field before that field is declared or attempting to access the contents of an instance field before that field is declared causes the compiler to report an *illegal forward reference*.

Collecting Garbage

Objects are created via reserved word new, but how are they destroyed? Without some way to destroy objects, they will eventually fill up the heap's available space and the application will not be able to continue. Java doesn't provide the developer with the ability to remove them from memory. Instead, Java handles this task by providing a *garbage collector*, which is code that runs in the background and occasionally checks for unreferenced objects. When the garbage collector discovers an unreferenced object, it removes the object from the heap, making more heap space available.

An *unreferenced object* is an object that cannot be accessed from anywhere within an application. For example, `new Employee("John", "Doe");` is an unreferenced object because the `Employee` reference returned by `new` is thrown away. In contrast, a *referenced object* is an object where the application stores at least one reference. For example, `Employee emp = new Employee("John", "Doe");` is a referenced object because variable `emp` contains a reference to the `Employee` object.

A referenced object becomes unreferenced when the application removes its last stored reference. For example, if `emp` is a local variable that contains the only reference to an `Employee` object, this object becomes unreferenced when the method in which `emp` is declared returns. An application can also remove a stored reference by assigning `null` to its reference variable. For example, `emp = null;` removes the reference to the `Employee` object that was previously stored in `emp`.

Garbage collection inside Java is an own science—a lot of effort has been spent in the last decades to make garbage collection fast and efficient. The Internet reveals garbage collector history and methodology details in case you are interested.

Revisiting Arrays

In Chapter 2, we introduced you to *arrays*, which are regions of memory (specifically, the heap) that store values in equal-size and contiguous slots, known as *elements*. We also presented several examples, including the following:

```
char[] gradeLetters = { 'A', 'B', 'C', 'D', 'F' };
```

Here you have an array variable named `gradeLetters` that stores a reference to a five-element region of memory, which stores the characters A, B, C, D, and F in contiguous and equal-size (16-bit) memory locations.

You access an element by specifying `gradeLetters[x]`, where `x` is an integer that identifies an array element and is known as an *index*; the first array element is always located at index 0. The following example shows you how to output and change the first element's value:

```
System.out.println(gradeLetters[0]); // Output the first grade letter.  
gradeLetters[0] = 'a'; // Perhaps you prefer lowercase grade letters.
```

The { 'A', 'B', 'C', 'D', 'F' } array-creation syntax is an example of *syntactic sugar* (syntax that simplifies a language, making it “sweeter” to use). Behind the scenes, the array is created with the new operator and initializes to these values, as follows:

```
char gradeLetters[] = new char[] { 'A', 'B', 'C', 'D', 'F' };
```

First, a five-character region of memory is allocated. Next, the region’s five-character elements are initialized to A, B, C, D, and F. Finally, a reference to these elements is stored in array variable gradeLetters.

Caution It’s an error to place an integer value between the square brackets following char. For example, the compiler reports an error when it encounters the 5 in new char[5] { 'A', 'B', 'C', 'D', 'F' };

You can think of an array as a special kind of object, although it’s not an object in the same sense that a class instance is an object. This pseudo-object has a solitary and read-only length field that contains the array’s size (the number of elements). For example, gradeLetters.length returns the number of elements (5) in the gradeLetters array.

Although you can use either of the previous two approaches to create an array, you will often specify a third approach that doesn’t involve explicit element initialization and subsequently initialize the array. This approach is demonstrated by the following code:

```
char[] gradeLetters = new char[5];
```

You specify the number of elements as a positive integer between the square brackets. Operator new zeros the bits in each array element’s storage location, which you interpret at the source code level as literal value false, '\u0000', 0, 0L, 0.0, 0.0F, or null (depending on element type).

You can then initialize the array, as follows:

```
gradeLetters[0] = 'A';
gradeLetters[1] = 'B';
gradeLetters[2] = 'C';
gradeLetters[3] = 'D';
gradeLetters[4] = 'F';
```

However, you will probably find it more convenient to use a loop for this task, as follows:

```
for (int i = 0; i < gradeLetters.length; i++)
    gradeLetters[i] = 'A' + i;
```

The previous examples focused on creating an array whose values share a common primitive type (character, represented by the `char` keyword). You can also create an array of object references. For example, you can create an array to store three `Image` object references, as follows:

```
Image[] imArray = { new Image("image0.png"),
new Image("image1.png"), new Image("image2.png") };
```

Here you have an array variable named `imArray` that stores a reference to a three-element region of memory, where each element stores a reference to an `Image` object. The `Image` object is located elsewhere in memory.

You access an `Image` element by specifying `imArray[x]`. The following example assumes the existence of a `getLength()` method that returns the image's length (in bytes) and calls this method on the first `Image` object to return the first image's length, which is subsequently output:

```
System.out.println(imArray[0].getLength());
```

As with the previous `gradeLetters` example, you can combine the `new` operator with the syntactic sugar initializer, as follows:

```
Image[] imArray = new Image[] { new Image("image0.png"),
new Image("image1.png"),
    new Image("image2.png") };
```

Finally, you can use the third approach, which initializes each object reference to the null reference by setting all of the bits in each element to 0. This approach is demonstrated as follows:

```
Image[] imArray = new Image[3];
```

Because `new` initializes each element to the null reference, you must explicitly initialize this array, and you can conveniently do so as follows:

```
for (int i = 0; i < imArray.length; i++)
    imArray[i] = new Image("image" + i + ".png"); // image0.png, image1.png,
    and so on
```

The "`image`" + `i` + ".`png`" expression uses the string concatenation operator (+) to combine `image` with the string equivalent of the integer value stored in variable `i` with ".`png`. The resulting string is passed to `Image`'s `Image(String filename)` constructor, and the resulting reference is stored in one of the array elements.

Note Use of the string concatenation operator in a loop context can result in a lot of unnecessary `String` object creation, depending on the length of the loop. We will discuss this topic in Chapter 7 when we introduce you to the `String` class.

The previous examples have focused on creating *one-dimensional arrays*. However, you can also create *multidimensional arrays* (i.e., arrays with two or more dimensions). For example, consider a two-dimensional array of temperature values.

Although you can use any of the three approaches to create the `temperatures` array, the third approach is preferable when the values vary greatly. The following example creates this array as a three-row-by-two-column table of double-precision floating-point temperature values:

```
double[][] temperatures = new double[3][2];
```

Notice the two sets of square brackets between `double` and `temperatures`. These two sets of brackets signify the array as two-dimensional (a table). Also notice the two sets of square brackets following `new` and `double`. Each set contains a positive integer value signifying the number of rows (3) or the number of columns (2) for each row.

Note When creating a multidimensional array, the number of square bracket pairs that are associated with the array variable and the number of square bracket pairs that follow `new` and the type name must be the same.

After creating the array, you can populate its elements with suitable values. The following example initializes each `temperatures` element, which is accessed as `temperatures[row][col]`, to a randomly generated temperature value via `Math.random()`, which we'll explain in Chapter 7:

```
for (int row = 0; row < temperatures.length; row++)
    for (int col = 0; col < temperatures[row].length; col++)
        temperatures[row][col] = Math.random() * 100;
```

The outer `for` loop selects each row from row 0 to the length of the array (which identifies the number of rows in the array). The inner `for` loop selects each column from 0 to the length of the current row array (which identifies the number of columns represented by that array). In essence, you're looking at a one-dimensional row array where each element references a one-dimensional column array.

You can subsequently output these values in a tabular format by using another `for` loop, as demonstrated by the following example where the code makes no attempt to align the temperature values in perfect columns:

```
for (int row = 0; row < temperatures.length; row++) {
    for (int col = 0; col < temperatures[row].length; col++)
        System.out.print(temperatures[row][col] + " ");
    System.out.println();
}
```

Java provides an alternative for creating a multidimensional array in which you create each dimension separately. For example, to create the previous two-dimensional `temperatures` array via `new` in this manner, first create a one-dimensional row array (the outer array), and then create a one-dimensional column array (the inner array), like so:

```
// Create the row array.
double[][] temperatures = new double[3][]; // Note the extra empty pair of brackets.

// Create a column array for each row.
for (int row = 0; row < temperatures.length; row++)
    temperatures[row] = new double[2]; // 2 columns per row
```

When you specify a different number of columns for each row, the resulting array is known as a *ragged array* because the array isn't rectangular.

Note When creating the row array, you must specify an extra pair of empty brackets as part of the expression following new. (For a three-dimensional array—a one-dimensional array of tables, where this array's elements reference row arrays—you must specify two pairs of empty brackets as part of the expression following new.)

EXERCISES

The following exercises are designed to test your understanding of Chapter 3’s content:

1. What is a class?
2. How do you declare a class?
3. True or false: You can declare multiple public classes in a source file.
4. What is an object?
5. How do you obtain an object?
6. What is a constructor?
7. True or false: Java creates a default noargument constructor when a class declares no constructors.
8. What is a parameter list and what is a parameter?
9. What is an argument list and what is an argument?
10. True or false: You invoke another constructor by specifying the name of the class followed by an argument list.
11. Define arity.
12. What is a local variable?
13. Define lifetime.
14. Define scope.
15. What is encapsulation?
16. Define field.

17. What is the difference between an instance field and a class field?
18. What is a blank final, and how does it differ from a true constant?
19. How do you prevent a field from being shadowed?
20. Define method.
21. What is the difference between an instance method and a class method?
22. Define recursion.
23. How do you overload a method?
24. What is a class initializer and an instance initializer?
25. Define garbage collector.
26. True or false: `String[] letters = new String[2] { "A", "B" };` is correct syntax.
27. What is a ragged array?
28. Merge Listings 3-3 and 3-4 into a complete Image application that demonstrates its constructors.
29. Create a `Conversions` class with `c2f()` and `f2c()` class methods that convert their double arguments to degrees Fahrenheit or degrees Celsius. Introduce a `main()` method into this class to test these methods.
30. Create a `Utilities` class that incorporates the previous `factorial()` methods (using iteration and recursion) along with the `sum()` method that used the variable arguments feature to sum a variable number of arguments. Modify the recursive `factorial()` method to also handle the case of $0!$, which equals 1. Introduce a `main()` method into this class that demonstrates these methods.
31. The `factorial()` method provides an example of *tail recursion*, a special case of recursion in which the method's last statement contains a recursive call, which is known as a *tail call*. Provide another example of tail recursion in which you create a GCD application whose `static int gcd(int a, int b)` class method returns the highest integer that divides evenly into both arguments. In other words, this method returns the *greatest common divisor* of the integer arguments passed to `a` and `b`.

32. Create a Book class with private name, author, and International Standard Book Number (ISBN) fields. Provide a suitable constructor and getter methods that return field values. Introduce a `main()` method into this class that creates an array of Book objects and iterates over this array outputting each book's name, author, and ISBN.
-

Summary

A class is a container for housing an application, and it is also a template for manufacturing objects. You declare a class by minimally specifying reserved word `class` followed by a name that identifies the class (so that it can be referred to from elsewhere in the source code), followed by a body. The body starts with an open brace character (`{`) and ends with a close brace (`}`). Sandwiched between these delimiters are various kinds of member declarations.

Objects are instances of classes. You create objects by using the `new` operator to allocate memory and a constructor to initialize the object. A constructor is a block of code that's declared in a class for constructing an object from that class by initializing it in some manner. The `new` operator returns a reference to the newly created and initialized object.

A constructor doesn't have an own name. Instead, you must specify the name of the class that declares the constructor. This name is followed by a round bracket-delimited parameter list, which is a comma-separated list of zero or more parameter declarations. A parameter is a constructor or method variable that receives an expression value passed to the constructor or method when it's called. This expression value is known as an argument.

When a class doesn't declare a constructor, Java implicitly creates a constructor for that class. The created constructor is known as the default noargument constructor because no arguments appear between its `(` and `)` characters when the constructor is invoked. The default noargument constructor isn't created when at least one constructor is declared in the class.

Classes typically combine state with behaviors. State refers to attributes that are read and/or written when an application runs, and behaviors refer to sequences of code that read/write attributes and perform other tasks. Combining state with corresponding behaviors is known as encapsulation.

Java represents state via fields, which are variables declared within a class's body. State associated with a class is described by class fields, whereas state associated with objects is described by object fields (also known as instance fields).

Java represents behaviors via methods, which are named blocks of code declared within a class's body. Behaviors associated with a class are described by class methods, whereas behaviors associated with objects are described by object methods (also known as instance methods).

Every class exposes an interface, which are the constructors, methods, and (possibly) fields that can be accessed from outside of the class. An interface serves as a contract between a class and its clients, which are external classes that communicate with the class and/or its instances by accessing fields and calling constructors and methods. The contract is such that the class promises to not change its interface, which would break dependent clients.

The class also provides an implementation (the code within exposed methods along with optional helper methods and optional supporting fields that shouldn't be exposed) that codifies the interface. Helper methods assist exposed methods and shouldn't be exposed.

When designing a class, your goal is to expose a useful interface while hiding details of that interface's implementation. You hide the implementation to prevent developers from accidentally accessing parts of your class that don't belong to the class's interface so that you're free to change the implementation without breaking client code. Hiding the implementation is often referred to as information hiding. Furthermore, many developers consider implementation hiding to be part of encapsulation.

Java supports implementation hiding by providing four levels of access control, where three of these levels are indicated via a reserved word: `public`, `private`, and `protected`. You can use these access-control levels to control access to fields, methods, and constructors and two of these levels to control access to classes.

Classes and objects need to be properly initialized before they're used. You've already learned that class fields are initialized to default zero values after a class loads and can be subsequently initialized by assigning values to them in their declarations via class field initializers. Similarly, instance fields are initialized to default values when an object's memory is allocated via `new` and can be subsequently initialized by assigning values to them in their declarations via instance field initializers or via constructors.

Java also supports class initializers and instance initializers for this task. A class initializer is a `static`-prefixed block that's introduced into a class body. It's used to initialize a loaded class via a sequence of statements. An instance initializer is a block that's introduced into a class body as opposed to being introduced as the body of a method or a constructor. The instance initializer is used to initialize an object via a sequence of statements.

Objects are created via reserved word `new`, but how are they destroyed? Without some way to destroy objects, they will eventually fill up the heap's available space and the application will not be able to continue. Java doesn't provide the developer with the ability to remove them from memory. Instead, Java handles this task by providing a garbage collector, which is code that runs in the background and occasionally checks for unreferenced objects.

You can think of an array as a special kind of object, although it's not an object in the same sense that a class instance is an object. This pseudo-object has a solitary and read-only `length` field that contains the array's size (the number of elements). In addition to using the syntactic sugar first presented in Chapter 2 for creating an array, you can also create an array using the `new` operator, with or without the syntactic sugar.

Chapter 4 continues to explore the Java language by examining its support for inheritance, polymorphism, and interfaces.

CHAPTER 4

Discovering Inheritance, Polymorphism, and Interfaces

An *object-based language* is a language that encapsulates state and behaviors in objects. Java's support for encapsulation (discussed in Chapter 3) qualifies it as an object-based language. However, Java is also an *object-oriented language* because it supports inheritance and polymorphism (as well as encapsulation). (Object-oriented languages are a subset of object-based languages.) In this chapter, we will introduce you to Java's language features that support inheritance and polymorphism. Also, we will introduce you to interfaces, Java's ultimate abstract type mechanism.

Building Class Hierarchies

We tend to categorize stuff by saying things like “cars are vehicles” or “savings accounts are bank accounts.” By making these statements, we really are saying (from a software development perspective) that cars inherit vehicular state (such as make and color) and behaviors (such as park and display mileage) and that savings accounts inherit bank account state (such as balance) and behaviors (such as deposit and withdraw). Car, vehicle, savings account, and bank account are examples of real-world entity categories, and *inheritance* is a hierarchical relationship between similar entity categories in which one category inherits state and behaviors from at least one other entity category. Inheriting from a single category is *single inheritance*, and inheriting from at least two categories is *multiple inheritance*.

Java supports single inheritance in a class context to facilitate code reuse—why reinvent the wheel? In this context, a class inherits state and behaviors from another

class through class extension. Because classes are involved, Java refers to this kind of inheritance as *implementation inheritance*.

Contrary to the class-related single inheritance, Java supports multiple inheritance in an interface context in which a class inherits behavior templates from one or more interfaces through interface implementation or in which an interface inherits behavior templates from one or more interfaces through interface extension. Because interfaces are involved, Java refers to this kind of inheritance as *interface inheritance*. (We discuss interfaces later in this chapter.)

Note You reuse code by carefully extending classes, implementing interfaces, and extending interfaces. You start with something that is close to what you want and then you extend it to meet your goal. You don't reuse code by simply copying and pasting it. Copying and pasting often results in redundant (i.e., nonreusable) and buggy code.

In the next section, we will introduce you to Java's support for implementation inheritance by first focusing on class extension. We will then introduce you to a special class that sits at the top of Java's class hierarchy. After introducing you to composition, which is an alternative to implementation inheritance for reusing code, we will show you how composition can be used to overcome problems with implementation inheritance.

Extending Classes

Java provides the reserved word `extends` for specifying a hierarchical relationship between two classes. For example, suppose you have a `Vehicle` class and want to introduce a `Car` class as a kind of `Vehicle`. Listing 4-1 uses `extends` to cement this relationship.

Throughout the chapter we present listings with one or more classes. If a listing contains more than one public class, you have to separate the classes into several accordingly named Java files.

Listing 4-1. Relating Two Classes via extends

```
public class Vehicle {
    // member declarations
}

public class Car extends Vehicle {
    // member declarations
}
```

Listing 4-1 codifies a relationship that is known as an “is-a” relationship: a car is a kind of vehicle. In this relationship, `Vehicle` is known as the *base class*, *parent class*, or *superclass*; and `Car` is known as the *derived class*, *child class*, or *subclass*.

Note When you don’t want anyone to extend one of your classes (for security or another reason), you must declare that class `final`. For example, if you specified `final class Vehicle {}`, the compiler would report an error upon encountering `class Car extends Vehicle {}`.

As well as being capable of providing its own member declarations, `Car` is capable of inheriting member declarations from its `Vehicle` superclass. As Listing 4-2 shows, non-private inherited members become accessible to members of the `Car` class.

Listing 4-2. Inheriting Members

```
public class Vehicle {
    private String make;
    private String model;
    private int year;

    public Vehicle(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }
}
```

```

public String getMake() {
    return make;
}

public String getModel() {
    return model;
}

public int getYear() {
    return year;
}

public class Car extends Vehicle {
    private int numWheels;

    public Car(String make, String model, int year, int numWheels) {
        super(make, model, year);
        this.numWheels = numWheels;
    }

    public static void main(String[] args) {
        Car car = new Car("Ford", "Fiesta", 2009, 4);
        System.out.println("Make = " + car.getMake());
        System.out.println("Model = " + car.getModel());
        System.out.println("Year = " + car.getYear());
        // Normally, you cannot access a private field via an object
        // reference. However, numWheels is being accessed from
        // within a method (main()) that is part of the Car class.
        System.out.println("Number of wheels = " + car.numWheels);
    }
}

```

Listing 4-2's `Vehicle` class declares private fields that store a vehicle's make, model, and year; a constructor that initializes these fields to passed arguments; and getter methods that retrieve these fields' values.

The `Car` subclass provides a private `numWheels` field, a constructor that initializes a `Car` object's `Vehicle` and `Car` layers, and a `main()` class method for testing this class.

Car's constructor uses reserved word `super` to call Vehicle's constructor with Vehicle-oriented arguments and then initializes Car's `numWheels` instance field. The `super()` call is analogous to specifying `this()` to call another constructor in the same class, but invokes a superclass constructor instead.

Caution The `super()` call can only appear in a constructor. Furthermore, it must be the first code that is specified in the constructor. If `super()` is not specified, and if the superclass does not have a noargument constructor, the compiler will report an error because the subclass constructor must call a noargument superclass constructor when `super()` is not present.

Car's `main()` method creates a Car object, initializing this object to a specific make, model, year, and number of wheels. Four `System.out.println()` method calls subsequently output this information.

The first three `System.out.println()` method calls retrieve their pieces of information by calling the Car instance's inherited `getMake()`, `getModel()`, and `getYear()` methods. The final `System.out.println()` method call accesses the instance's `numWheels` field directly. Because Car is declared to be a public class, Listing 4-2 would be stored in a file named `Car.java`. Therefore, execute `javac Car.java` to compile this source code into `Vehicle.class` and `Car.class`. Then execute `java Car` to test the Car class. This execution results in the following output:

```
Make = Ford
Model = Fiesta
Year = 2009
Number of wheels = 4
```

Note A class whose instances cannot be modified (its fields cannot be modified) is known as an *immutable class*. Vehicle is an example. If Car's `main()` method, which can directly read or write `numWheels`, was not present, Car would also be an example of an immutable class. Also, a class cannot inherit constructors, nor can it inherit private fields and methods. For example, Car doesn't inherit Vehicle's constructor, nor does it inherit Vehicle's private `make`, `model`, and `year` fields.

A subclass can *override* (replace) an inherited method so that the subclass's version of the method is called instead. Listing 4-3 shows you that the overriding method must specify the same name, parameter list, and return type as the method being overridden.

Listing 4-3. Overriding a Method

```
public class Vehicle {  
    private String make;  
    private String model;  
    private int year;  
  
    public Vehicle(String make, String model, int year) {  
        this.make = make;  
        this.model = model;  
        this.year = year;  
    }  
  
    public void describe() {  
        System.out.println(year + " " + make + " " + model);  
    }  
}  
  
public class Car extends Vehicle {  
    private int numWheels;  
  
    public Car(String make, String model, int year, int numWheels) {  
        super(make, model, year);  
    }  
  
    public void describe() {  
        System.out.print("This car is a ");  
        // Print without newline—see Chapter 1.  
        super.describe();  
    }  
  
    public static void main(String[] args) {  
        Car car = new Car("Ford", "Fiesta", 2009, 4);  
        car.describe();  
    }  
}
```

Listing 4-3's Car class declares a `describe()` method that overrides Vehicle's `describe()` method to output a car-oriented description. After outputting an initial message, this method uses reserved word `super` to call Vehicle's `describe()` method via `super.describe();`.

Note Call a superclass method from the overriding subclass method by prefixing the method's name with reserved word `super` and the member access operator. If you don't do this, you end up recursively calling the subclass's overriding method. Use `super` and the member access operator to access non-private superclass fields from subclasses that mask these fields by declaring same-named fields.

If you were to compile Listing 4-3 (`javac Car.java`) and run the Car application (`java Car`), you would discover that Car's overriding `describe()` method executes instead of Vehicle's overridden `describe()` method and outputs `This car is a 2009 Ford Fiesta.`

Note When you don't want anyone to extend one of your methods (for security or another reason), you must declare that method `final`. For example, if you specified `final public void describe()` for Vehicle's `describe()` method, the compiler would report an error upon encountering an attempt to override this method in the Car class. Also, you cannot make an overriding method less accessible than the method it overrides. For example, if Car's `describe()` method was declared as `private void describe()`, the compiler would report an error because private access is less accessible than the default package access. However, a method could be made more accessible by adding `protected` or `public` to an overridden, originally `private` method.

Suppose you happened to replace Listing 4-3's `describe()` method with the method shown here:

```
public void describe(String owner) {  
    System.out.print("This car, which is owned by " + owner + ", is a ");  
    super.describe();  
}
```

The modified Car class now has two `describe()` methods, the preceding explicitly declared method and the method inherited from Vehicle. The `void describe(String owner)` method doesn't override Vehicle's `describe()` method. Instead, it overloads this method.

The Java compiler helps you detect an attempt to overload instead of override a method at compile time by letting you prefix a subclass's method header with the `@Override` annotation, as shown here (we will discuss annotations in Chapter 6):

```
@Override  
public void describe() {  
    System.out.print("This car is a ");  
    super.describe();  
}
```

Specifying `@Override` tells the compiler that the method overrides another method. If you overload the method instead, the compiler reports an error. Without this annotation, the compiler would not report an error because method overloading is a valid feature.

Tip Get into the habit of prefixing overriding methods with the `@Override` annotation. This habit will help you detect overloading mistakes much sooner.

In Chapter 3, we discussed the initialization order of classes and objects, where you learned that class members are always initialized first and in a top-down order (the same order applies to instance members). Implementation inheritance adds a couple more details:

- A superclass's class initializers always execute before a subclass's class initializers.
- A subclass's constructor always calls the superclass constructor to initialize an object's superclass layer and then initializes the subclass layer.

Java's support for implementation inheritance only permits you to extend a single class. You cannot extend multiple classes because doing so can lead to problems. For example, suppose Java supported multiple implementation inheritance, and you decided to model a flying horse (from Greek mythology) via the class structure shown in Listing 4-4.

Listing 4-4. A Fictional Demonstration of Multiple Implementation Inheritance

```

public class Bird {
    public void describe() {
        // code that outputs a description of a bird's appearance and
        behaviors
    }
}

public class Horse {
    public void describe() {
        // code that outputs a description of a horse's appearance and
        behaviors
    }
}

public class FlyingHorse extends Bird, Horse {
    public static void main(String[] args) {
        FlyingHorse pegasus = new FlyingHorse();
        pegasus.describe();
    }
}

```

Listing 4-4’s class structure reveals an ambiguity resulting from each of `Bird` and `Horse` declaring a `describe()` method. Which of these methods does `FlyingHorse` inherit? A related ambiguity arises from same-named fields, possibly of different types. Which field is inherited?

The Ultimate Superclass

A class that doesn’t explicitly extend another class implicitly extends Java’s `Object` class (located in the `java.lang` package—we will discuss packages in the next chapter). For example, Listing 4-1’s `Vehicle` class extends `Object`, whereas `Car` extends `Vehicle`.

`Object` is Java’s ultimate superclass because it serves as the ancestor of every other class but doesn’t itself extend any other class. `Object` provides a common set of methods that other classes inherit. Table 4-1 describes these methods.

Table 4-1. *Object's Methods*

Method	Description
<code>Object clone()</code>	Create and return a copy of the current object.
<code>boolean equals(Object obj)</code>	Determine if the current object is equal to the object identified by <code>obj</code> .
<code>void finalize()</code>	Finalize the current object.
<code>Class<?> getClass()</code>	Return the current object's <code>java.lang.Class</code> object.
<code>int hashCode()</code>	Return the current object's hash code.
<code>void notify()</code>	Wake up one of the threads that are waiting on the current object's monitor.
<code>void notifyAll()</code>	Wake up all threads that are waiting on the current object's monitor.
<code>String toString()</code>	Return a string representation of the current object.
<code>void wait()</code>	Cause the current thread to wait on the current object's monitor until it is woken up via <code>notify()</code> or <code>notifyAll()</code> .
<code>void wait(long timeout)</code>	Cause the current thread to wait on the current object's monitor until it is woken up via <code>notify()</code> or <code>notifyAll()</code> or until the specified <code>timeout</code> value (in milliseconds) has elapsed, whichever comes first.
<code>void wait(long timeout, int nanos)</code>	Cause the current thread to wait on the current object's monitor until it is woken up via <code>notify()</code> or <code>notifyAll()</code> or until the specified <code>timeout</code> value (in milliseconds) plus <code>nanos</code> value (in nanoseconds) has elapsed, whichever comes first.

We will discuss the `clone()`, `equals()`, `finalize()`, `hashCode()`, and `toString()` methods shortly, but will defer a discussion of `notify()`, `notifyAll()`, and the `wait()` methods until Chapter 7, and defer a discussion of the `getClass()` method until Chapter 8.

Cloning

The `clone()` method *clones* (duplicates) an object without calling a constructor. It copies each primitive or reference field's value to its counterpart in the clone, a task known as *shallow copying* or *shallow cloning*. Listing 4-5 demonstrates this behavior.

Listing 4-5. Shallowly Cloning an Employee Object

```

public class Employee implements Cloneable {
    String name;
    int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public static void main(String[] args) throws CloneNotSupportedException
    {
        Employee e1 = new Employee("John Doe", 46);
        Employee e2 = (Employee) e1.clone();
        System.out.println(e1 == e2); // Output: false
        System.out.println(e1.name == e2.name); // Output: true
    }
}

```

Listing 4-5 declares an `Employee` class with `name` and `age` instance fields and a constructor for initializing these fields. The `main()` method uses this constructor to initialize a new `Employee` object's copies of these fields to John Doe and 46.

Note A class must implement the `java.lang.Cloneable` interface or its instances cannot be shallowly cloned via `Object`'s `clone()` method—this method performs a runtime check to see if the class implements `Cloneable`. (We will discuss interfaces later in this chapter.) If a class doesn't implement `Cloneable`, `clone()` throws `java.lang.CloneNotSupportedException`. (Because `CloneNotSupportedException` is a checked exception, it's necessary for Listing 4-5 to satisfy the compiler by appending `throws CloneNotSupportedException` to the `main()` method's header. We will discuss exceptions in the next chapter.) The `java.lang.String` class is an example of a class that doesn't implement `Cloneable`; hence, `String` objects cannot be shallowly cloned.

After assigning the `Employee` object's reference to local variable `e1`, `main()` calls the `clone()` method on this variable to duplicate the object and then assigns the resulting reference to variable `e2`. The `(Employee)` cast is needed because `clone()` returns `Object`.

To prove that the objects whose references were assigned to `e1` and `e2` are different, `main()` next compares these references via `==` and outputs the Boolean result, which happens to be false. To prove that the `Employee` object was shallowly cloned, `main()` next compares the references in both `Employee` objects' `name` fields via `==` and outputs the Boolean result, which happens to be true.

Shallow cloning is not always desirable because the original object and its clone refer to the same object via their equivalent reference fields. For example, each of Listing 4-5's two `Employee` objects refers to the same `String` object via its `name` field.

Equality

The `==` and `!=` operators compare two primitive values (such as integers) for equality (`==`) or inequality (`!=`). These operators also compare two references to see whether they refer to the same object or not. This latter comparison is known as an *identity check*.

You cannot use `==` and `!=` to determine whether two objects are logically the same (or not). For example, two `Car` objects with the same field values are logically equivalent. However, `==` reports them as unequal because of their different references.

Recognizing the need to support logical equality in addition to reference equality, Java provides an `equals()` method in the `Object` class. Because this method defaults to comparing references, you need to override `equals()` to compare object contents.

Before overriding `equals()`, make sure that this is necessary. For example, Java's `java.lang.StringBuffer` class doesn't override `equals()`. Perhaps this class's designers didn't think it necessary to determine whether two `StringBuffer` objects are logically equivalent or not.

You cannot override `equals()` with arbitrary code. Doing so will probably prove disastrous to your applications. Instead, you need to adhere to the contract that is specified in the Java documentation for this method, which we present next.

The `equals()` method implements an equivalence relation on nonnull object references:

- *It is reflexive:* For any nonnull reference value `x`, `x.equals(x)` must return true.

- *It is symmetric:* For any nonnull reference values x and y , $x.equals(y)$ must return true if and only if $y.equals(x)$ returns true.
- *It is transitive:* For any nonnull reference values x , y , and z , if $x.equals(y)$ returns true and $y.equals(z)$ returns true, then $x.equals(z)$ must return true.
- *It is consistent:* For any nonnull reference values x and y , multiple invocations of $x.equals(y)$ consistently must return true or consistently must return false, provided no information used in $equals()$ comparisons on the objects is modified.
- For any nonnull reference value x , $x.equals(null)$ must return false.

Although this contract probably looks somewhat intimidating, it isn't that difficult to satisfy. For proof, take a look at the implementation of the `equals()` method in Listing 4-6's Point class.

Listing 4-6. Logically Comparing Point Objects

```
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
```

```

    Point p = (Point) o;
    return p.x == x && p.y == y;
}

public static void main(String[] args) {
    Point p1 = new Point(10, 20);
    Point p2 = new Point(20, 30);
    Point p3 = new Point(10, 20);
    // Test reflexivity.
    System.out.println(p1.equals(p1)); // Output: true
    // Test symmetry.
    System.out.println(p1.equals(p2)); // Output: false
    System.out.println(p2.equals(p1)); // Output: false
    // Test transitivity.
    System.out.println(p2.equals(p3)); // Output: false
    System.out.println(p1.equals(p3)); // Output: true
    // Test nullability.
    System.out.println(p1.equals(null)); // Output: false
    // Extra test to further prove the instanceof operator's usefulness.
    System.out.println(p1.equals("abc")); // Output: false
}
}

```

Listing 4-6’s overriding `equals()` method begins with an if statement that uses the `instanceof` operator to determine whether the argument passed to parameter `o` is an instance of the `Point` class. If not, the if statement executes `return false;`.

The `o instanceof Point` expression satisfies the last portion of the contract: for any nonnull reference value `x`, `x.equals(null)` returns false. Because the null reference is not an instance of any class, passing this value to `equals()` causes the expression to evaluate to false.

The `o instanceof Point` expression also prevents a `java.lang.ClassCastException` instance from being thrown via expression `(Point) o` in the event that you pass an object other than a `Point` object to `equals()`. (We will discuss exceptions in the next chapter.)

Following the cast, the contract’s reflexivity, symmetry, and transitivity requirements are met by only allowing `Points` to be compared with other `Points` via expression `p.x == x && p.y == y`.

The final contract requirement, consistency, is met by making sure that the `equals()` method is deterministic. In other words, this method doesn’t rely on any field value that could change from method call to method call.

Tip You can optimize the performance of a time-consuming `equals()` method by first using `==` to determine if `o`’s reference identifies the current object. Simply specify `if (o == this) return true;` as the `equals()` method’s first statement.

It’s important to always override the `hashCode()` method when overriding `equals()`. We didn’t do so in Listing 4-6 because we have yet to formally introduce `hashCode()`. See the following texts.

Finalization

Finalization refers to cleanup via the `finalize()` method, which is known as a *finalizer*. The `finalize()` method’s Java documentation states that `finalize()` is “called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the `finalize()` method to dispose of system resources or to perform other cleanup.”

`Object`’s version of `finalize()` does nothing; you must override this method with any needed cleanup code. Because the virtual machine might never call `finalize()` before an application terminates, your application design must not rely on it.

If you decide to override `finalize()`, your object’s subclass layer must give its superclass layer an opportunity to perform finalization. You can accomplish this task by specifying `super.finalize();` as the last statement in your method, which the following example demonstrates:

```
@Override
protected void finalize() throws Throwable {
    try {
        // Perform subclass cleanup.
    } finally {
```

```
    super.finalize();  
}  
}
```

The example's `finalize()` declaration appends `throws Throwable` to the method header because the cleanup code might throw an exception. If an exception is thrown, execution leaves the method and, in the absence of try-finally, `super.finalize();` never executes. (We will discuss exceptions and try-finally in Chapter 5.)

To guard against this possibility, the subclass's cleanup code executes in a block that follows reserved word `try`. If an exception is thrown, Java's exception-handling logic executes the block following the `finally` reserved word, and `super.finalize();` executes the superclass's `finalize()` method.

Hash Codes

The `hashCode()` method returns a 32-bit integer that identifies the current object's *hash code*, a single numeric value that results from applying a mathematical function to a potentially large amount of data. The calculation of this value is known as *hashing*.

You must override `hashCode()` when overriding `equals()` and in accordance with the following contract, which is specified in `hashCode()`'s Java documentation:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used in `equals(Object)` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects might improve the performance of hashtables.

Fail to obey this contract and your class's instances will not work properly with Java's hash-based Collections Framework classes, such as `java.util.HashMap`. (We will discuss `HashMap` and other Collections Framework classes in Chapter 9.)

If you override `equals()` but not `hashCode()`, you most importantly violate the second item in the contract: the hash codes of equal objects must also be equal. This violation can lead to serious consequences, as, for example, broken map lookup code using some key (see Chapter 9).

String Representation

The `toString()` method returns a string-based representation of the current object. This representation defaults to the object's class name, followed by the @ symbol, followed by a hexadecimal representation of the object's hash code.

For example, if you were to execute `System.out.println(p1);` to output Listing 4-6's `p1` object, you would see a line of output similar to `Point@3e25a5`. (`System.out.println()` calls `p1`'s inherited `toString()` method behind the scenes.)

You should strive to override `toString()` so that it returns a concise but meaningful description of the object. For example, you might declare, in Listing 4-6's `Point` class, a `toString()` method that is similar to the following:

```
@Override
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

This time, executing `System.out.println(p1);` results in more meaningful output, such as `(10, 20)`.

Composition

Implementation inheritance and composition offer two different approaches to reusing code. As you have learned, implementation inheritance is concerned with extending a class with a new class, which is based upon an “is-a” relationship between them: a `Car` is a `Vehicle`, for example.

On the other hand, *composition* is concerned with composing classes out of other classes, which is based upon a “has-a” relationship between them. For example, a `Car` has an `Engine`, `Wheels`, and a `SteeringWheel`.

You have already seen examples of composition in this chapter. For example, Listing 4-2's Car class includes String make and String model fields. Listing 4-7's Car class provides another example of composition.

*****Listing 4-7.*** A Car Class Whose Instances Are Composed of Other Objects**

```
public class Car extends Vehicle {
    private Engine engine; // bicycles don't have engines
    private Wheel[] wheels; // boats don't have wheels
    private SteeringWheel steeringWheel; // hang gliders don't have steering
                                         wheels
}
```

Listing 4-7 demonstrates that composition and implementation inheritance are not mutually exclusive. Although not shown, Car inherits various members from its Vehicle superclass, in addition to providing its own engine, wheels, and steeringWheel fields.

Changing Form

Some real-world entities can change their forms. For example, water (on Earth as opposed to interstellar space) is normally a liquid, but it changes to a solid when frozen, and it changes to a gas when heated to its boiling point. Insects such as butterflies that undergo metamorphosis are another example.

The ability to change form is known as *polymorphism* and is useful to model in a programming language. For example, code that draws arbitrary shapes can be expressed more concisely by introducing a single Shape class and its draw() method and by invoking that method for each Circle instance, Rectangle instance, and other Shape instance stored in an array. When Shape's draw() method is called for an array instance, it is the Circle's, Rectangle's, or other Shape instance's draw() method that gets called. There are many forms of Shape's draw() method. In other words, this method is polymorphic.

Java supports four kinds of polymorphism:

- *Coercion:* An operation serves multiple types through implicit type conversion. For example, division lets you divide an integer by another integer or divide a floating-point value by another floating-point value. If one operand is an integer and the other operand is a

floating-point value, the compiler *coerces* (implicitly converts) the integer to a floating-point value to prevent a type error. (There is no division operation that supports an integer operand and a floating-point operand.) Passing a subclass object reference to a method's superclass parameter is another example of coercion polymorphism. The compiler coerces the subclass type to the superclass type to restrict operations to those of the superclass.

- *Overloading:* The same operator symbol or method name can be used in different contexts. For example, + can be used to perform integer addition, floating-point addition, or string concatenation, depending on the types of its operands. Also, multiple methods having the same name can appear in a class (through declaration and/or inheritance).
- *Parametric:* Within a class declaration, a field name can associate with different types and a method name can associate with different parameter and return types. The field and method can then take on different types in each class instance. For example, a field might be of type `java.lang.Integer` and a method might return an `Integer` in one class instance, and the same field might be of type `String` and the same method might return a `String` in another class instance. Java supports parametric polymorphism via generics, which we will discuss in Chapter 6.
- *Subtype:* A type can serve as another type's subtype. When a subtype instance appears in a supertype context, executing a supertype operation on the subtype instance results in the subtype's version of that operation executing. For example, suppose that `Circle` is a subclass of `Point` and that both classes contain a `draw()` method. Assigning a `Circle` instance to a variable of type `Point`, and then calling the `draw()` method via this variable, results in `Circle`'s `draw()` method being called.

Many developers don't regard coercion and overloading as valid kinds of polymorphism. They see coercion and overloading as nothing more than type conversions and syntactic sugar. In contrast, parametric and subtype are regarded as valid kinds of polymorphism.

In this section, we focus on subtype polymorphism by first examining upcasting and late binding. We then introduce you to abstract classes and abstract methods, downcasting and runtime type identification, and covariant return types.

Upcasting and Late Binding

[Listing 4-6](#)'s Point class represents a point as an x-y pair. Because a circle (in this example) is an x-y pair denoting its center, and has a radius denoting its extent, you can extend Point with a Circle class that introduces a radius field. Check out [Listing 4-8](#).

Listing 4-8. A Circle Class Extending the Point Class

```
public class Circle extends Point {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }

    public int getRadius() {
        return radius;
    }

    @Override
    public String toString() {
        return "" + radius;
    }
}
```

[Listing 4-8](#)'s Circle class describes a Circle as a Point with a radius, which implies that you can treat a Circle instance as if it was a Point instance. Accomplish this task by assigning the Circle instance to a Point variable, as demonstrated here:

```
Circle c = new Circle(10, 20, 30);
Point p = c;
```

The cast operator isn't needed to convert from Circle to Point because access to a Circle instance via Point's interface is legal. After all, a Circle is at least a Point.

This assignment is known as *upcasting* because you are implicitly casting up the type hierarchy (from the `Circle` subclass to the `Point` superclass).

After upcasting `Circle` to `Point`, you cannot call `Circle`'s `getRadius()` method because this method is not part of `Point`'s interface. Losing access to subtype features after narrowing a subclass to its superclass seems useless but is necessary for achieving subtype polymorphism.

In addition to upcasting the subclass instance to a variable of the superclass type, subtype polymorphism involves declaring a method in the superclass and overriding this method in the subclass. For example, suppose `Point` and `Circle` are to be part of a graphics application, and you need to introduce a `draw()` method into each class to draw a point and a circle, respectively. You end with the class structure shown in Listing 4-9.

Listing 4-9. Declaring a Graphics Application's `Point` and `Circle` Classes

```
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }

    public void draw() {
        System.out.println("Point drawn at " + toString());
    }
}
```

```

public class Circle extends Point {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }

    public int getRadius() {
        return radius;
    }

    @Override
    public String toString() {
        return "" + radius;
    }

    @Override
    public void draw() {
        System.out.println("Circle drawn at " + super.toString() +
                           " with radius " + toString());
    }
}

```

Listing 4-9's `draw()` methods will ultimately draw graphics shapes, but simulating their behaviors via `System.out.println()` method calls is sufficient during the early testing phase of the graphics application.

Now that you have temporarily finished with `Point` and `Circle`, you will want to test their `draw()` methods in a simulated version of the graphics application. To achieve this objective, you write Listing 4-10's `Graphics` class.

Listing 4-10. A `Graphics` Class for Testing `Point`'s and `Circle`'s `draw()` Methods

```

public class Graphics {
    public static void main(String[] args) {
        Point[] points = new Point[] {
            new Point(10, 20), new Circle(10, 20, 30) };
    }
}

```

```

    for (int i = 0; i < points.length; i++)
        points[i].draw();
}
}

```

Listing 4-10's `main()` method first declares an array of `Points`. Upcasting is demonstrated by first having the array's initializer instantiate the `Circle` class and then by assigning this instance's reference to the second element in the `points` array.

Moving on, `main()` uses a for loop to call each `Point` element's `draw()` method. Because the first iteration calls `Point`'s `draw()` method, whereas the second iteration calls `Circle`'s `draw()` method, you observe the following output:

```

Point drawn at (10, 20)
Circle drawn at (10, 20) with radius 30

```

Java supposes that for the second iteration it must call `Circle`'s `draw()` method. Internally it does so by letting the compiler keep a hint to which method is to be called based on the object's original type. This procedure is called *late binding*.

You can also upcast from one array to another provided that the array being upcast is a subtype of the other array. Consider Listing 4-11.

Listing 4-11. Demonstrating Array Upcasting

```

public class Point {
    ...
}

public class ColoredPoint extends Point {
    ...
}

public class UpcastArrayDemo {
    public static void main(String[] args) {
        ColoredPoint[] cptArray = new ColoredPoint[] {
            new ColoredPoint(10, 20, 5)};
        Point[] ptArray = cptArray; // legal
    }
}

```

Listing 4-11's `main()` method first creates a `ColoredPoint` array consisting of one element. It then instantiates this class and assigns the object's reference to this element. Because `ColoredPoint[]` is a subtype of `Point[]`, `main()` is able to upcast `cptArray`'s `ColoredPoint[]` type to `Point[]` and assign its reference to `ptArray`.

Abstract Classes and Abstract Methods

Suppose new requirements dictate that your graphics application must include a `Rectangle` class. Furthermore, this class must include a `draw()` method, and this method must be tested in a manner similar to that shown in Listing 4-10's `Graphics` application class.

In contrast to `Circle`, which is a `Point` with a radius, it doesn't make sense to think of a `Rectangle` as being a `Point` with a width and height. Rather, a `Rectangle` instance would probably be composed of a `Point` instance indicating its origin and a `Point` instance indicating its width and height extents.

Because circles, points, and rectangles are examples of shapes, it makes sense to declare a `Shape` class with its own `draw()` method. Listing 4-12 presents `Shape`'s declaration.

Listing 4-12. Declaring a Shape Class

```
public class Shape {
    public void draw() {
    }
}
```

Listing 4-12's `Shape` class declares an empty `draw()` method that only exists to be overridden and to demonstrate subtype polymorphism.

You can now refactor Listing 4-9's `Point` class to extend Listing 4-12's `Shape` class, leave `Circle` as is, and introduce a `Rectangle` class that extends `Shape`. You can then refactor Listing 4-10's `Graphics` class's `main()` method to take `Shape` into account. Listing 4-13 presents the resulting `Graphics` class.

Listing 4-13. A `Graphics` Class with a New `main()` Method That Takes `Shape` into Account

```
public class Graphics {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[] { new Point(10, 20), new Circle(10, 20, 30),
                                      new Rectangle(20, 30, 15, 25) };
    }
}
```

```

        for (int i = 0; i < shapes.length; i++)
            shapes[i].draw();
    }
}

```

Because Point and Rectangle directly extend Shape, and because Circle indirectly extends Shape by extending Point, Listing 4-13's main() method will call the appropriate subclass's draw() method in response to shapes[i].draw();.

Although Shape makes the code more flexible, there is a problem. What is to stop the developer from instantiating Shape and adding this meaningless instance to the shapes array, as follows?

```

Shape[] shapes = new Shape[] { new Point(10, 20), new Circle(10, 20, 30),
                             new Rectangle(20, 30, 15, 25), new Shape()
};

```

What does it mean to instantiate Shape? Because this class describes an abstract concept, what does it mean to draw a generic shape? Fortunately, Java provides a solution to this problem, which is demonstrated in Listing 4-14.

Listing 4-14. Abstracting the Shape Class

```

abstract public class Shape {
    abstract public void draw(); // semicolon is required
}

```

Listing 4-14 uses Java's `abstract` reserved word to declare a class that cannot be instantiated. The compiler reports an error when you try to instantiate this class.

Tip Get into the habit of declaring classes that describe generic categories (such as `shape`, `animal`, `vehicle`, and `account`) `abstract`. This way, you will not inadvertently instantiate them.

The `abstract` reserved word is also used to declare a method without a body. The `draw()` method doesn't need a body because it cannot draw an abstract shape.

An abstract class can contain non-abstract methods in addition to or instead of abstract methods. For example, Listing 4-2's Vehicle class could have been declared abstract. The constructor would still be present, to initialize private fields, even though you could not instantiate the resulting class.

Downcasting and Runtime Type Identification

Moving up the type hierarchy, via upcasting, causes loss of access to subtype features. For example, assigning a Circle instance to Point variable p means that you cannot use p to call Circle's getRadius() method.

However, it is possible to once again access the Circle instance's getRadius() method by performing an explicit cast operation, for example, Circle c = (Circle) p;. This assignment is known as *downcasting* because you are explicitly moving down the type hierarchy (from the Point superclass to the Circle subclass).

Although an upcast is always safe (the superclass's interface is a subset of the subclass's interface), the same cannot be said of a downcast. Listing 4-15 shows you what kind of trouble you can get into when downcasting is used incorrectly.

Listing 4-15. The Trouble with Downcasting

```
public class A {  
}  
  
public class B extends A {  
    public void m() {  
    }  
}  
  
public class DowncastDemo {  
    public static void main(String[] args) {  
        A a = new A();  
        B b = (B) a;  
        b.m();  
    }  
}
```

Listing 4-15 presents a class hierarchy consisting of a superclass named A and a subclass named B. Although A doesn't declare any members, B declares a single m() method.

A third class named `DowncastDemo` provides a `main()` method that first instantiates `A` and then tries to downcast this instance to `B` and assign the result to variable `b`. The compiler will not complain because downcasting from a superclass to a subclass in the same type hierarchy is legal.

However, if the assignment is allowed, the application will undoubtedly crash when it tries to execute `b.m();`. The crash happens because the virtual machine will attempt to call a method that doesn't exist—class `A` doesn't have an `m()` method.

Fortunately, this scenario will never happen because the virtual machine verifies that the cast is legal before performing the cast operation. Because it detects that `A` doesn't have an `m()` method, it doesn't permit the cast by throwing an instance of the `ClassCastException` class.

The virtual machine's cast verification illustrates *runtime type identification* (or *RTTI*, for short). Cast verification performs RTTI by examining the type of the cast operator's operand to see whether the cast should be allowed or not. Clearly, the cast should not be allowed.

A second form of RTTI involves the `instanceof` operator. This operator checks the left operand to see whether or not it is an instance of the right operand and returns true if this is the case. The following example introduces `instanceof` to Listing 4-15 to prevent the `ClassCastException`:

```
if(a instanceof B) {
    B b = (B) a;
    b.m();
}
```

The `instanceof` operator detects that variable `a`'s instance was not created from `B` and returns false to indicate this fact. As a result, the code that performs the illegal cast will not execute. (Overuse of `instanceof` probably indicates poor software design.)

Because a subtype is a kind of supertype, `instanceof` will return true when its left operand is a subtype instance or a supertype instance of its right operand supertype. The following example demonstrates this:

```
A a = new A();
B b = new B();
System.out.println(b instanceof A); // Output: true
System.out.println(a instanceof A); // Output: true
```

This example assumes the class structure shown in Listing 4-15 and instantiates superclass A and subclass B. The first `System.out.println()` method call outputs true because b's reference identifies an instance of a subclass of A; the second `System.out.println()` method call outputs true because a's reference identifies an instance of superclass A.

You can also downcast from one array to another provided that the array being downcast is a supertype of the other array, and its elements types are those of the subtype. Consider Listing 4-16.

Listing 4-16. Demonstrating Array Downcasting

```
public class Point {
    ...
}

public class ColoredPoint extends Point {
    ...
}

public class DowncastArrayDemo {
    public static void main(String[] args) {
        ColoredPoint[] cptArray = new ColoredPoint[]{
            new ColoredPoint(10, 20, 5)};
        Point[] ptArray = cptArray;
        System.out.println(ptArray[0].getX()); // Output: 10
        System.out.println(ptArray[0].getY()); // Output: 20
        if (ptArray instanceof ColoredPoint[]) {
            ColoredPoint cp = (ColoredPoint) ptArray[0];
            System.out.println(cp.getColor());
        }
    }
}
```

Listing 4-16 is similar to Listing 4-11 except that it also demonstrates downcasting. Notice its use of `instanceof` to verify that `ptArray`'s referenced object is of type `ColoredPoint[]`. If this operator returns true, it is safe to downcast `ptArray[0]` from `Point` to `ColoredPoint` and assign the reference to `ColoredPoint`.

Covariant Return Types

A *covariant return type* is a method return type that, in the superclass's method declaration, is the supertype of the return type in the subclass's overriding method declaration. Listing 4-17 provides a demonstration of this language feature.

Listing 4-17. A Demonstration of Covariant Return Types

```
public class SuperReturnType {
    @Override
    public String toString() {
        return "superclass return type";
    }
}

public class SubReturnType extends SuperReturnType {
    @Override
    public String toString() {
        return "subclass return type";
    }
}

public class Superclass {
    public SuperReturnType createReturnType() {
        return new SuperReturnType();
    }
}

public class Subclass extends Superclass {
    @Override
    public SubReturnType createReturnType() {
        return new SubReturnType();
    }
}

public class CovarDemo {
    public static void main(String[] args) {
        SuperReturnType suprt = new Superclass().createReturnType();
        System.out.println(suprt); // Output: superclass return type
    }
}
```

```

SubReturnType subrt = new Subclass().createReturnType();
System.out.println(subrt); // Output: subclass return type
}
}

```

Listing 4-17 declares SuperReturnType and Superclass superclasses and SubReturnType and Subclass subclasses; each of Superclass and Subclass declares a `createReturnType()` method. Superclass's method has its return type set to SuperReturnType, whereas Subclass's overriding method has its return type set to SubReturnType, a subclass of SuperReturnType.

Covariant return types minimize upcasting and downcasting. For example, Subclass's `createReturnType()` method doesn't need to upcast its SubReturnType instance to its SubReturnType return type. Furthermore, this instance doesn't need to be downcast to SubReturnType when assigning to variable `subrt`.

Formalizing Class Interfaces

In our introduction to information hiding (see Chapter 3), we stated that every class *X* exposes an interface (a protocol consisting of constructors, methods, and possibly fields that are made available to objects created from other classes for use in creating and communicating with *X*'s objects).

Java formalizes the interface concept by providing reserved word `interface`, which is used to introduce a type without implementation. Java also provides language features to declare, implement, and extend interfaces. After looking at interface declaration, implementation, and extension in this section, we explain the rationale for using interfaces.

Declaring Interfaces

An interface declaration consists of a header followed by a body. At minimum, the header consists of reserved word `interface` followed by a name that identifies the interface. The body starts with an open brace character and ends with a close brace. Sandwiched between these delimiters are constant and method header declarations. Consider Listing 4-18.

Listing 4-18. Declaring a Drawable Interface

```
public interface Drawable {
    int RED = 1;    // For simplicity, integer constants are used.
                    // These constants are
    int GREEN = 2; // not that descriptive, as you will see.
    int BLUE = 3;
    int BLACK = 4;
    void draw(int color);
}
```

Listing 4-18 declares an interface named `Drawable`. By convention, an interface's name begins with an uppercase letter. Furthermore, the first letter of each subsequent word in a multiword interface name is capitalized.

Note Many interface names end with the `able` suffix. For example, the standard class library includes interfaces named `Callable`, `Comparable`, `Cloneable`, `Iterable`, `Runnable`, and `Serializable`. It is not mandatory to use this suffix; the standard class library also provides interfaces named `CharSequence`, `Collection`, `Executor`, `Future`, `Iterator`, `List`, `Map`, and `Set`.

`Drawable` declares four fields that identify color constants. `Drawable` also declares a `draw()` method that must be called with one of these constants to specify the color used to draw something.

Note Because an interface is already abstract, it is redundant to specify `abstract` in the interface's declaration. An interface's fields are implicitly declared `public`, `static`, and `final`. It is therefore redundant to declare them with these reserved words. Finally, an interface's methods are implicitly declared `public` and `abstract`. Therefore, it is redundant to declare them with these reserved words.

`Drawable` identifies a type that specifies what to do (draw something) but not how to do it. It leaves implementation details to classes that implement this interface. Instances of such classes are known as *drawables* because they know how to draw themselves.

Note An interface that declares no members is known as a *marker interface* or a *tagging interface*. It associates metadata with a class. For example, the presence of the `Cloneable` marker/tagging interface implies that instances of its implementing class can be shallowly cloned. RTTI is used to detect that an object's class implements a marker/tagging interface. For example, when `Object`'s `clone()` method detects, via RTTI, that the calling instance's class implements `Cloneable`, it shallowly clones the object.

Implementing Interfaces

By itself, an interface is useless. To be of any benefit to an application, the interface needs to be implemented by a class. Java provides the `implements` reserved word for this task. This reserved word is demonstrated in Listing 4-19.

Listing 4-19. Implementing the `Drawable` Interface

```
public class Point implements Drawable {  
    private int x, y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    @Override  
    public String toString() {  
        return "(" + x + ", " + y + ")";  
    }  
}
```

```

@Override
public void draw(int color) {
    System.out.println("Point drawn at " + toString() + " in color " +
        color);
}
}

public class Circle extends Point implements Drawable {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }

    public int getRadius() {
        return radius;
    }

    @Override
    public String toString() {
        return "" + radius;
    }

    @Override
    public void draw(int color) {
        System.out.println("Circle drawn at " + super.toString() +
            " with radius " + toString() + " in color " +
            color);
    }
}

```

Listing 4-19 retrofits Listing 4-9's class hierarchy to take advantage of Listing 4-18's `Drawable` interface. You will notice that each of classes `Point` and `Circle` implements this interface by attaching the `implements Drawable` clause to its class header.

To implement an interface, the class must specify, for each interface method header, a method whose header has the same signature and return type as the interface's method header and a code body to go with the method header.

Caution When implementing a method, don't forget that the interface's methods are implicitly declared `public`. If you forget to include `public` in the implemented method's declaration, the compiler will report an error because you are attempting to assign weaker access to the implemented method.

When a class implements an interface, the class inherits the interface's constants and method headers and overrides the method headers by providing implementations (hence the `@Override` annotation). This is known as *interface inheritance*.

It turns out that `Circle`'s header doesn't need the `implements Drawable` clause. If this clause is not present, `Circle` inherits `Point`'s `draw()` method and is still considered to be a `Drawable`, whether it overrides this method or not.

An interface specifies a type whose data values are the objects whose classes implement the interface and whose behaviors are those specified by the interface. This fact implies that you can assign an object's reference to a variable of the interface type, provided that the object's class implements the interface. The following example provides a demonstration:

```
public static void main(String[] args) {
    Drawable[] drawables = new Drawable[] { new Point(10, 20), new
    Circle(10, 20, 30) };
    for (int i = 0; i < drawables.length; i++)
        drawables[i].draw(Drawable.RED);
}
```

Because `Point` and `Circle` instances are drawables by virtue of these classes implementing the `Drawable` interface, it is legal to assign `Point` and `Circle` instance references to variables (including array elements) of type `Drawable`.

When you run this method, it generates the following output:

```
Point drawn at (10, 20) in color 1
Circle drawn at (10, 20) with radius 30 in color 1
```

[Listing 4-18](#)'s `Drawable` interface is useful for drawing a shape's outline. Suppose you also need to fill a shape's interior. You might attempt to satisfy this requirement by declaring [Listing 4-20](#)'s `Fillable` interface.

Listing 4-20. Declaring a Fillable Interface

```
public interface Fillable {
    int RED = 1;
    int GREEN = 2;
    int BLUE = 3;
    int BLACK = 4;
    void fill(int color);
}
```

Given Listings 4-18 and 4-20, you can declare that the `Point` and `Circle` classes implement both interfaces by specifying `class Point implements Drawable, Fillable` and `class Circle implements Drawable, Fillable`. You can then modify the `main()` method to also treat the drawables as *fillables* so that you can fill these shapes, as follows:

```
public static void main(String[] args) {
    Drawable[] drawables = new Drawable[] { new Point(10, 20),
                                             new Circle(10, 20, 30) };
    for (int i = 0; i < drawables.length; i++)
        drawables[i].draw(Drawable.RED);
    Fillable[] fillables = new Fillable[drawables.length];
    for (int i = 0; i < drawables.length; i++) {
        fillables[i] = (Fillable) drawables[i];
        fillables[i].fill(Fillable.GREEN);
    }
}
```

After invoking each drawable's `draw()` method, `main()` creates a `Fillable` array of the same length as the `Drawable` array. It then proceeds to copy each `Drawable` array element to a `Fillable` array element and then invoke the `fillable's fill()` method. The `(Fillable)` cast is necessary because a `drawable` is not a `fillable`. This cast operation will succeed because the `Point` and `Circle` instances being copied implement `Fillable` as well as `Drawable`.

Tip You can list as many interfaces as you need to implement by specifying a comma-separated list of interface names after `implements`.

Extending Interfaces

Just as a subclass can extend a superclass via reserved word `extends`, you can use this reserved word to have a *subinterface* extend a *superinterface*. This, too, is known as *interface inheritance*.

For example, the duplicate color constants in `Drawable` and `Fillable` lead to name collisions when you specify their names by themselves in an implementing class. To avoid these name collisions, prefix a name with its interface name and the member access operator, or place these constants in their own interface, and have `Drawable` and `Fillable` extend this interface, as demonstrated in Listing 4-21.

Listing 4-21. Extending the Colors Interface

```
public interface Colors {
    int RED = 1;
    int GREEN = 2;
    int BLUE = 3;
    int BLACK = 4;
}

public interface Drawable extends Colors {
    void draw(int color);
}

public interface Fillable extends Colors {
    void fill(int color);
}
```

The fact that `Drawable` and `Fillable` both inherit constants from `Colors` is not a problem for the compiler. There is only a single copy of these constants (in `Colors`) and no possibility of a name collision, and so the compiler is satisfied.

If a class can implement multiple interfaces by declaring a comma-separated list of interface names after `implements`, it seems that an interface should be able to extend multiple interfaces in a similar way. This feature is demonstrated in Listing 4-22.

Listing 4-22. Extending a Pair of Interfaces

```
public interface A {
    int X = 1;
}

public interface B {
    double X = 2.0;
}

public interface C extends A, B {
```

}

Listing 4-22 will compile even though C inherits two same-named constants X with different types and initializers. However, if you implement C and then try to access X, as in Listing 4-23, you will run into a name collision.

Listing 4-23. Discovering a Name Collision

```
public class Collision implements C {
    public void output() {
        System.out.println(X); // Which X is accessed?
    }
}
```

Suppose you introduce a `void foo();` method header declaration into interface A and an `int foo();` method header declaration into interface B. This time, the compiler will report an error when you attempt to compile the modified Listing 4-22.

Why Use Interfaces?

Now that the mechanics of declaring, implementing, and extending interfaces are out of the way, you can focus on the rationale for using them. Unfortunately, newcomers to Java's interfaces feature are often told that this feature was created as a workaround to Java's lack of support for multiple implementation inheritance. While interfaces are useful in this capacity, this is not their reason for existence. Instead, ***Java's interfaces feature was created to give developers the utmost flexibility in designing their applications by decoupling interface from implementation. You should always code to the interface (supplied by an interface type or an abstract class).***

Those who are adherents to *agile software development* (a group of software development methodologies based on iterative development that emphasizes keeping code simple, testing frequently, and delivering functional pieces of the application as soon as they are deliverable) know the importance of flexible coding. They cannot afford to tie their code to a specific implementation because a change in requirements for the next iteration could result in a new implementation, and they might find themselves rewriting significant amounts of code, which wastes time and slows development.

Interfaces help you achieve flexibility by decoupling interface from implementation. For example, the `main()` method in Listing 4-13's `Graphics` class creates an array of objects from classes that subclass the `Shape` class, and then iterates over these objects, calling each object's `draw()` method. The only objects that can be drawn are those that subclass `Shape`.

Suppose you also have a hierarchy of classes that model resistors, transistors, and other electronic components. Each component has its own symbol that allows the component to be shown in a schematic diagram of an electronic circuit. Perhaps you want to add a drawing capability to each class that draws that component's symbol.

You might consider specifying `Shape` as the superclass of the electronic component class hierarchy. However, electronic components are not shapes (although they have shapes), so it makes no sense to place these classes in a class hierarchy rooted in `Shape`.

However, you can make each component class implement the `Drawable` interface, which lets you add expressions that instantiate these classes to the `drawables` array in the `main()` method appearing prior to Listing 4-20 (so you can draw their symbols). This is legal because these instances are drawables.

Wherever possible, you should strive to specify interfaces instead of classes in your code to keep your code adaptable to change. This is especially true when working with Java's Collections Framework, which we will discuss at length in Chapter 9.

Tip Always strive to specify interfaces instead of classes to keep your code adaptable to change.

For now, consider a simple example that consists of the Collections Framework's `java.util.List` interface and its `java.util.ArrayList` and `java.util.LinkedList` implementation classes. The following example presents inflexible code based on the `ArrayList` class:

```
ArrayList<String> arrayList = new ArrayList<String>();  
void dump(ArrayList<String> arrayList) {  
    // suitable code to dump out the arrayList  
}
```

This example uses the generics-based parameterized type language feature (which we will discuss in Chapter 6) to identify the kind of objects stored in an `ArrayList` instance. In this example, `String` objects are stored.

The example is inflexible because it hardwires the `ArrayList` class into multiple locations. This hardwiring focuses the developer into thinking specifically about array lists instead of generically about lists.

Lack of focus is problematic when requirements change, or perhaps a performance issue brought about by *profiling* (analyzing a running application to check its performance) suggests that the developer should have used `LinkedList`.

The example only requires a minimal number of changes to satisfy the new requirement. In contrast, a larger code base might need many more changes. Although you only need to change `ArrayList` to `LinkedList`, to satisfy the compiler, consider changing `arrayList` to `linkedList` to keep *semantics* (meaning) clear—you might have to change multiple occurrences of names that refer to an `ArrayList` instance throughout the source code.

The developer is bound to lose time while refactoring the code to adapt to `LinkedList`. Instead, time could have been saved by writing this example to use the equivalent of constants. In other words, the example could have been written to rely on interfaces and to only specify `ArrayList` in one place. The following example shows you what the resulting code would look like:

```
List<String> list = new ArrayList<String>();  
void dump(List<String> list) {  
    // suitable code to dump out the list  
}
```

This example is much more flexible than the previous example. If a requirements or profiling change suggests that `LinkedList` should be used instead of `ArrayList`, simply replace `Array` with `Linked` and you are done. You don't even have to change the parameter name.

Note Java provides interfaces and abstract classes for describing *abstract types* (types that cannot be instantiated). Abstract types represent abstract concepts (e.g., `drawable` and `shape`), and instances of such types would be meaningless.

Interfaces promote flexibility through lack of implementation—`Drawable` and `List` illustrate this flexibility. They are not tied to any single class hierarchy but can be implemented by any class in any hierarchy. In contrast, abstract classes support implementation but can be genuinely abstract (e.g., Listing 4-14's abstract `Shape` class). However, they are limited to appearing in the upper levels of class hierarchies.

Interfaces and abstract classes can be used together. For example, the Collections Framework's `java.util` package provides `List`, `Map`, and `Set` interfaces and `AbstractList`, `AbstractMap`, and `AbstractSet` abstract classes that provide skeletal implementations of these interfaces.

By implementing many interface methods, the skeletal implementations make it easy for you to create your own interface implementations, to address your unique requirements. If they don't meet your needs, you can optionally have your class directly implement the appropriate interface.

EXERCISES

The following exercises are designed to test your understanding of Chapter 4's content:

1. What is implementation inheritance?
2. How does Java support implementation inheritance?
3. Can a subclass have two or more superclasses?
4. How do you prevent a class from being subclassed?
5. True or false: The `super()` call can appear in any method.

6. If a superclass declares a constructor with one or more parameters, and if a subclass constructor doesn't use `super()` to call that constructor, why does the compiler report an error?
7. What is an immutable class?
8. True or false: A class can inherit constructors.
9. What does it mean to override a method?
10. What is required to call a superclass method from its overriding subclass method?
11. How do you prevent a method from being overridden?
12. Why can you not make an overriding subclass method less accessible than the superclass method it is overriding?
13. How do you tell the compiler that a method overrides another method?
14. Why does Java not support multiple implementation inheritance?
15. What is the name of Java's ultimate superclass?
16. What is the purpose of the `clone()` method?
17. When does `Object`'s `clone()` method throw `CloneNotSupportedException`?
18. Can the `==` operator be used to determine if two objects are logically equivalent? Why or why not?
19. What does `Object`'s `equals()` method accomplish?
20. How can you optimize a time-consuming `equals()` method?
21. What is the purpose of the `finalize()` method?
22. Should you rely on `finalize()` for closing open files? Why or why not?
23. What is a hash code?
24. True or false: You should override the `hashCode()` method whenever you override the `equals()` method.
25. What does `Object`'s `toString()` method return?
26. Why should you override `toString()`?

27. Define composition.
28. True or false: Composition is used to describe “is-a” relationships and implementation inheritance is used to describe “has-a” relationships.
29. Define subtype polymorphism.
30. How is subtype polymorphism accomplished?
31. Why would you use abstract classes and abstract methods?
32. Can an abstract class contain concrete methods?
33. What is the purpose of downcasting?
34. List two forms of RTTI.
35. What is a covariant return type?
36. How do you formally declare an interface?
37. Define marker interface.
38. What is interface inheritance?
39. How do you implement an interface?
40. How do you form a hierarchy of interfaces?
41. Why is Java’s interfaces feature so important?
42. What do interfaces and abstract classes accomplish?
43. How do interfaces and abstract classes differ?
44. Model part of an animal hierarchy by declaring Animal, Bird, Fish, AmericanRobin, DomesticCanary, RainbowTrout, and SockeyeSalmon classes:
 - Animal is public and abstract, declares private String-based kind and appearance fields, declares a public constructor that initializes these fields to passed-in arguments, declares public and abstract eat() and move() methods that take no arguments and whose return type is void, and overrides the toString() method to output the contents of kind and appearance.

- Bird is public and abstract, extends Animal, declares a public constructor that passes its kind and appearance parameter values to its superclass constructor, overrides its eat() method to output eats seeds and insects (via System.out.println()), and overrides its move() method to output flies through the air.
- Fish is public and abstract; extends Animal; declares a public constructor that passes its kind and appearance parameter values to its superclass constructor; overrides its eat() method to output eats krill, algae, and insects; and overrides its move() method to output swims through the water.
- AmericanRobin is public, extends Bird, and declares a public noargument constructor that passes "americanrobin" and "red breast" to its superclass constructor.
- DomesticCanary is public, extends Bird, and declares a public noargument constructor that passes "domesticcanary" and "yellow, orange, black, brown, white, red" to its superclass constructor.
- RainbowTrout is public, extends Fish, and declares a public noargument constructor that passes "rainbowtrout" and "bands of brilliant speckled multicolored stripes running nearly the whole length of its body" to its superclass constructor.
- SockeyeSalmon is public, extends Fish, and declares a public noargument constructor that passes "sockeyesalmon" and "bright red with a green head" to its superclass constructor.

Note For brevity, we have omitted from the Animal hierarchy abstract Robin, Canary, Trout, and Salmon classes that generalize robins, canaries, trout, and salmon. Perhaps you might want to include these classes in the hierarchy.

Although this exercise illustrates the accurate modeling of a natural scenario using inheritance, it also reveals the potential for *class explosion*—too many classes may be introduced to model a scenario, and it might be difficult to maintain all of these classes. Keep this in mind when modeling with inheritance.

45. Continuing from the previous exercise, declare an `Animals` class with a `main()` method. This method first declares an `animals` array that is initialized to `AmericanRobin`, `RainbowTrout`, `DomesticCanary`, and `SockeyeSalmon` objects. The method then iterates over this array, first outputting `animals[i]` (which causes `toString()` to be called) and then calling each object's `eat()` and `move()` methods (demonstrating subtype polymorphism).
46. Continuing from the previous exercise, declare a `public Countable` interface with a `String getID()` method. Modify `Animal` to implement `Countable` and have this method return kind's value. Modify `Animals` to initialize the `animals` array to `AmericanRobin`, `RainbowTrout`, `DomesticCanary`, `SockeyeSalmon`, `RainbowTrout`, and `AmericanRobin` objects. Also, introduce code that computes a census of each kind of animal. This code will use the `Census` class that is declared in Listing 4-24.

Listing 4-24. The `Census` Class Stores Census Data on Four Kinds of Animals

```
public class Census {
    public final static int SIZE = 4;
    private String[] IDs;
    private int[] counts;

    public Census() {
        IDs = new String[SIZE];
        counts = new int[SIZE];
    }

    public String get(int index) {
        return IDs[index] + " " + counts[index];
    }

    public void update(String ID) {
        for (int i = 0; i < IDs.length; i++) {
            // If ID is not already stored in the IDs array (which is indicated by
            // the first null entry that is found), store ID in this array, and
            // also assign 1 to the associated element in the counts array, to
            // initialize the census for that ID.
        }
    }
}
```

```
if (IDs[i] == null) {  
    IDs[i] = ID;  
    counts[i] = 1;  
    return;  
}  
  
// If a matching ID is found, increment the associated element in  
// the counts array to update the census for that ID.  
if (IDs[i].equals(ID)) {  
    counts[i]++;  
    return;  
}  
}  
}  
}
```

Summary

Inheritance is a hierarchical relationship between similar entity categories in which one category inherits state and behaviors from at least one other entity category. Inheriting from a single category is called single inheritance, and inheriting from at least two categories is called multiple inheritance.

Java supports single inheritance and multiple inheritance to facilitate code reuse. Java supports single inheritance in a class context (via reserved word `extends`), in which a class inherits fields and methods from another class through class extension. Because classes are involved, Java refers to this kind of inheritance as implementation inheritance. Java supports multiple inheritance only in an interface context, in which a class inherits method templates from one or more interfaces through interface implementation (via reserved word `implements`), or in which an interface inherits method templates from one or more interfaces through interface extension (via reserved word `extends`). Because interfaces are involved, Java refers to this kind of inheritance as interface inheritance.

CHAPTER 4 DISCOVERING INHERITANCE, POLYMORPHISM, AND INTERFACES

Some real-world entities have the ability to change their forms. The ability to change form is known as polymorphism and is useful to model in a programming language. Although Java supports the coercion, overloading, parametric, and subtype kinds of polymorphism, in this chapter we focused only on subtype polymorphism, which is achieved through upcasting and method overriding.

Every class *X* exposes an interface (a protocol consisting of constructors, methods, and [possibly] fields that are made available to objects created from other classes for use in creating and communicating with *X*'s objects). Java formalizes the interface concept by providing reserved word `interface`, which is used to introduce a type without implementation.

Although many believe that the interfaces language feature was created as a workaround to Java's lack of support for multiple implementation inheritance, this is not the real reason for its existence. Instead, Java's interfaces feature was created to give developers the utmost flexibility in designing their applications by decoupling interface from implementation. You should always code to the interface.

Chapter 5 continues to explore the Java language by focusing on nested types, packages, static imports, and exceptions.

CHAPTER 5

Mastering Advanced Language Features, Part 1

In Chapters 2 through 4, we laid a foundation for learning the Java language. In Chapter 5, we will add to this foundation by introducing you to some of Java’s more advanced language features, specifically those features related to nested types, packages, static imports, and exceptions. Additional advanced language features are covered in Chapter 6.

Mastering Nested Types

Classes that are declared outside of any class are known as *top-level classes*. Java also supports *nested classes*, which are classes that are declared as members of other classes or scopes. Nested classes help you implement top-level class architecture.

There are four kinds of nested classes: static member classes, nonstatic member classes, anonymous classes, and local classes. The latter three categories are known as *inner classes*.

In this section, we will introduce you to static member classes and inner classes. We will then briefly examine nesting interfaces within classes.

Static Member Classes

A *static member class* is a *static* member of an enclosing class. Although enclosed, it doesn’t have an enclosing instance of that class and cannot access the enclosing class’s instance fields and invoke its instance methods. However, it can access the enclosing class’s static fields and invoke its static methods, even those members that are declared *private*. Listing 5-1 presents a static member class declaration.

Listing 5-1. Declaring a Static Member Class

```
public class EnclosingClass {  
    private static int i;  
  
    private static void m1() {  
        System.out.println(i);  
    }  
  
    static void m2() {  
        EnclosedClass.accessEnclosingClass();  
    }  
  
    static public class EnclosedClass {  
        static public void accessEnclosingClass() {  
            i = 1;  
            m1();  
        }  
  
        public void accessEnclosingClass2() {  
            m2();  
        }  
    }  
}
```

[Listing 5-1](#) declares a top-level class named `EnclosingClass` with class field `i`, class methods `m1()` and `m2()`, and static member class `EnclosedClass`. Also, `EnclosedClass` declares class method `accessEnclosingClass()` and instance method `accessEnclosingClass2()`.

Because `accessEnclosingClass()` is declared `static`, `m2()` must be prefixed with `EnclosedClass` and the member access operator to call this method.

[Listing 5-2](#) presents the source code to an application class that demonstrates how to invoke `EnclosedClass`'s `accessEnclosingClass()` class method and instantiate `EnclosedClass` and invoke its `accessEnclosingClass2()` instance method.

Listing 5-2. Invoking a Static Member Class's Class and Instance Methods

```
public class SMCDemo {
    public static void main(String[] args) {
        EnclosingClass.EnclosedClass.accessEnclosingClass(); // Output: 1
        EnclosingClass.EnclosedClass ec = new EnclosingClass.EnclosedClass();
        ec.accessEnclosingClass2(); // Output: 1
    }
}
```

Listing 5-2's `main()` method reveals that you must prefix the name of an enclosed class with the name of its enclosing class to invoke a class method, for example, `EnclosingClass.EnclosedClass.accessEnclosingClass();`.

This listing also reveals that you must prefix the name of the enclosed class with the name of its enclosing class when instantiating the enclosed class, for example, `EnclosingClass.EnclosedClass ec = new EnclosingClass.EnclosedClass();`. You can then invoke the instance method in the normal manner, for example, `ec.accessEnclosingClass2();`.

Static member classes have their uses. For example, Listing 5-3's Double and Float static member classes provide different implementations of their enclosing Rectangle class. The Float version occupies less memory because of its 32-bit float fields, and the Double version provides greater accuracy because of its 64-bit double fields.

Listing 5-3. Using Static Member Classes to Declare Multiple Implementations of Their Enclosing Class

```
abstract public class Rectangle {
    abstract public double getX();
    abstract public double getY();
    abstract public double getWidth();
    abstract public double getHeight();

    static public class Double extends Rectangle {
        private double x, y, width, height;

        public Double(double x, double y, double width, double height) {
            this.x = x;
            this.y = y;
        }
    }
}
```

```
    this.width = width;
    this.height = height;
}

public double getX() { return x; }
public double getY() { return y; }
public double getWidth() { return width; }
public double getHeight() { return height; }

}

static public class Float extends Rectangle {
    private float x, y, width, height;

    public Float(float x, float y, float width, float height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    public double getX() { return x; }
    public double getY() { return y; }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
}

// Prevent subclassing. Use the type-specific Double and Float
// implementation subclass classes to instantiate.

private Rectangle() {}

public boolean contains(double x, double y) {
    return (x >= getX() && x < getX() + getWidth()) &&
           (y >= getY() && y < getY() + getHeight());
}
}
```

Listing 5-3's Rectangle class demonstrates nested subclasses. Each of the Double and Float static member classes subclasses the abstract Rectangle class, providing

private floating-point or double-precision floating-point fields and overriding Rectangle's abstract methods to return these fields' values as doubles.

Rectangle is abstract because it makes no sense to instantiate this class. Because it also makes no sense to directly extend Rectangle with new implementations (the Double and Float nested subclasses should be sufficient), its default constructor is declared private. Instead, you must instantiate Rectangle.Float (to save memory) or Rectangle.Double (when accuracy is required), as demonstrated by Listing 5-4.

Listing 5-4. Creating and Using Different Rectangle Implementations

```
public class SMCDemo {
    public static void main(String[] args) {
        Rectangle r = new Rectangle.Double(10.0, 10.0, 20.0, 30.0);
        System.out.println("x = " + r.getX());
        System.out.println("y = " + r.getY());
        System.out.println("width = " + r.getWidth());
        System.out.println("height = " + r.getHeight());
        System.out.println("contains(15.0, 15.0) = " + r.contains(15.0, 15.0));
        System.out.println("contains(0.0, 0.0) = " + r.contains(0.0, 0.0));
        System.out.println();
        r = new Rectangle.Float(10.0f, 10.0f, 20.0f, 30.0f);
        System.out.println("x = " + r.getX());
        System.out.println("y = " + r.getY());
        System.out.println("width = " + r.getWidth());
        System.out.println("height = " + r.getHeight());
        System.out.println("contains(15.0, 15.0) = " + r.contains(15.0, 15.0));
        System.out.println("contains(0.0, 0.0) = " + r.contains(0.0, 0.0));
    }
}
```

Listing 5-4 first instantiates Rectangle's Double subclass via new Rectangle.Double(10.0, 10.0, 20.0, 30.0) and then invokes its various methods. Continuing, Listing 5-4 instantiates Rectangle's Float subclass via new Rectangle.Float(10.0f, 10.0f, 20.0f, 30.0f) before invoking Rectangle methods on this instance.

Compile both listings (`javac SMCDemo.java` or `javac *.java`) and run the application (`java SMCDemo`). You will then observe the following output:

```
x = 10.0
y = 10.0
width = 20.0
height = 30.0
contains(15.0, 15.0) = true
contains(0.0, 0.0) = false

x = 10.0
y = 10.0
width = 20.0
height = 30.0
contains(15.0, 15.0) = true
contains(0.0, 0.0) = false
```

Java's class library contains many static member classes. For example, the `java.lang.Character` class encloses a static member class named `Subset` whose instances represent subsets of the Unicode character set. Additional examples include `java.util.AbstractMap.SimpleEntry` and `java.io.ObjectInputStream.GetField`.

Note When you compile an enclosing class that contains a static member class, the compiler creates a class file for the static member class whose name consists of its enclosing class's name, a dollar-sign character, and the static member class's name. For example, compile Listing 5-1 and you will discover `EnclosingClass$EnclosedClass.class` in addition to `EnclosingClass.class`. This format also applies to nonstatic member classes.

Nonstatic Member Classes

A *nonstatic member class* is a non-static member of an enclosing class. Each instance of the nonstatic member class implicitly associates with an instance of the enclosing class. The nonstatic member class's instance methods can call instance methods in the enclosing class and access the enclosing class instance's nonstatic fields. Listing 5-5 presents a nonstatic member class declaration.

Listing 5-5. Declaring a Nonstatic Member Class

```
public class EnclosingClass {
    private int i;

    private void m() {
        System.out.println(i);
    }

    public class EnclosedClass {
        public void accessEnclosingClass() {
            i = 1;
            m();
        }
    }
}
```

Listing 5-5 declares a top-level class named `EnclosingClass` with instance field `i`, instance method `m()`, and nonstatic member class `EnclosedClass`. Furthermore, `EnclosedClass` declares instance method `accessEnclosingClass()`.

Because `accessEnclosingClass()` is nonstatic, `EnclosedClass` must be instantiated before this method can be called. This instantiation must take place via an instance of `EnclosingClass`. Listing 5-6 accomplishes these tasks.

Listing 5-6. Calling a Nonstatic Member Class's Instance Method

```
public class NSMCDemo {
    public static void main(String[] args) {
        EnclosingClass ec = new EnclosingClass();
        ec.new EnclosedClass().accessEnclosingClass(); // Output: 1
    }
}
```

Listing 5-6's `main()` method first instantiates `EnclosingClass` and saves its reference in local variable `ec`. Then, `main()` uses this reference as a prefix to the `new` operator to instantiate `EnclosedClass`, whose reference is then used to call `accessEnclosingClass()`, which outputs 1.

Note Prefixing new with a reference to the enclosing class is rare. Instead, you will typically call an enclosed class's constructor from within a constructor or an instance method of its enclosing class.

Java's class library presents many examples of nonstatic member classes. For example, the `java.util` package's `HashMap` class declares private `HashIterator`, `ValueIterator`, `KeyIterator`, and `EntryIterator` classes for iterating over a hashmap's values, keys, and entries. (We will discuss `HashMap` in Chapter 9.)

Note Code within an enclosed class can obtain a reference to its enclosing class instance by qualifying reserved word `this` with the enclosing class's name and the member access operator. For example, if code within `accessEnclosingClass()` needed to obtain a reference to its `EnclosingClass` instance, it would specify `EnclosingClass.this`.

Anonymous Classes

An *anonymous class* is a class without a name. Furthermore, it is not a member of its enclosing class. Instead, an anonymous class is simultaneously declared (as an anonymous extension of a class or as an anonymous implementation of an interface) and instantiated any place where it is legal to specify an expression. Listing 5-7 demonstrates an anonymous class declaration and instantiation.

Listing 5-7. Declaring and Instantiating an Anonymous Class That Extends a Class

```
abstract public class Speaker {  
    abstract void speak();  
}  
  
public class ACDemo {  
    public static void main(final String[] args) {  
        new Speaker() {  
            String msg = (args.length == 1) ? args[0] : "nothing to say";  
        }  
    }  
}
```

```

@Override
void speak() {
    System.out.println(msg);
}
}.speak();
}
}

```

Listing 5-7 introduces an abstract class named Speaker and a concrete class named ACDemo. The latter class's main() method declares an anonymous class that extends Speaker and overrides its speak() method. When this method is called, it outputs main()'s first command-line argument or a default message when there are no arguments.

An anonymous class doesn't have a constructor (because the anonymous class doesn't have a name). However, its class file does contain an <init>() method that performs instance initialization. This method calls the superclass's noargument constructor (prior to any other initialization), which is the reason for specifying Speaker() after new.

Anonymous class instances should be able to access the surrounding scope's local variables and parameters. However, an instance might outlive the method in which it was conceived (as a result of storing the instance's reference in a field) and try to access local variables and parameters that no longer exist after the method returns.

Because Java cannot allow this illegal access, which would most likely crash the virtual machine, it lets an anonymous class instance only access local variables and parameters that are declared final (see Listing 5-7).

Listing 5-8 demonstrates an alternative anonymous class declaration and instantiation.

Listing 5-8. Declaring and Instantiating an Anonymous Class That Implements an Interface

```

public interface Speakable {
    void speak();
}

public class ACDemo {
    public static void main(final String[] args) {
        new Speakable(){
            String msg = (args.length == 1) ? args[0] : "nothing to say";

```

```

@Override
public void speak() {
    System.out.println(msg);
}
}.speak();
}
}

```

Listing 5-8 is very similar to Listing 5-7. However, instead of subclassing a Speaker class, this listing's anonymous class implements an interface named Speakable. Apart from the <init>() method calling java.lang.Object() (interfaces have no constructors), Listing 5-8 behaves like Listing 5-7.

Although an anonymous class doesn't have a constructor, you can provide an instance initializer to handle complex initialization. For example, new Office() {{addEmployee(new Employee("John Doe"));}}; instantiates an anonymous subclass of Office and adds one Employee object to this instance by calling Office's addEmployee() method.

You will often find yourself creating and instantiating anonymous classes for their convenience. For example, suppose you need to return a list of all file names having the .java suffix. The following example shows you how an anonymous class simplifies using the java.io package's File and FilenameFilter classes to achieve this objective:

```

String[] list = new File(directory).list(new FilenameFilter()
{
    @Override
    public boolean accept(File f, String s) {
        return s.endsWith(".java");
    }
});

```

However, keep in mind that there is a downside to using anonymous classes. Because they are anonymous, you cannot reuse anonymous classes in other parts of your applications.

Local Classes

A *local class* is a class that is declared anywhere that a local variable is declared. Furthermore, it has the same scope as a local variable. Like anonymous classes, local classes only have enclosing instances when used in nonstatic contexts.

A local class instance can access the surrounding scope's local variables and parameters. However, the local variables and parameters that are accessed must be declared `final`. For example, Listing 5-9's local class declaration accesses a `final` parameter and a `final` local variable.

Listing 5-9. Declaring a Local Class

```
public class EnclosingClass {  
    void m(final int x) {  
        final int y = x * 2;  
        class LocalClass {  
            int a = x;  
            int b = y;  
        }  
        LocalClass lc = new LocalClass();  
        System.out.println(lc.a);  
        System.out.println(lc.b);  
    }  
}
```

Listing 5-9 declares `EnclosingClass` with its instance method `m()` declaring a local class named `LocalClass`. This local class declares a pair of instance fields (`a` and `b`) that are initialized to the values of `final` parameter `x` and `final` local variable `y` when `LocalClass` is instantiated: `new EnclosingClass().m(10);`, for example. Listing 5-10 demonstrates this local class.

Listing 5-10. Demonstrating a Local Class

```
public class LCDemo {  
    public static void main(String[] args) {  
        EnclosingClass ec = new EnclosingClass();  
        ec.m(10);  
    }  
}
```

After instantiating `EnclosingClass`, Listing 5-10's `main()` method invokes `m(10)`. The called `m()` method multiplies this argument by 2; instantiates `LocalClass`, whose `<init>()` method assigns the argument and the doubled value to its pair of instance fields (in lieu of using a constructor to perform this task); and outputs the `LocalClass` instance fields. The following output results:

```
10
20
```

Local classes might help to improve code clarity because they can be moved closer to where they are needed.

Interfaces Within Classes

Interfaces can be nested within classes. Once declared, an interface is considered to be static even when it is not declared `static`. For example, Listing 5-11 declares an enclosing class named `X` along with two nested static interfaces named `A` and `B`.

Listing 5-11. Declaring a Pair of Interfaces Within a Class

```
public class X {
    interface A {
    }

    static interface B {
    }
}
```

You would access Listing 5-11's interfaces in the same way. For example, you would specify `class C implements X.A {}` or `class D implements X.B {}`.

As with nested classes, nested interfaces help to implement top-level class architecture by being implemented by nested classes. Collectively, these types are nested because they cannot (as in Listing 5-14's `Iter` local class) or need not appear at the same level as a top-level class and pollute its package namespace. The `java.util.Map.Entry` interface and `HashMap` class is a good example.

Mastering Packages

Hierarchical structures organize items in terms of hierarchical relationships that exist between those items. For example, a filesystem might contain a taxes directory with multiple year subdirectories, where each subdirectory contains tax information pertinent to that year. Also, an enclosing class might contain multiple nested classes that only make sense in the context of the enclosing class.

Hierarchical structures also help to avoid name conflicts. For example, two files cannot have the same name in a nonhierarchical filesystem (which consists of a single directory). In contrast, a hierarchical filesystem lets same-named files exist in different directories. Similarly, two enclosing classes can contain same-named nested classes. Name conflicts don't exist because items are partitioned into different *namespaces*.

Java also supports the partitioning of top-level user-defined types into multiple namespaces to better organize these types and to also prevent name conflicts. Java uses packages to accomplish these tasks.

In this section, we will introduce you to packages. After defining this term and explaining why package names must be unique, we will present the package and import statements. We will next explain how the virtual machine searches for packages and types and then we will present an example that shows you how to work with packages. We will close this section by showing you how to encapsulate a package of class files into JAR files.

Tip Except for the most trivial of top-level types and (typically) those classes that serve as application entry points (they have `main()` methods), you should consider storing your types (especially when they are reusable) in packages. Get into the habit now because you'll use packages extensively when developing Android apps. Each Android app must be stored in its own unique package.

What Are Packages?

A *package* is a unique namespace that can contain a combination of top-level classes, other top-level types, and subpackages. Only types that are declared `public` can be accessed from outside the package. Furthermore, the constants, constructors, methods, and nested types that describe a class's interface must be declared `public` to be accessible from beyond the package.

Every package has a name, which must be a nonreserved identifier. The member access operator separates a package name from a subpackage name and separates a package or subpackage name from a type name. For example, the two member access operators in `graphics.shapes.Circle` separate package name `graphics` from the `shapes` subpackage name and separate subpackage name `shapes` from the `Circle` type name.

Note Each of Oracle's and Google Android's standard class libraries organizes its many classes and other top-level types into multiple packages. Many of these packages are subpackages of the standard `java` package. Examples include `java.io` (types related to input/output operations), `java.lang` (language-oriented types), `java.net` (network-oriented types), and `java.util` (utility types).

Package Names Must Be Unique

Package full names must be unique in order to avoid the compiler and runtime engine getting confused. The convention in choosing this name is to take your Internet domain name and reverse it. For example, we would choose `ca.tutortutor` as our top-level package name because `tutortutor.ca` is our domain name. We would then specify `ca.tutortutor.graphics.shapes.Circle` to access `Circle`.

Note Reversed Internet domain names are not always valid package names. One or more of its component names might start with a digit (`6.com`), contain a hyphen (-) or other illegal character (`a-q-x.com`), or be one of Java's reserved words (`int.com`). Convention dictates that you prefix the digit with an underscore (`com._6`), replace the illegal character with an underscore (`com.a-q_x`), and suffix the reserved word with an underscore (`com.int_`).

The Package Statement

The package statement identifies the package in which a source file's types are located. This statement consists of reserved word `package`, followed by a member access operator-separated list of package and subpackage names, followed by a semicolon.

For example, `package graphics;` specifies that the source file's types locate in a package named `graphics`, and `package graphics.shapes;` specifies that the source file's types locate in the `graphics` package's `shapes` subpackage.

By convention, a package name is expressed in lowercase. When the name consists of multiple words, each word except for the first word is capitalized.

Only one package statement can appear in a source file. When it is present, nothing apart from comments must precede this statement.

Java implementations map package and subpackage names to same-named directories. For example, an implementation would map `graphics` to a directory named `graphics` and would map `graphics.shapes` to a `shapes` subdirectory of `graphics`. The Java compiler stores the class files that implement the package's types in the corresponding directory.

Note When a source file doesn't contain a package statement, the source file's types are said to belong to the *unnamed package*. This package corresponds to the current directory.

The Import Statement

Imagine having to repeatedly specify `ca.tutortutor.graphics.shapes.Circle` or some other lengthy package-qualified type name for each occurrence of that type in source code. Java provides an alternative that lets you avoid having to specify package details. This alternative is the import statement.

The import statement imports types from a package by telling the compiler where to look for unqualified type names during compilation. This statement consists of reserved word `import`, followed by a member access operator-separated list of package and subpackage names, followed by a type name or `*` (asterisk), followed by a semicolon.

The `*` symbol is a wildcard that represents all unqualified type names in the last hierarchy level. It tells the compiler to look for such names in the import statement's specified package, unless the type name is found in a previously searched package. (Using the wildcard doesn't have a performance penalty or lead to code bloat but can lead to name conflicts, as you will see.)

For example, `import ca.tutortutor.graphics.shapes.Circle;` tells the compiler that an unqualified `Circle` class exists in the `ca.tutortutor.graphics.shapes` package. Similarly, `import ca.tutortutor.graphics.shapes.*;` tells the compiler to look in this package when it encounters a `Circle` class, a `Rectangle` class, or even an `Employee` class (if `Employee` hasn't already been found).

Because Java is case sensitive, package and subpackage names specified in an import statement must be expressed in the same case as that used in the package statement.

When import statements are present in source code, only a package statement and comments can precede them.

You can run into name conflicts when using the wildcard version of the import statement because any unqualified type name matches the wildcard. For example, you have `graphics.shapes` and `geometry` packages that each contain a `Circle` class, the source code begins with `import geometry.*;` and `import graphics.shape.*;` statements, and it also contains an unqualified occurrence of `Circle`. Because the compiler doesn't know if `Circle` refers to `geometry`'s `Circle` class or `graphics.shape`'s `Circle` class, it reports an error. You can fix this problem by qualifying `Circle` with the correct package name.

Note The compiler automatically imports the `String` class and other types from the `java.lang` package, which is why it's not necessary to qualify `String` with `java.lang`.

Searching for Packages and Types

Newcomers to Java who first start to work with packages often become frustrated by “no class definition found” and other errors. This frustration can be partly avoided by understanding how the virtual machine searches for packages and types.

This section explains how the search process works. To understand this process, you need to realize that the compiler is a special Java application that runs under the control of the virtual machine. Furthermore, there are two different forms of search.

Compile-Time Search

When the compiler encounters a type expression (such as a method call) in source code, it must locate that type's declaration to verify that the expression is legal (e.g., a method exists in the type's class whose parameter types match the types of the arguments passed in the method call).

The compiler first searches the Java platform packages (which contain class library types). It then searches extension packages (for extension types). If the `-sourcepath` command-line option is specified when starting the virtual machine (via `javac`), the compiler searches the indicated path's source files.

Otherwise, the compiler searches the user classpath (in left-to-right order) for the first user class file or source file containing the type. If no user classpath is present, the current directory is searched. If no package matches or the type still cannot be found, the compiler reports an error. Otherwise, the compiler records the package information in the class file.

Note The user classpath is specified via the `-classpath` (or `-cp`) option used to start the virtual machine or, when not present, the `CLASSPATH` environment variable.

Runtime Search

When the compiler or any other Java application runs, the virtual machine will encounter types and must load their associated class files via special code known as a *classloader*. The virtual machine will use the previously stored package information that is associated with the encountered type in a search for that type's class file.

The virtual machine searches the Java platform packages, followed by extension packages, followed by the user classpath (in left-to-right order) for the first class file that contains the type. If no user classpath is present, the current directory is searched. If no package matches or the type cannot be found, a “no class definition found” error is reported. Otherwise, the class file is loaded into memory.

Note Whether you use the `-classpath/-cp` option or the `CLASSPATH` environment variable to specify a user classpath, there is a specific format that must be followed. Under Windows, this format is expressed as `path1;path2;...`, where `path1`, `path2`, and so on are the locations of package directories. Under Mac OS X, Unix, and Linux, this format changes to `path1:path2:....`

Playing with Packages

Suppose your application needs to log messages to the console, to a file, or to another destination. It can accomplish this task with the help of a logging library. My implementation of this library consists of an interface named `Logger`, an abstract class named `LoggerFactory`, and a pair of package-private classes named `Console` and `File`.

Note The logging library that we present is an example of the *Abstract Factory design pattern*, which is presented on page 87 of *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995; ISBN: 0201633612).

[Listing 5-12](#) presents the `Logger` interface, which describes objects that log messages.

Listing 5-12. Describing Objects That Log Messages via the `Logger` Interface

```
package logging;

public interface Logger {
    boolean connect();
    boolean disconnect();
    boolean log(String msg);
}
```

Each of the `connect()`, `disconnect()`, and `log()` methods returns true upon success and false upon failure. (Later in this chapter, you will discover a better technique for dealing with failure.) These methods are not declared `public` explicitly because an interface's methods are implicitly `public`.

[Listing 5-13](#) presents the `LoggerFactory` abstract class.

Listing 5-13. Obtaining a `Logger` for Logging Messages to a Specific Destination

```
package logging;

public abstract class LoggerFactory {
    public final static int CONSOLE = 0;
    public final static int FILE = 1;
```

```

public static Logger newLogger(int dstType, String... dstName) {
    switch (dstType) {
        case CONSOLE: return new Console(dstName.length == 0 ?
                                         null : dstName[0]);
        case FILE    : return new File(dstName.length == 0 ?
                                         null : dstName[0]);
        default      : return null;
    }
}
}

```

`newLogger()` returns a `Logger` object for logging messages to an appropriate destination. It uses the varargs (variable arguments) feature (see Chapter 3) to optionally accept an extra `String` argument for those destination types that require the argument. For example, `FILE` requires a file name.

Listing 5-14 presents the package-private `Console` class; this class is not accessible beyond the classes in the logging package because reserved word `class` is not preceded by reserved word `public`.

Listing 5-14. Logging Messages to the Console

```

package logging;

class Console implements Logger {
    private String dstName;

    public Console(String dstName) {
        this.dstName = dstName;
    }

    @Override
    public boolean connect() {
        return true;
    }

    @Override
    public boolean disconnect() {
        return true;
    }
}

```

```
@Override  
public boolean log(String msg) {  
    System.out.println(msg);  
    return true;  
}  
}
```

Console's package-private constructor saves its argument, which most likely will be null because there is no need for a String argument. Perhaps a future version of Console will use this argument to identify one of multiple console windows.

Listing 5-15 presents the package-private File class.

Listing 5-15. Logging Messages to a File (Eventually)

```
package logging;  
  
class File implements Logger {  
    private String dstName;  
  
    public File(String dstName) {  
        this.dstName = dstName;  
    }  
  
    @Override  
    public boolean connect() {  
        if (dstName == null)  
            return false;  
        System.out.println("opening file " + dstName);  
        return true;  
    }  
  
    @Override  
    public boolean disconnect() {  
        if (dstName == null)  
            return false;  
        System.out.println("closing file " + dstName);  
        return true;  
    }  
}
```

```

@Override
public boolean log(String msg) {
    if (dstName == null)
        return false;
    System.out.println("writing "+msg+" to file " + dstName);
    return true;
}
}

```

Unlike `Console`, `File` requires a nonnull argument. Each method first verifies that this argument is not `null`. If the argument is `null`, the method returns `false` to signify failure. (In Chapter 12, we refactor `File` to incorporate appropriate file-writing code.)

The logging library allows us to introduce portable logging code into an application. Apart from a call to `newLogger()`, this code will remain the same regardless of the logging destination. Listing 5-16 presents an application that tests this library.

Listing 5-16. Testing the Logging Library

```

import logging.Logger;
import logging.LoggerFactory;

public class TestLogger {
    public static void main(String[] args) {
        Logger logger = LoggerFactory.newLogger(LoggerFactory.CONSOLE);
        if (logger.connect()) {
            logger.log("test message #1");
            logger.disconnect();
        } else
            System.out.println("cannot connect to console-based logger");
        logger = LoggerFactory.newLogger(LoggerFactory.FILE, "x.txt");
        if (logger.connect()) {
            logger.log("test message #2");
            logger.disconnect();
        } else
            System.out.println("cannot connect to file-based logger");
        logger = LoggerFactory.newLogger(LoggerFactory.FILE);
    }
}

```

```
if (logger.connect()) {  
    logger.log("test message #3");  
    logger.disconnect();  
} else  
    System.out.println("cannot connect to file-based logger");  
}  
}
```

Follow the steps (which assume that the JDK has been installed) to create the logging package and TestLogger application, and to run this application.

1. Create a new directory and make this directory current.
2. Create a logging directory in the current directory.
3. Copy Listing 5-12 to a file named Logger.java in the logging directory.
4. Copy Listing 5-13 to a file named LoggerFactory.java in the logging directory.
5. Copy Listing 5-14 to a file named Console.java in the logging directory.
6. Copy Listing 5-15 to a file named File.java in the logging directory.
7. Copy Listing 5-16 to a file named TestLogger.java in the current directory.
8. Execute `javac TestLogger.java`, which also compiles logger's source files.
9. Execute `java TestLogger`.

After completing these steps, you should observe the following output from the TestLogger application:

```
test message #1  
opening file x.txt  
writing test message #2 to file x.txt  
closing file x.txt  
cannot connect to file-based logger
```

What happens when logging is moved to another location? For example, move logging to the root directory and run `TestLogger`. You will now observe an error message about the virtual machine not finding the logging package and its `LoggerFactory` class file.

You can solve this problem by specifying `-classpath/-cp` when running the `java` tool or by adding the location of the logging package to the `CLASSPATH` environment variable. For example, we chose to use `-classpath` (which we find more convenient) in the following Windows-specific command line:

```
java -classpath \;. TestLogger
```

The backslash represents the root directory in Windows. (We could have specified a forward slash as an alternative.) Also, the period represents the current directory. If it is missing, the virtual machine complains about not finding the `TestLogger` class file.

Tip If you discover an error message where the virtual machine reports that it cannot find an application class file, try appending a period character to the `classpath`. Doing so will probably fix the problem.

Packages and JAR Files

The JDK provides a `jar` tool that is used to archive class files in *JAR* (Java ARchive) files and is also used to extract a JAR file's class files. It probably comes as no surprise that you can store packages in JAR files, which greatly simplify the distribution of your package-based class libraries.

To show you how easy it is to store a package in a JAR file, you will create a `logger.jar` file that contains the logging package's four class files (`Logger.class`, `LoggerFactory.class`, `Console.class`, and `File.class`). Complete the following steps to accomplish this task:

1. Make sure that the current directory contains the previously created logging directory with its four class files.
2. Execute the following command:

```
jar cf logger.jar logging\*.class
```

The `c` option stands for “create new archive” and the `f` option stands for “specify archive file name”.

You could alternatively execute the following command:

```
jar cf logger.jar logging/*.class
```

You should now find a `logger.jar` file in the current directory. To prove to yourself that this file contains the four class files, execute the following command, where the `t` option stands for “list table of contents”:

```
jar tf logger.jar
```

You can run `TestLogger.class` by adding `logger.jar` to the classpath. For example, you can run `TestLogger` under Windows via the following command:

```
java -classpath logger.jar;. TestLogger
```

Note Although you can create your own logging framework, doing so is a waste of time. Instead, you should leverage the `java.util.logging` package that's included in the standard class library, or any of the other logging frameworks you can find in the Internet (like Log4j).

Mastering Static Imports

We already know that we can omit the package names if we import them. But we can even go one step further and access static constant fields and static methods without even using the class name. This static imports feature lets you import a class's static members so that you don't have to qualify them with their class names. It's implemented via a small modification to the import statement, as follows:

```
import static packagespec . classname . ( staticmembername | * );
```

The static import statement specifies `static` after `import`. It then specifies a member access operator-separated list of package and subpackage names, which is followed by the member access operator and a class's name. Once again, the member access operator is specified, followed by a single static member name or the asterisk wildcard.

You specify a single static member name to import only that name:

```
import static java.lang.Math.PI; // Import the PI static field only.  
import static java.lang.Math.cos; // Import the cos() static method only.
```

In contrast, you specify the wildcard to import all static member names:

```
import static java.lang.Math.*; // Import all static members from Math.
```

You can now refer to the static member(s) without having to specify the class name:

```
System.out.println(cos(PI));
```

Using multiple static import statements can result in name conflicts, which causes the compiler to report errors. For example, suppose your geom package contains a Circle class with a static member named PI. Now suppose you specify `import static java.lang.Math.*;` and `import static geom.Circle.*;` at the top of your source file. Finally, suppose you specify `System.out.println(PI);` somewhere in that file's code. The compiler reports an error because it doesn't know whether PI belongs to Math or to Circle.

Caution Overuse of static imports can make your code unreadable and unmaintainable. Anyone reading your code could have a hard time finding out which class a static member comes from, especially when importing all static member names from a class. Also, static imports pollute the code's namespace with all of the static members you import. Eventually, you may run into name conflicts that are hard to resolve.

Mastering Exceptions

In an ideal world, nothing bad ever happens when an application runs. For example, a file always exists when the application needs to open the file, the application is always able to connect to a remote computer, and the virtual machine never runs out of memory when the application needs to instantiate objects.

In contrast, real-world applications occasionally attempt to open files that don't exist, attempt to connect to remote computers that are unable to communicate with them, and require more memory than the virtual machine can provide. Your goal is to write code that properly responds to these and other exceptional situations (exceptions).

This section introduces you to exceptions. After defining this term, we will look at representing exceptions in source code. We will then examine the topics of throwing and handling exceptions and conclude by discussing how to perform cleanup tasks before a method returns, whether or not an exception has been thrown.

What Are Exceptions?

An *exception* is a divergence from an application's normal behavior. For example, the application attempts to open a nonexistent file for reading. The normal behavior is to successfully open the file and begin reading its contents. However, the file cannot be read when the file doesn't exist.

This example illustrates an exception that cannot be prevented. However, a workaround is possible. For example, the application can detect that the file doesn't exist and take an alternate course of action, which might include telling the user about the problem. Unpreventable exceptions where workarounds are possible must not be ignored.

Exceptions can occur because of poorly written code. For example, an application might contain code that accesses each element in an array. Because of careless oversight, the array-access code might attempt to access a nonexistent array element, which leads to an exception. This kind of exception is preventable by writing correct code.

Finally, an exception might occur that cannot be prevented and for which there is no workaround. For example, the virtual machine might run out of memory, or perhaps it cannot find a class file. This kind of exception, known as an *error*, is so serious that it's impossible (or at least inadvisable) to work around; the application must terminate, presenting a message to the user that explains why it's terminating.

Representing Exceptions in Source Code

An exception can be represented via error codes or objects. After discussing each kind of representation and explaining why objects are superior, we will introduce you to Java's exception and error class hierarchy, emphasizing the difference between checked and runtime exceptions. We will close our discussion on representing exceptions in source code by discussing custom exception classes.

Error Codes vs. Objects

One way to represent exceptions in source code is to use error codes. For example, a method might return true on success and false when an exception occurs. Alternatively, a method might return 0 on success and a nonzero integer value that identifies a specific kind of exception.

Developers traditionally designed methods to return error codes; we demonstrated this tradition in each of the three methods in Listing 5-12's Logger interface. Each method returns true on success or returns false to represent an exception (e.g., unable to connect to the logger).

Although a method's return value must be examined to see if it represents an exception, error codes are all too easy to ignore. For example, a lazy developer might ignore the return code from Logger's `connect()` method and attempt to call `log()`. Ignoring error codes is one reason why a new approach to dealing with exceptions has been invented.

This new approach is based on objects. When an exception occurs, an object representing the exception is created by the code that was running when the exception occurred. Details describing the exception's surrounding context are stored in the object. These details are later examined to work around the exception.

The object is then *thrown* or handed off to the virtual machine to search for a *handler*, code that can handle the exception. (If the exception is an error, the application should not provide a handler because errors are so serious, such as the virtual machine has run out of memory, that there's practically nothing that can be done about them.) When a handler is located, its code is executed to provide a workaround. Otherwise, the virtual machine terminates the application.

Caution Code that handles exceptions can be a source of bugs because it's often not thoroughly tested. Always make sure to test any code that handles exceptions.

Apart from being too easy to ignore, an error code's Boolean or integer value is less meaningful than an object name. For example, `FileNotFoundException` is self-evident, but what does `false` mean? Also, an object can contain information about what led to the exception. These details can be helpful to a suitable workaround.

The Throwable Class Hierarchy

Java provides a hierarchy of classes that represent different kinds of exceptions. These classes are rooted in `java.lang.Throwable`, the ultimate superclass for all *throwables* (exception and error objects—exceptions and errors, for short—that can be thrown). Figure 5-1 reveals `Throwable` and its immediate subclasses.

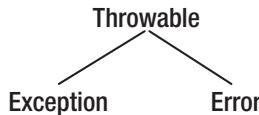


Figure 5-1. The exceptions hierarchy is rooted in the `Throwable` class

`Exception` is the root class for all exception-oriented `throwables`. Similarly, `Error` is the root class for all error-oriented `throwables`.

It's not uncommon for a class's public methods to call helper methods that throw various exceptions. A public method will probably not document exceptions thrown from a helper method because they are implementation details that often should not be visible to the public method's caller.

However, because this exception might be helpful in diagnosing the problem, the public method can wrap the lower-level exception in a higher-level exception that is documented in the public method's contract interface. The wrapped exception is known as a *cause* because its existence causes the higher-level exception to be thrown.

A cause can be specified in most constructors of throwable classes or subclasses, and you can also invoke the `throwable.initCause(Throwable cause)` method, but at most once (during construction or immediately after construction).

Unless you use a logging framework, a common usage scenario for using `Throwable`, or more precise, exception classes, is:

```

try {
    // some code which might throw exceptions
} catch(Exception e) {
    e.printStackTrace(System.err);
}
  
```

We talk about the details of this construct in the following text.

Java has a sophisticated built-in exception hierarchy you can use for your application, apart from defining your own exception classes. Have a look at the API documentation to learn about them.

Caution Never instantiate throwable, exception, or error. The resulting objects are meaningless because they are too generic: checked exceptions vs. runtime exceptions.

A *checked exception* is an exception that represents a problem with the possibility of recovery and for which the developer must provide a workaround. The compiler checks the code to ensure that the exception is handled in the method where it is thrown, or is explicitly identified as being handled elsewhere.

Exception and all subclasses except for `java.lang.RuntimeException` (and its subclasses) describe checked exceptions. For example, the `CloneNotSupportedException` and `IOException` classes describe checked exceptions. (`CloneNotSupportedException` should not be checked because there is no runtime workaround for this kind of exception.)

A *runtime exception* is an exception that represents a coding mistake. This kind of exception is also known as an *unchecked exception* because it doesn't need to be handled or explicitly identified—the mistake must be fixed. Because these exceptions can occur in many places, it would be burdensome to be forced to handle them.

`RuntimeException` and its subclasses describe unchecked exceptions. For example, `java.lang.ArithmaticException` describes arithmetic problems such as integer division by zero. Another example is `java.lang.ArrayIndexOutOfBoundsException`, which is thrown when you attempt to access an array element with a negative index or an index that is greater than or equal to the length of the array. (In hindsight, `RuntimeException` should have been named `UncheckedException` because all exceptions occur at runtime.)

Note Many developers are unhappy with checked exceptions because of the work involved in having to handle them. This problem is made worse by libraries providing methods that throw checked exceptions when they should throw unchecked exceptions. As a result, many modern languages support only unchecked exceptions.

Custom Exception Classes

You can declare your own exception classes. Before doing so, ask yourself if an existing exception class in the standard class library meets your needs. If you find a suitable class, you should reuse it. (Why reinvent the wheel?) Other developers will already be familiar with the existing class, and this knowledge will make your code easier to learn. When no existing class meets your needs, think about whether to subclass `Exception` or `RuntimeException`. In other words, will your exception class be checked or unchecked? As a rule of thumb, your class should subclass `RuntimeException` if you think that it will describe a coding mistake.

Tip When you name your class, follow the convention of providing an `Exception` suffix. This suffix clarifies that your class describes an exception.

Suppose you are creating a `Media` class whose static methods are to perform media-oriented utility tasks. For example, one method converts sound files in non-MP3 media formats to MP3 format. This method will be passed source file and destination file arguments and will convert the source file to the format implied by the destination file's extension.

Before performing the conversion, the method needs to verify that the source file's format agrees with the format implied by its file extension. If there is no agreement, an exception must be thrown. Furthermore, this exception must store the expected and existing media formats so that a handler can identify them when presenting a message to the user.

Because Java's class library doesn't provide a suitable exception class, you decide to introduce a class named `InvalidMediaFormatException`. Detecting an invalid media format is not the result of a coding mistake, and so you also decide to extend `Exception` to indicate that the exception is checked. Listing 5-17 presents this class's declaration.

Listing 5-17. Declaring a Custom Exception Class

```
package media;

public class InvalidMediaFormatException extends Exception {
    private String expectedFormat;
    private String existingFormat;
```

```
public InvalidMediaFormatException(String expectedFormat,
                                  String existingFormat) {
    super("Expected format: " + expectedFormat + ", Existing format: " +
          existingFormat);
    this.expectedFormat = expectedFormat;
    this.existingFormat = existingFormat;
}

public String getExpectedFormat() {
    return expectedFormat;
}

public String getExistingFormat() {
    return existingFormat;
}
}
```

`InvalidMediaFormatException` provides a constructor that calls `Exception`'s `public Exception(String message)` constructor with a detail message that includes the expected and existing formats. It is wise to capture such details in the detail message because the problem that led to the exception might be hard to reproduce.

`InvalidMediaFormatException` also provides `getExpectedFormat()` and `getExistingFormat()` methods that return these formats. Perhaps a handler will present this information in a message to the user. Unlike the detail message, this message might be *localized*, expressed in the user's language (French, German, English, and so on).

Throwing Exceptions

Now that you have created an `InvalidMediaFormatException` class, you can declare the `Media` class and begin to code its `convert()` method. The initial version of this method validates its arguments and then verifies that the source file's media format agrees with the format implied by its file extension. Check out Listing 5-18.

Listing 5-18. Throwing Exceptions from the convert() Method

```

package media;

import java.io.IOException;

public class Media {
    public static void convert(String srcName, String dstName)
        throws InvalidMediaFormatException, IOException {
        if (srcName == null)
            throw new NullPointerException(srcName + " is null");
        if (dstName == null)
            throw new NullPointerException(dstName + " is null");
        // Code to access source file and verify that its format matches the
        // format implied by its file extension.
        //
        // Assume that the source file's extension is RM (for Real Media) and
        // that the file's internal signature suggests that its format is
        // Microsoft WAVE.
        String expectedFormat = "RM";
        String existingFormat = "WAVE";
        throw new InvalidMediaFormatException(expectedFormat, existingFormat);
    }
}

```

Media's convert() method appends throws InvalidMediaFormatException, IOException to its header. A *throws clause* identifies all checked exceptions that are thrown out of the method and must be handled by some other method. It consists of reserved word throws followed by a comma-separated list of checked exception class names and is always appended to a method header. The convert() method's throws clause indicates that this method is capable of throwing an InvalidMediaException or IOException instance to the virtual machine.

convert() also demonstrates the throw statement, which consists of reserved word throw followed by an instance of Throwable or a subclass. (You will typically instantiate an Exception subclass.) This statement throws the instance to the virtual machine, which then searches for a suitable handler to handle the exception.

The first use of the throw statement is to throw a `java.lang.NullPointerException` instance when a null reference is passed as the source or destination file name. This unchecked exception is commonly thrown to indicate that a contract has been violated via a passed null reference. For example, you cannot pass null file names to `convert()`.

The second use of the throw statement is to throw a `media.InvalidMediaFormatException` instance when the expected media format doesn't match the existing format. In the contrived example, the exception is thrown because the expected format is RM and the existing format is WAVE.

Unlike `InvalidMediaFormatException`, `NullPointerException` is not listed in `convert()`'s throws clause because `NullPointerException` instances are unchecked. They can occur so frequently that it is too big a burden to force the developer to properly handle these exceptions. Instead, the developer should write code that minimizes their occurrences.

Although not thrown from `convert()`, `IOException` is listed in this method's throws clause in preparation for refactoring this method to perform the conversion with the help of file-handling code.

There are a few additional items to keep in mind when working with throws clauses and throw statements:

- You can append a throws clause to a constructor and throw an exception from the constructor when something goes wrong while the constructor is executing. The resulting object will not be created.
- When an exception is thrown out of an application's `main()` method, the virtual machine terminates the application and calls the exception's `printStackTrace()` method to print, to the console, the sequence of nested method calls that was awaiting completion when the exception was thrown.
- If a superclass method declares a throws clause, the overriding subclass method doesn't have to declare a throws clause. However, if the subclass method does declare a throws clause, the clause must not include the names of checked exception classes that are not also included in the superclass method's throws clause, unless they are the names of exception subclasses. For example, given superclass method `void foo() throws IOException {}`, the overriding subclass method could be declared as `void foo()`

{}, void foo() throws IOException {}, or void foo() throws FileNotFoundException {}; the java.io.FileNotFoundException class subclasses IOException.

- A checked exception class name doesn't need to appear in a throws clause when the name of its superclass appears.
- The compiler reports an error when a method throws a checked exception and doesn't also handle the exception or list the exception in its throws clause.
- If at all possible, don't include the names of unchecked exception classes in a throws clause. These names are not required because such exceptions should never occur. Furthermore, they only clutter source code and possibly confuse someone who is trying to understand that code.
- You can declare a checked exception class name in a method's throws clause without throwing an instance of this class from the method. (Perhaps the method has yet to be fully coded.) However, Java requires that you provide code to handle this exception, even though it is not thrown.

Handling Exceptions

A method indicates its intention to handle one or more exceptions by specifying a try statement that includes one or more appropriate catch blocks. The try statement consists of reserved word try followed by a brace-delimited body. You place code that throws exceptions into this block.

A catch block consists of reserved word catch, followed by a round bracket-delimited single-parameter list that specifies an exception class name, followed by a brace-delimited body. You place code that handles exceptions whose types match the type of the catch block's parameter list's exception class parameter in this block.

A catch block is specified immediately after a try block. When an exception is thrown, the virtual machine will search for a handler. It first examines the catch block to see whether its parameter type matches or is the superclass type of the exception that has been thrown.

If the catch block is found, its body executes and the exception is handled. Otherwise, the virtual machine proceeds up the method-call stack, looking for the first method whose try statement contains an appropriate catch block. This process continues unless a catch block is found or execution leaves the `main()` method.

The following example illustrates try and catch:

```
try {  
    int x = 1 / 0;  
} catch (ArithmetricException ae) {  
    System.out.println("attempt to divide by zero");  
}
```

When execution enters the try block, an attempt is made to divide integer 1 by integer 0. The virtual machine responds by instantiating `ArithmetricException` and throwing this exception. It then detects the catch block, which is capable of handling thrown `ArithmetricException` objects, and transfers execution to this block, which invokes `System.out.println()` to output a suitable message; the exception is handled.

Because `ArithmetricException` is an example of an unchecked exception type, and because unchecked exceptions represent coding mistakes that must be fixed, you typically don't catch them, as demonstrated previously. Instead, you would fix the problem that led to the thrown exception.

Tip You might want to name your catch block parameters using the abbreviated style shown in the preceding section. Not only does this convention result in more meaningful exception-oriented parameter names (ae indicates that an `ArithmetricException` has been thrown), it can help reduce compiler errors. For example, it is common practice to name a catch block's parameter e, for convenience. (Why type a long name?) However, the compiler will report an error when a previously declared local variable or parameter also uses e as its name; multiple same-named local variables and parameters cannot exist in the same scope.

Handling Multiple Exception Types

You can specify multiple catch blocks after a try block. For example, Listing 5-18's `convert()` method specifies a `throws` clause indicating that `convert()` can throw `InvalidMediaFormatException`, which is currently thrown, and `IOException`, which will be thrown when `convert()` is refactored. This refactoring will result in `convert()` throwing `IOException` when it cannot read from the source file or write to the destination file and throwing `FileNotFoundException` (a subclass of `IOException`) when it cannot open the source file or create the destination file. All these exceptions must be handled, as demonstrated in Listing 5-19.

Listing 5-19. Handling Different Kinds of Exceptions

```
import java.io.FileNotFoundException;
import java.io.IOException;

import media.InvalidMediaFormatException;
import media.Media;

public class Converter {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Converter srcfile dstfile");
            return;
        }
        try {
            Media.convert(args[0], args[1]);
        } catch (InvalidMediaFormatException imfe) {
            System.out.println("Unable to convert " + args[0] + " to " + args[1]);
            System.out.println("Expecting " + args[0] + " to conform to " +
                               imfe.getExpectedFormat() + " format.");
            System.out.println("However, " + args[0] + " conformed to " +
                               imfe.getExistingFormat() + " format.");
        } catch (FileNotFoundException fnfe) {
            fnfe.printStackTrace(System.err);
        }
    }
}
```

```

        } catch (IOException ioe){
            ioe.printStackTrace(System.err);
        }
    }
}

```

The call to Media's convert() method in Listing 5-19 is placed in a try block because this method is capable of throwing an instance of the checked InvalidFormatException, IOException, or FileNotFoundException class; checked exceptions must be handled or be declared to be thrown via a throws clause that is appended to the method.

The catch (InvalidFormatException imfe) block's statements are designed to provide a descriptive error message to the user. A more sophisticated application would *localize* these names so that the user could read the message in the user's language. The developer-oriented detail message is not output because it is not necessary in this trivial application.

Note A developer-oriented detail message is typically not localized. Instead, it is expressed in the developer's language. Users should never see detail messages.

Although not thrown, a catch block for IOException is required because this checked exception type appears in convert()'s throws clause. Because the catch (IOException ioe) block can also handle thrown FileNotFoundException instances (because FileNotFoundException subclasses IOException), the catch (FileNotFoundException fnfe) block isn't necessary at this point but is present to separate out the handling of a situation where a file cannot be opened for reading or created for writing (which will be addressed once convert() is refactored to include file code).

Assuming that the current directory contains Listing 5-19 and a media subdirectory containing InvalidFormatException.java and Media.java, compile this listing (javac Converter.java), which also compiles media's source files, and run the application, as in java Converter A B. Converter responds by presenting the following output:

```

Unable to convert A to B
Expecting A to conform to RM format.
However, A conformed to WAVE format.

```

Listing 5-19's `FileNotFoundException` and `IOException` catch blocks just output the exception to the console. You could be tempted to leave them empty, but this is almost never a good idea. Unless you have a really good reason, don't create an empty catch block. It swallows exceptions and you don't know that the exceptions were thrown. (For brevity, we don't always code catch blocks in this book's examples.)

Caution The compiler reports an error when you specify two or more catch blocks with the same parameter type after a try body. Example: `try {} catch (IOException ioe1) {} catch (IOException ioe2) {}`. You must merge these catch blocks into one block.

Although you can write catch blocks in any order, the compiler restricts this order when one catch block's parameter is a supertype of another catch block's parameter. The subtype parameter catch block must precede the supertype parameter catch block; otherwise, the subtype parameter catch block will never be executed.

For example, the `FileNotFoundException` catch block must precede the `IOException` catch block. If the compiler allowed the `IOException` catch block to be specified first, the `FileNotFoundException` catch block would never execute because a `FileNotFoundException` instance is also an instance of its `IOException` superclass.

There is also a variant where you can catch several exceptions with just one catch clause. Just separate the types with a bar, as in

```
catch( IOException | SQLException e) {  
    ...  
}
```

This can be done with three and more types as well.

Rethrowing Exceptions

While discussing the `Throwable` class, we discussed wrapping lower-level exceptions in higher-level exceptions. This activity will typically take place in a catch block and is illustrated in the following example:

```
catch (IOException ioe) {  
    throw new ReportCreationException(ioe);  
}
```

This example assumes that a helper method has just thrown a generic `IOException` instance as the result of trying to create a report. The public method's contract states that `ReportCreationException` is thrown in this case. To satisfy the contract, the latter exception is thrown. To satisfy the developer who is responsible for debugging a faulty application, the `IOException` instance is wrapped inside the `ReportCreationException` instance that is thrown to the public method's caller.

Sometimes, a catch block might not be able to fully handle an exception. Perhaps it needs access to information provided by some ancestor method in the method-call stack. However, the catch block might be able to partly handle the exception. In this case, it should partly handle the exception and then rethrow the exception so that a handler in an ancestor method can finish handling it. Another possibility is to log the exception (for later analysis), which is demonstrated in the following example:

```
catch (FileNotFoundException fnfe) {  
    logger.log(fnfe);  
    throw fnfe; // Rethrow the exception here.  
}
```

Performing Cleanup

In some situations, you might want to execute cleanup code before execution leaves a method following a thrown exception. For example, you might want to close a file that was opened, but could not be written, possibly because of insufficient disk space. Java provides the `finally` block for this situation.

The `finally` block consists of reserved word `finally` followed by a body, which provides the cleanup code. A `finally` block follows either a catch block or a try block. In the former case, the exception may be handled (and possibly rethrown) before `finally` executes. In the latter case, the exception is handled (and possibly rethrown) after `finally` executes.

Listing 5-20 demonstrates the first scenario in the context of a simulated file-copying application's `main()` method.

Listing 5-20. Cleaning Up by Closing Files After Handling a Thrown Exception

```
import java.io.IOException;

public class Copy {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Copy srcFile dstFile");
            return;
        }

        int fileHandleSrc = 0;
        int fileHandleDst = 1;
        try {
            fileHandleSrc = open(args[0]);
            fileHandleDst = create(args[1]);
            copy(fileHandleSrc, fileHandleDst);
        } catch (IOException ioe) {
            System.err.println("I/O error: " + ioe.getMessage());
            return;
        } finally {
            close(fileHandleSrc);
            close(fileHandleDst);
        }
    }

    static public int open(String filename) {
        return 1; // Assume that filename is mapped to integer.
    }

    static public int create(String filename) {
        return 2; // Assume that filename is mapped to integer.
    }

    static public void close(int fileHandle) {
        System.out.println("closing file: " + fileHandle);
    }
}
```

```

static public void copy(int fileHandleSrc, int fileHandleDst)
    throws IOException {
    System.out.println("copying file " + fileHandleSrc + " to file " +
        fileHandleDst);
    if (Math.random() < 0.5)
        throw new IOException("unable to copy file");
}
}

```

Listing 5-20 presents a Copy application class that simulates the copying of bytes from a source file to a destination file. The try block invokes the `open()` method to open the source file and the `create()` method to create the destination file. Each method returns an integer-based *file handle* that uniquely identifies the file.

Next, this block calls the `copy()` method to perform the copy. After outputting a suitable message, `copy()` invokes the `Math` class's `random()` method (officially discussed in Chapter 7) to return a random number between 0 and 1. When this method returns a value less than 0.5, which simulates a problem (perhaps the disk is full), the `IOException` class is instantiated and this instance is thrown.

The virtual machine locates the catch block that follows the try block and causes its handler to execute, which outputs a message. Then, the code in the finally block that follows the catch block is allowed to execute. Its purpose is to close both files by invoking the `close()` method on the passed file handle.

Compile this source code (`javac Copy.java`) and run the application with two arbitrary arguments (`java Copy x.txt x.bak`). You should observe the following output when there is no problem:

```

copying file 1 to file 2
closing file: 1
closing file: 2

```

When something goes wrong, you should observe the following output:

```

copying file 1 to file 2
I/O error: unable to copy file
closing file: 1
closing file: 2

```

Whether or not an I/O error occurs, notice that the finally block is the final code to execute. The finally block executes even though the catch block ends with a return statement.

It is also allowed to have a finally block without any catch block. In this case the finally block gets executed and the exception gets forwarded to external exception handling (surrounding try-catch or the method being declared to throwing exceptions)5-.

Automatic Resource Management

Having to make sure resources like input and output streams get properly closed at the end of their usage as in

```
FileInputStream fis = null;
try {
    fis = new FileInputStream(new File("someFile.txt"));
    ...
} catch(Exception e) {
    e.printStackTrace(System.err);
} finally {
    if(fis != null) {
        try {
            fis.close();
        }catch(Exception e2) {
        }
    }
}
```

is a rather tedious task, with all that finally clauses and null checks. That is why with JDK version 7 the automatic resource management was invented, or “try-with-resources”. You can now write

```
try( FileInputStream fis = new FileInputStream(new File("someFile.txt")) )
{
    ...
} catch(Exception e) {
    e.printStackTrace(System.err);
}
```

which significantly improves readability. Because of the stream being defined inside the try-clause, Java will automatically close the resource at the end of the block. You neither have to initialize the stream outside the block, nor is it necessary to explicitly call a `close()` on the resource.

It is also possible to initialize two or more resources inside the try-clause—just use a semicolon as a separator.

This syntax works with all resources (streams and other things) which implement the `java.lang.AutoCloseable` interface. And you can of course define your own autocloseable classes by implementing this interface.

EXERCISES

The following exercises are designed to test your understanding of Chapter 5's content:

1. What is a nested class?
2. Identify the four kinds of nested classes.
3. Which nested classes are also known as inner classes?
4. True or false: A static member class has an enclosing instance.
5. How do you instantiate a nonstatic member class from beyond its enclosing class?
6. When is it necessary to declare local variables and parameters `final`?
7. True or false: An interface can be declared within a class or within another interface.
8. Define package.
9. How do you ensure that package names are unique?
10. What is a package statement?
11. True or false: You can specify multiple package statements in a source file.
12. What is an import statement?

13. How do you indicate that you want to import multiple types via a single import statement?
14. During a runtime search, what happens when the virtual machine cannot find a class file?
15. How do you specify the user classpath to the virtual machine?
16. What is a static import statement?
17. How do you specify a static import statement?
18. What is an exception?
19. In what ways are objects superior to error codes for representing exceptions?
20. What is a throwable?
21. What is the difference between `Exception` and `Error`?
22. What is a checked exception?
23. What is a runtime exception?
24. Under what circumstance would you introduce your own exception class?
25. True or false: You use a `throw` statement to identify exceptions that are thrown from a method by appending this statement to a method's header.
26. What is the purpose of a `try` statement, and what is the purpose of a `catch` block?
27. What is the purpose of a `finally` block?
28. A 2D graphics package supports two-dimensional drawing and transformations (rotation, scaling, translation, etc.). These transformations require a 3-by-3 *matrix* (a table). Declare a `G2D` class that encloses a private `Matrix` nonstatic member class. Instantiate `Matrix` within `G2D`'s noargument constructor, and initialize the `Matrix` instance to the *identity matrix* (a matrix where all entries are 0 except for those on the upper-left to lower-right diagonal, which are 1).
29. Extend the logging package to support a null device in which messages are thrown away.

30. Modify the logging package so that Logger's connect() method throws CannotConnectException when it cannot connect to its logging destination, and the other two methods each throw NotConnectedException when connect() was not called or when it threw CannotConnectException.
 31. Modify TestLogger to respond appropriately to thrown CannotConnectException and NotConnectedException objects.
-

Summary

Classes that are declared outside of any class are known as top-level classes. Java also supports nested classes, which are classes that are declared as members of other classes or scopes.

There are four kinds of nested classes: static member classes, nonstatic member classes, anonymous classes, and local classes. The latter three categories are known as inner classes.

Java supports the partitioning of top-level types into multiple namespaces, to better organize these types and to also prevent name conflicts. Java uses packages to accomplish these tasks.

The package statement identifies the package in which a source file's types are located. The import statement imports types from a package by telling the compiler where to look for unqualified type names during compilation.

An exception is a divergence from an application's normal behavior. Although it can be represented by an error code or object, Java uses objects because error codes are meaningless and cannot contain information about what led to the exception.

Java provides a hierarchy of classes that represent different kinds of exceptions. These classes are rooted in `Throwable`. Moving down the throwable hierarchy, you encounter the `Exception` and `Error` classes, which represent nonerror exceptions and errors.

`Exception` and its subclasses, except for `RuntimeException` (and its subclasses), describe checked exceptions. They are checked because you must check the code to ensure that an exception is handled where thrown or identified as being handled elsewhere.

`RuntimeException` and its subclasses describe unchecked exceptions. You don't have to handle these exceptions because they represent coding mistakes (fix the mistakes). Although the names of their classes can appear in throws clauses, doing so adds clutter.

The throw statement throws an exception to the virtual machine, which searches for an appropriate handler. When the exception is checked, its name must appear in the method's throws clause, unless the name of the exception's superclass is listed in this clause.

A method handles one or more exceptions by specifying a try statement and appropriate catch blocks. A finally block can be included to execute cleanup code whether an exception is thrown or not, and before a thrown exception leaves the method.

Using the try-with-resources syntax, it is possible to have resources automatically closed (without explicitly calling `close()` on the resources) at the end of a try-catch block.

Chapter 6 continues to explore the Java language by focusing on assertions, annotations, generics, and enums.

CHAPTER 6

Mastering Advanced Language Features, Part 2

In Chapters 2 through 4, we laid a foundation for learning the Java language, and in Chapter 5, we built onto this foundation by introducing some of Java's more advanced language features. In this chapter, we will continue to cover advanced language features by focusing on those features related to annotations, generics, and enums.

Mastering Annotations

While developing a Java application, you might want to *annotate* (associate *metadata*, which is data that describes other data, with) various application elements. For example, you might want to identify methods that are not fully implemented so that you will not forget to implement them. Java's annotations language feature lets you accomplish this task.

In this section we will introduce you to annotations. After defining this term and presenting three kinds of compiler-supported annotations as examples, we will show you how to declare your own annotation types and use these types to annotate source code. Finally, you will discover how to process your own annotations to accomplish useful tasks.

Discovering Annotations

An *annotation* is an instance of an annotation type and associates metadata with an application element. It is expressed in source code by prefixing the type name with the @ symbol. For example, @Readonly is an annotation and Readonly is its type.

Note You can use annotations to associate metadata with constructors, fields, local variables, methods, packages, parameters, and types (annotation, class, enum, and interface).

The compiler supports the `Override`, `Deprecated`, and `SuppressWarnings` annotation types. These types are located in the `java.lang` package.

`@Override` annotations are useful for expressing that a subclass method overrides a method in the superclass and doesn't overload that method instead. The following example reveals this annotation being used to prefix the overriding method:

```
@Override  
public void draw(int color) {  
    // drawing code  
}
```

`@Deprecated` annotations are useful for indicating that the marked application element is *deprecated* (phased out) and should no longer be used. The compiler warns you when a deprecated application element is accessed by nondeprecated code.

```
@Deprecated  
public Date(int year, int month, int date) {  
    this(year, month, date, 0, 0, 0);  
}
```

This example excerpts one of the constructors in Java's `Date` class (located in the `java.util` package). Its Javadoc comment reveals that `Date(int year, int month, int date)` has been deprecated in favor of using the `set()` method in the `Calendar` class (also located in the `java.util` package).

The compiler suppresses warnings when a compilation unit (typically a class or interface) refers to a deprecated class, method, or field inside the very same unit. This feature lets you modify legacy APIs without generating deprecation warnings and is demonstrated in Listing 6-1.

Listing 6-1. Referencing a Deprecated Field from the Same Class Declaration

```
public class Employee {
    /**
     * Employee's name
     * @deprecated New version uses firstName and lastName fields.
     */
    @Deprecated
    String name;
    String firstName;
    String lastName;

    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.name = "John Doe";
    }
}
```

Listing 6-1 declares an `Employee` class with a `name` field that has been deprecated. Although `Employee`'s `main()` method refers to `name`, the compiler will suppress a deprecation warning because the deprecation and reference occur in the same class.

Suppose you refactor this listing by introducing a new `UseEmployee` class and moving `Employee`'s `main()` method to this class. *Listing 6-2* presents the resulting class structure.

Listing 6-2. Referencing a Deprecated Field from Another Class Declaration

```
public class Employee {
    /**
     * Employee's name
     * @deprecated New version uses firstName and lastName fields.
     *           <= do not confuse, this is for javadoc
     */
    @Deprecated
    String name;
    String firstName;
    String lastName;
}
```

```
public class UseEmployee {
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.name = "John Doe";
    }
}
```

If you attempt to compile this source code via the `javac` compiler tool, you will discover the following messages:

Note: `UseEmployee.java` uses or overrides a deprecated API.
Note: Recompile with `-Xlint:deprecation` for details.

You will need to specify `-Xlint:deprecation` as one of `javac`'s command-line arguments (as in `javac -Xlint:deprecation UseEmployee.java`) to discover the deprecated item and the code that refers to this item:

```
Employee.java:18: warning: [deprecation] name in Employee has been deprecated
    emp.name = "John Doe";
               ^
1 warning
```

`@SuppressWarnings` annotations are useful for suppressing deprecation or unchecked warnings via a "deprecation" or an "unchecked" argument. (Unchecked warnings occur when mixing code that uses generics with pre-generics legacy code. We will discuss generics and unchecked warnings later in this chapter.)

For example, Listing 6-3 uses `@SuppressWarnings` with a "deprecation" argument to suppress the compiler's deprecation warnings when code in the `UseEmployee` class's `main()` method accesses the `Employee` class's `name` field.

Listing 6-3. Suppressing the Previous Deprecation Warning

```
public class UseEmployee {
    @SuppressWarnings("deprecation")
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.name = "John Doe";
    }
}
```

Note As a matter of style, you should always specify `@SuppressWarnings` on the most deeply nested element where it is effective. For example, if you want to suppress a warning in a particular method, you should annotate that method rather than its class.

Declaring Annotation Types and Annotating Source Code

Before you can annotate source code, you need annotation types that can be instantiated. Java supplies many annotation types in addition to `Override`, `Deprecated`, and `SuppressWarnings`. Java also lets you declare your own types.

You declare an annotation type by specifying the `@` symbol, immediately followed by reserved word `interface`, followed by the type's name, followed by a body. For example, Listing 6-4 uses `@interface` to declare an annotation type named `Stub`.

Listing 6-4. Declaring the Stub Annotation Type

```
public @interface Stub {  
}
```

Instances of annotation types that supply no data apart from a name—their bodies are empty—are known as *marker annotations* because they mark application elements for some purpose. As Listing 6-5 reveals, `@Stub` is used to mark empty methods (stubs).

Listing 6-5. Annotating a Stubbed-Out Method

```
public class Deck // Describes a deck of cards.{  
    @Stub  
    public void shuffle() {  
        // This method is empty and will presumably be filled in with  
        // appropriate  
        // code at some later date.  
    }  
}
```

Listing 6-5's Deck class declares an empty shuffle() method. This fact is indicated by instantiating Stub and prefixing shuffle()'s method header with the resulting @Stub annotation.

Note Although marker interfaces (introduced in Chapter 4) appear to have been replaced by marker annotations, this is not the case, because marker interfaces have advantages over marker annotations. One advantage is that a marker interface specifies a type that is implemented by a marked class, which lets you catch problems at compile time. For example, when a class doesn't implement the Cloneable interface, its instances cannot be shallowly cloned via Object's clone() method. If Cloneable had been implemented as a marker annotation, this problem would not be detected until runtime.

Although marker annotations are useful (@Override and @Deprecated are good examples), you will typically want to enhance an annotation type so that you can store metadata via its instances. You accomplish this task by adding elements to the type.

An *element* is a method header that appears in the annotation type's body. It cannot have parameters or a throws clause, and its return type must be a primitive type (such as int, java.lang.String, java.lang.Class), an enum (discussed later in this chapter), an annotation type, or an array of the preceding types. However, it can have a default value.

Listing 6-6 adds three elements to Stub.

Listing 6-6. Adding Three Elements to the Stub Annotation Type

```
public @interface Stub {  
    int id(); // A semicolon must terminate an element declaration.  
    String dueDate();  
    String developer() default "unassigned";  
}
```

The id() element specifies a 32-bit integer that identifies the stub. The dueDate() element specifies a String-based date that identifies when the method stub is to be implemented. Finally, developer() specifies the String-based name of the developer responsible for coding the method stub.

Unlike `id()` and `dueDate()`, `developer()` is declared with a default value, "unassigned". When you instantiate `Stub` and don't assign a value to `developer()` in that instance, as is the case with Listing 6-7, this default value is assigned to `developer()`.

Listing 6-7. Initializing a Stub Instance's Elements

```
public class Deck {
    @Stub(
        id = 1,
        dueDate = "12/21/2012"
    )
    public void shuffle() {
    }
}
```

Listing 6-7 reveals one `@Stub` annotation that initializes its `id()` element to 1 and its `dueDate()` element to "12/21/2012". Each element name doesn't have a trailing `()`, and the comma-separated list of two element initializers appears between `(` and `)`.

Suppose you decide to replace `Stub`'s `id()`, `dueDate()`, and `developer()` elements with a single `String value()` element whose string specifies comma-separated ID, due date, and developer name values. Listing 6-8 shows you two ways to initialize `value`.

Listing 6-8. Initializing Each Stub Instance's `value()` Element

```
public class Deck {
    @Stub(value = "1,12/21/2012,unassigned")
    public void shuffle() {
    }

    @Stub("2,12/21/2012,unassigned")
    public Card[] deal(int ncards) {
        return null;
    }
}
```

Listing 6-8 reveals special treatment for the `value()` element. When it's an annotation type's only element, you can omit `value()`'s name and `=` from the initializer. We used this fact to specify `@SuppressWarnings("deprecation")` in Listing 6-3.

Using Meta-annotations in Annotation Type Declarations

Each of the `Override`, `Deprecated`, and `SuppressWarnings` annotation types is itself annotated with *meta-annotations* (annotations that annotate annotation types). For example, Listing 6-9 shows you that the `SuppressWarnings` annotation type is annotated with two meta-annotations.

Listing 6-9. The Annotated `SuppressWarnings` Type Declaration

```
@Target(value={TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(value=SOURCE)
public @interface SuppressWarnings
```

The `Target` annotation type, which is located in the `java.lang.annotation` package, identifies the kinds of application elements to which an annotation type applies. `@Target` indicates that `@SuppressWarnings` annotations can be used to annotate types, fields, methods, parameters, constructors, and local variables.

Each of `TYPE`, `FIELD`, `METHOD`, `PARAMETER`, `CONSTRUCTOR`, and `LOCAL_VARIABLE` is a member of the `ElementType` enum, which is also located in the `java.lang.annotation` package. (We discuss enums later in this chapter.)

The { and } characters surrounding the comma-separated list of values assigned to `Target`'s `value()` element signify an array—`value()`'s return type is `String[]`. Although these braces are necessary (unless the array consists of one item), `value=` could be omitted when initializing `@Target` because `Target` declares only a `value()` element.

The `Retention` annotation type, which is located in the `java.lang.annotation` package, identifies the retention (also known as lifetime) of an annotation type's annotations. `@Retention` indicates that `@SuppressWarnings` annotations have a lifetime that is limited to source code—they don't exist after compilation.

`SOURCE` is one of the members of the `RetentionPolicy` enum (located in the `java.lang.annotation` package). The other members are `CLASS` and `RUNTIME`. These three members specify the following retention policies:

- `CLASS`: The compiler records annotations in the class file, but the virtual machine doesn't retain them (to save memory space). This policy is the default.

- **RUNTIME:** The compiler records annotations in the class file, and the virtual machine retains them so that they can be read via the Reflection API (see Chapter 8) at runtime.
- **SOURCE:** The compiler discards annotations after using them.

There are two problems with the `Stub` annotation type shown in Listings 6-4 and 6-6. First, the lack of an `@Target` meta-annotation means that you can annotate any application element `@Stub`. However, this annotation only makes sense when applied to methods and constructors. Check out Listing 6-10.

Listing 6-10. Annotating Undesirable Application Elements

```
@Stub("1,12/21/2012,unassigned")
public class Deck {
    @Stub("2,12/21/2012,unassigned")
    private Card[] cardsRemaining = new Card[52];

    @Stub("3,12/21/2012,unassigned")
    public Deck(){
    }

    @Stub("4,12/21/2012,unassigned")
    public void shuffle() {
    }

    @Stub("5,12/21/2012,unassigned")
    public Card[] deal(@Stub("5,12/21/2012,unassigned") int ncards) {
        return null;
    }
}
```

Listing 6-10 uses `@Stub` to annotate the `Deck` class, the `cardsRemaining` field, and the `ncards` parameter as well as annotating the constructor and the two methods. The first three application elements are inappropriate to annotate because they are not stubs.

You can fix this problem by prefixing the `Stub` annotation type declaration with `@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})` so that `Stub` only applies to methods and constructors. After doing this, the `javac` compiler tool will output the following error messages when you attempt to compile Listing 6-10:

```
Deck.java:1: error: annotation type not applicable to this kind of
declaration
@Stub("1,12/21/2012,unassigned")
^

Deck.java:4: error: annotation type not applicable to this kind of
declaration
@Stub("2,12/21/2012,unassigned")
^

Deck.java:18: error: annotation type not applicable to this kind of
declaration
public Card[] deal(@Stub("5,12/21/2012,unassigned") int ncards)
^

3 errors
```

The second problem is that the default CLASS retention policy makes it impossible to process @Stub annotations at runtime. You can fix this problem by prefixing the Stub type declaration with @Retention(RetentionPolicy.RUNTIME).

Listing 6-11 presents the Stub annotation type with the desired @Target and @Retention meta-annotations.

Listing 6-11. A Revamped Stub Annotation Type

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Stub {
    String value();
}
```

Note Java also provides Documented and Inherited meta-annotation types in the `java.lang.annotation` package. Instances of @Documented-annotated annotation types are to be documented by javadoc and similar tools, whereas instances of @Inherited-annotated annotation types are automatically inherited. According to Inherited’s Java documentation, if “the user queries the annotation type on a class declaration, and the class declaration has no annotation for this type, then the class’s superclass will automatically be queried for the annotation type. This process will be repeated until an annotation for this type is found, or the top of the class hierarchy (`Object`) is reached. If no superclass has an annotation for this type, then the query will indicate that the class in question has no such annotation.”

Processing Annotations

It’s not enough to declare an annotation type and use that type to annotate source code. Unless you do something specific with those annotations, they remain dormant. One way to accomplish something specific is to write an application that processes the annotations. Listing 6-12’s StubFinder application does just that.

Listing 6-12. The StubFinder Application

```
import java.lang.reflect.Method;

public class StubFinder {
    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("usage: java StubFinder classfile");
            return;
        }
        Method[] methods = Class.forName(args[0]).getMethods();
        for (int i = 0; i < methods.length; i++)
            if (methods[i].isAnnotationPresent(Stub.class)){
                Stub stub = methods[i].getAnnotation(Stub.class);
                String[] components = stub.value().split(",");
                System.out.println("Stub ID = " + components[0]);
            }
    }
}
```

```

        System.out.println("Stub Date = " + components[1]);
        System.out.println("Stub Developer = " + components[2]);
        System.out.println();
    }
}
}

```

StubFinder loads a class file whose name is specified as a command-line argument and outputs the metadata associated with each @Stub annotation that precedes each public method header. These annotations are instances of Listing 6-11's Stub annotation type.

StubFinder next uses a special class named Class and its `forName()` class method to load a class file. Class also provides a `getMethods()` method that returns an array of `java.lang.reflect.Method` objects (see Chapter 8) describing the loaded class's public methods.

For each loop iteration, a `Method` object's `isAnnotationPresent()` method is called to determine if the method is annotated with the annotation described by the `Stub` class (referred to as `Stub.class`).

If `isAnnotationPresent()` returns true, `Method`'s `getAnnotation()` method is called to return the annotation `Stub` instance. This instance's `value()` method is called to retrieve the string stored in the annotation.

Next, `String`'s `split()` method is called to split the string's comma-separated list of ID, date, and developer values into an array of `String` objects. Each object is then output along with descriptive text. (You will be formally introduced to `split()` in Chapter 7.)

`Class`'s `forName()` method is capable of throwing various exceptions that must be handled or explicitly declared as part of a method's header. For simplicity, we chose to append a `throws Exception` clause to the `main()` method's header.

Caution There are two problems with `throws Exception`. First, it is often better to handle the exception and present a suitable error message than to “pass the buck” by throwing it out of `main()`. Second, `Exception` is generic; it hides the names of the kinds of exceptions that are thrown. However, we find it convenient to specify `throws Exception` in a throwaway utility.

After compiling `StubFinder` (`javac StubFinder.java`), `Stub` (`javac Stub.java`), and Listing 6-8's `Deck` class (`javac Deck.java`), run `StubFinder` with `Deck` as its single command-line argument (`java StubFinder Deck`). You will observe the following output:

```
Stub ID = 1
Stub Date = 12/21/2012
Stub Developer = unassigned

Stub ID = 2
Stub Date = 12/21/2012
Stub Developer = unassigned
```

Mastering Generics

Java 5 introduced *generics*, language features for declaring and using type-agnostic classes and interfaces. While working with Java's Collections Framework (which we discuss in Chapter 9), these features help you avoid `java.lang.ClassCastException`.

Note Although the main use for generics is the Collections Framework, the standard class library also contains *generified* (retrofitted to make use of generics) classes that have nothing to do with this framework: `java.lang.Class`, `java.lang.ThreadLocal`, and `java.lang.ref.WeakReference` are three examples.

In this section, we will introduce you to generics. You will first learn how generics promote type safety in the context of the Collections Framework classes, and then you will explore generics in the contexts of generic types and generic methods. Finally, you will learn about generics in the context of arrays.

Collections and the Need for Type Safety

Java's Collections Framework makes it possible to store objects in various kinds of containers (known as collections) and later retrieve those objects. For example, you can store objects in a list, a set, or a map. You can then retrieve a single object, or iterate over the collection and retrieve all objects.

Before Java 5 overhauled the Collections Framework to take advantage of generics, there was no way to prevent a collection from containing objects of mixed types. The compiler didn't check an object's type to see if it was suitable before it was added to a collection, and this lack of static type checking led to `ClassCastException`s. It was legal from a compiler perspective to store a collection element as one type and retrieve it as a totally different type.

[Listing 6-13](#)'s generics-based homogenous list avoids such a `ClassCastException`.

Listing 6-13. Lack of Type Safety Leading to a Compiler Error

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Employee {
    private String name;

    public Employee(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

public class TypeSafety {
    public static void main(String[] args){
        List<Employee> employees = new ArrayList<Employee>();
        // or List<Employee> employees = new ArrayList<>();
        employees.add(new Employee("John Doe"));
        employees.add(new Employee("Jane Smith"));
        employees.add("Jack Frost"); // Illegal
        Iterator<Employee> iter = employees.iterator();
        while (iter.hasNext()) {
            Employee emp = iter.next();
            System.out.println(emp.getName());
        }
    }
}
```

Listing 6-13's `main()` method illustrates the central feature of generics, which is the *parameterized type* (a class or interface name followed by an angle bracket-delimited type list identifying what kinds of objects are legal in that context).

For example, `java.util.List<Employee>` indicates only `Employee` objects can be stored in the `List`. As shown, the `<Employee>` designation must be repeated with `ArrayList`, as in `ArrayList<Employee>`, which is the collection implementation that stores the `Employees`. It is however allowed here to omit the type and just write `ArrayList<>`, because the compiler is smart enough to infer the type from the left-hand side of the assignment.

Also, `Iterator<Employee>` indicates that `iterator()` returns an `Iterator` whose `next()` method returns only `Employee` objects. It's not necessary to cast `iter.next()`'s returned value to `Employee` because the compiler inserts the cast on your behalf.

If you attempt to compile this listing, the compiler will report an error when it encounters `employees.add("Jack Frost");`. The error message will tell you that the compiler cannot find an `add(java.lang.String)` method in the `java.util.List<Employee>` interface.

Unlike in the pre-generics `List` interface, which declares an `add(Object)` method, the generified `List` interface's `add()` method parameter reflects the interface's parameterized type name. For example, `List<Employee>` implies `add(Employee)`.

Rather than having to deal with angry clients while hunting down the unsafe code that ultimately led to the `ClassCastException`, you can rely on the compiler saving you this frustration and effort by reporting an error when it detects this code during compilation. ***Detecting type safety violations at compile time is the main benefit of using generics.***

Generic Types

A *generic type* is a class or interface that introduces a family of parameterized types by declaring a *formal type parameter list* (a comma-separated list of *type parameter* names between angle brackets). This syntax is expressed as follows:

```
class identifier<formal_type_parameter_list> {}
interface identifier<formal_type_parameter_list> {}
```

For example, `List<E>` is a generic type, where `List` is an interface and type parameter `E` identifies the list's element type. Similarly, `Map<K, V>` is a generic type, where `Map` is an interface and type parameters `K` and `V` identify the map's key and value types.

Note When declaring a generic type, it's conventional to specify single uppercase letters as type parameter names. Furthermore, these names should be meaningful. For example, E indicates element, T indicates type, K indicates key, and V indicates value. If possible, you should avoid choosing a type parameter name that is meaningless where it is used. For example, `List<E>` means list of elements, but what does `List<S>` mean?

Parameterized types are instances of generic types. Each parameterized type replaces the generic type's type parameters with type names. For example, `List<Employee>` (`List` of `Employee`) and `List<String>` (`List` of `String`) are examples of parameterized types based on `List<E>`. Similarly, `Map<String, Employee>` is an example of a parameterized type based on `Map<K, V>`.

The type name that replaces a type parameter is known as an *actual type argument*. Five kinds of actual type arguments are supported by generics:

- *Concrete type*: The name of a class or interface is passed to the type parameter. For example, `List<Employee> employees`; specifies that the list elements are `Employee` instances.
- *Concrete parameterized type*: The name of a parameterized type is passed to the type parameter. For example, `List<List<String>> nameLists`; specifies that the list elements are lists of strings.
- *Array type*: An array is passed to the type parameter. For example, `List<String[]> countries`; specifies that the list elements are arrays of `String`s, possibly city names.
- *Type parameter*: A type parameter is passed to the type parameter. For example, given class declaration `class X<E> { List<E> queue; }`, `X`'s type parameter `E` is passed to `List`'s type parameter `E`.
- *Wildcard*: The `?` is passed to the type parameter. For example, `List<?> list`; specifies that the list elements are unknown. You will learn about wildcards later in this chapter.

A generic type also identifies a *raw type*, which is a generic type without its type parameters. For example, `List<Employee>`'s raw type is `List`. Raw types are nongeneric and can hold any `Object`.

Note Java allows raw types to be intermixed with generic types to support the vast amount of legacy code that was written prior to the arrival of generics. However, the compiler outputs a warning message whenever it encounters a raw type in source code.

Declaring and Using Your Own Generic Types

It's not difficult to declare your own generic types. In addition to specifying a formal type parameter list, your generic type specifies its type parameter(s) throughout its implementation. For example, Listing 6-14 declares a `Queue<E>` generic type.

Listing 6-14. Declaring and Using a `Queue<E>` Generic Type

```
public class Queue<E> {
    private E[] elements;
    private int head, tail;

    @SuppressWarnings("unchecked")
    public Queue(int size) {
        if (size < 2)
            throw new IllegalArgumentException("" + size);
        elements = (E[]) new Object[size];
        head = 0;
        tail = 0;
    }

    public void insert(E element) throws QueueFullException {
        if (isFull())
            throw new QueueFullException();
        elements[tail] = element;
        tail = (tail + 1) % elements.length;
    }
}
```

```
public E remove() throws QueueEmptyException {
    if (isEmpty())
        throw new QueueEmptyException();
    E element = elements[head];
    head = (head + 1) % elements.length;
    return element;
}

public boolean isEmpty() {
    return head == tail;
}

public boolean isFull(){
    return (tail + 1) % elements.length == head;
}

public static void main(String[] args)
    throws QueueFullException, QueueEmptyException
{
    Queue<String> queue = new Queue<String>(6);
    System.out.println("Empty: " + queue.isEmpty());
    System.out.println("Full: " + queue.isFull());
    System.out.println("Adding A");
    queue.insert("A");
    System.out.println("Adding B");
    queue.insert("B");
    System.out.println("Adding C");
    queue.insert("C");
    System.out.println("Adding D");
    queue.insert("D");
    System.out.println("Adding E");
    queue.insert("E");
    System.out.println("Empty: " + queue.isEmpty());
    System.out.println("Full: " + queue.isFull());
    System.out.println("Removing " + queue.remove());
    System.out.println("Empty: " + queue.isEmpty());
    System.out.println("Full: " + queue.isFull());
```

```

System.out.println("Adding F");
queue.insert("F");
while (!queue.isEmpty())
    System.out.println("Removing " + queue.remove());
System.out.println("Empty: " + queue.isEmpty());
System.out.println("Full: " + queue.isFull());
}
}

public class QueueEmptyException extends Exception {
}

public class QueueFullException extends Exception {
}

```

Listing 6-14 declares `Queue`, `QueueEmptyException`, and `QueueFullException` classes. The latter two classes describe checked exceptions that are thrown from methods of the former class.

`Queue` implements a *queue*, a data structure that stores elements in first-in, first-out (FIFO) order. An element is inserted at the *tail* and removed at the *head*. The queue is empty when the head equals the tail and full when the tail is one less than the head. As a result, a queue of size n can store a maximum of $n - 1$ elements.

Notice that `Queue<E>`'s `E` type parameter appears throughout the source code. For example, `E` appears in the `elements` array declaration to denote the array's element type. `E` is also specified as the type of `insert()`'s parameter and as `remove()`'s return type.

`E` also appears in `elements = (E[]) new Object[size];`. (We will explain later why we specified this expression instead of specifying the apparently more compact `elements = new E[size];` expression.)

The `E[]` cast results in the compiler warning about this cast being unchecked. The compiler is concerned that downcasting from `Object[]` to `E[]` might result in a violation of type safety because any kind of object can be stored in `Object[]`.

The compiler's concern isn't justified in this example. There is no way that a non-`E` object can appear in the `E[]` array. Because the warning is meaningless in this context, it is suppressed by prefixing the constructor with `@SuppressWarnings("unchecked")`.

Caution Be careful when suppressing an unchecked warning. You must first prove that a `ClassCastException` cannot occur, and then you can suppress the warning.

When you run this application, it generates the following output:

```
Empty: true
Full: false
Adding A
Adding B
Adding C
Adding D
Adding E
Empty: false
Full: true
Removing A
Empty: false
Full: false
Adding F
Removing B
Removing C
Removing D
Removing E
Removing F
Empty: true
Full: false
```

Type Parameter Bounds

`List<E>`'s `E` type parameter and `Map<K, V>`'s `K` and `V` type parameters are examples of *unbounded type parameters*. You can pass any actual type argument to an unbounded type parameter.

It is sometimes necessary to restrict the kinds of actual type arguments that can be passed to a type parameter. For example, you might want to declare a class whose instances can only store instances of classes that subclass an abstract `Shape` class (such as `Circle` and `Rectangle`).

To restrict actual type arguments, you can specify an *upper bound*, a type that serves as an upper limit on the types that can be chosen as actual type arguments. The upper bound is specified via reserved word extends followed by a type name.

For example, `ShapesList<E extends Shape>` identifies `Shape` as an upper bound. You can specify `ShapesList<Circle>`, `ShapesList<Rectangle>`, and even `ShapesList<Shape>`, but not `ShapesList<String>` because `String` is not a subclass of `Shape`.

You can assign more than one upper bound to a type parameter, where the first bound is a class or interface and where each additional upper bound is an interface, by using the ampersand character (&) to separate bound names. Consider Listing 6-15.

Listing 6-15. Assigning Multiple Upper Bounds to a Type Parameter

```
abstract public class Shape {  
}  
  
public class Circle extends Shape implements Comparable<Circle> {  
    private double x, y, radius;  
  
    public Circle(double x, double y, double radius) {  
        this.x = x;  
        this.y = y;  
        this.radius = radius;  
    }  
  
    @Override  
    public int compareTo(Circle circle) {  
        if (radius < circle.radius)  
            return -1;  
        else  
        if (radius > circle.radius)  
            return 1;  
        else  
            return 0;  
    }  
}
```

```
@Override
public String toString() {
    return "(" + x + ", " + y + ", " + radius + ")";
}
}

public class SortedShapesList<S extends Shape & Comparable<S>> {
    @SuppressWarnings("unchecked")
    private S[] shapes = (S[]) new Shape[2];
    private int index = 0;

    public void add(S shape) {
        shapes[index++] = shape;
        if (index < 2)
            return;
        System.out.println("Before sort: " + this);
        sort();
        System.out.println("After sort: " + this);
    }

    private void sort() {
        if (index == 1)
            return;
        if (shapes[0].compareTo(shapes[1]) > 0) {
            S shape = (S) shapes[0];
            shapes[0] = shapes[1];
            shapes[1] = shape;
        }
    }

    @Override
    public String toString() {
        return shapes[0].toString() + " " + shapes[1].toString();
    }
}
```

```

public class SortedShapesListDemo {
    public static void main(String[] args) {
        SortedShapesList<Circle> ssl = new SortedShapesList<Circle>();
        ssl.add(new Circle(100, 200, 300));
        ssl.add(new Circle(10, 20, 30));
    }
}

```

Listing 6-15's `Circle` class extends `Shape` and implements the `java.lang.Comparable` interface, which is used to specify the *natural ordering* of `Circle` objects. The interface's `compareTo()` method implements this ordering by returning a value to reflect the order.

- A negative value is returned when the current object should precede the object passed to `compareTo()` in some fashion.
- A zero value is returned when the current and argument objects are the same.
- A positive value is returned when the current object should succeed the argument object.

`Circle`'s overriding `compareTo()` method compares two `Circle` objects based on their radii. This method orders a `Circle` instance with the smaller radius before a `Circle` instance with a larger radius.

The `SortedShapesList` class specifies `<S extends Shape & Comparable<S>>` as its parameter list. The actual type argument passed to the `S` parameter must subclass `Shape`, and it must also implement the `Comparable` interface.

Note A type parameter bound that includes the type parameter is known as a *recursive type bound*. For example, `Comparable<S>` in `S extends Shape & Comparable<S>` is a recursive type bound. Recursive type bounds are rare and typically show up in conjunction with the `Comparable` interface for specifying a type's natural ordering.

`Circle` satisfies both criteria: it subclasses `Shape` and implements `Comparable`. As a result, the compiler doesn't report an error when it encounters the `main()` method's `SortedShapesList<Circle> ssl = new SortedShapesList<Circle>();` statement.

An upper bound offers extra static type checking that guarantees that a parameterized type adheres to its bounds. This assurance means that the upper bound's methods can be called safely. For example, `sort()` can call Comparable's `compareTo()` method.

If you run this application, you will discover the following output, which shows that the two `Circle` objects are sorted in ascending order of radius:

```
Before sort: (100.0, 200.0, 300.0) (10.0, 20.0, 30.0)
After sort: (10.0, 20.0, 30.0) (100.0, 200.0, 300.0)
```

Note Type parameters cannot have lower bounds. Angelika Langer explains the rationale for this restriction in her “Java Generics FAQs” at www.angelikalanger.com/GenericsFAQ/FAQSections/TypeParameters.html#FAQ107.

Type Parameter Scope

A type parameter's *scope* (visibility) is its generic type except where *masked* (hidden). This scope includes the formal type parameter list of which the type parameter is a member. For example, the scope of `S` in `SortedShapesList<S extends Shape & Comparable<S>>` is all of `SortedShapesList` and the formal type parameter list.

It is possible to mask a type parameter by declaring a same-named type parameter in a nested type's formal type parameter list. For example, Listing 6-16 masks an enclosing class's `T` type parameter.

Listing 6-16. Masking a Type Parameter

```
public class EnclosingClass<T> {
    static public class EnclosedClass<T extends Comparable<T>> {
    }
}
```

`EnclosingClass`'s `T` type parameter is masked by `EnclosedClass`'s `T` type parameter, which specifies an upper bound where only those types that implement the `Comparable` interface can be passed to `EnclosedClass`. Referencing `T` from within `EnclosedClass` refers to the bounded `T` and not the unbounded `T` passed to `EnclosingClass`.

If masking is undesirable, it is best to choose a different name for the type parameter. For example, you might specify `EnclosedClass<U extends Comparable<U>>`. Although `U` is not as meaningful a name as `T`, this situation justifies this choice.

The Need for Wildcards

Suppose that you have created a `List` of `String` and want to output this list. Because you might create a `List` of `Employee` and other kinds of lists, you want this method to output an arbitrary `List` of `Object`. You end up creating Listing 6-17.

Listing 6-17. Attempting to Output a `List` of `Object`

```
import java.util.ArrayList;
import java.util.List;

public class OutputList {
    public static void main(String[] args) {
        List<String> ls = new ArrayList<String>();
        ls.add("first");
        ls.add("second");
        ls.add("third");
        outputList(ls);
    }

    static public void outputList(List<Object> list) {
        for (int i = 0; i < list.size(); i++)
            System.out.println(list.get(i));
    }
}
```

Now that you've accomplished your objective (or so you think), you compile Listing 6-17 via `javac OutputList.java`. Much to your surprise, you receive the following error message:

```
OutputList.java:12: error: method outputList in class OutputList cannot be
applied to given types;
    outputList(ls);
           ^
           ^
```

```

required: List<Object>
found: List<String>
reason: actual argument List<String> cannot be converted to List<Object>
by method invocation conversion
1 error

```

This error message results from being unaware of the fundamental rule of generic types: ***for a given subtype x of type y, and given G as a raw type declaration, G<x> is not a subtype of G<y>.***

To understand this rule, you must refresh your understanding of subtype polymorphism (see Chapter 4). Basically, a subtype is a specialized kind of its supertype. For example, `Circle` is a specialized kind of `Shape` and `String` is a specialized kind of `Object`. This polymorphic behavior also applies to related parameterized types with the same type parameters. For example, `List<Object>` is a specialized kind of `java.util.Collection<Object>`.

However, this polymorphic behavior doesn't apply to multiple parameterized types that differ only in regard to one type parameter being a subtype of another type parameter. For example, `List<String>` is not a specialized kind of `List<Object>`. The following example reveals why parameterized types differing only in type parameters are not polymorphic:

```

List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
lo.add(new Employee());
String s = ls.get(0);

```

This example will not compile because it violates type safety. If it compiled, a `ClassCastException` instance would be thrown at runtime because of the implicit cast to `String` on the final line.

The first line instantiates a `List` of `String` and the second line upcasts its reference to a `List` of `Object`. The third line adds a new `Employee` object to the `List` of `Object`. The fourth line obtains the `Employee` object via `get()` and attempts to assign it to the `List` of `String` reference variable. However, `ClassCastException` is thrown because of the implicit cast to `String`—an `Employee` is not a `String`.

Note Although you cannot upcast `List<String>` to `List<Object>`, you can upcast `List<String>` to the raw type `List` in order to interoperate with legacy code.

The aforementioned error message reveals that `List` of `String` is not also `List` of `Object`. To call Listing 6-17's `outputList()` method without violating type safety, you can only pass an argument of `List<Object>` type, which limits the usefulness of this method.

However, generics offer a solution: the wildcard argument `(?)`, which stands for any type. By changing `outputList()`'s parameter type from `List<Object>` to `List<?>`, you can call `outputList()` with a `List` of `String`, a `List` of `Employee`, and so on.

Generic Methods

Suppose you need a method to copy a `List` of any kind of object to another `List`. Although you might consider coding a `void copyList(List<Object> src, List<Object> dest)` method, this method would have limited usefulness because it could only copy lists whose element type is `Object`. You couldn't copy a `List<Employee>`, for example.

If you want to pass source and destination lists whose elements are of arbitrary type (but their element types agree), you need to specify the wildcard character as a placeholder for that type. For example, you might consider writing the following `copyList()` class method that accepts collections of arbitrary-typed objects as its arguments:

```
static void copyList(List<?> src, List<?> dest) {  
    for (int i = 0; i < src.size(); i++)  
        dest.add(src.get(i));  
}
```

This method's parameter list is correct, but there is another problem: the compiler outputs the following error message when it encounters `dest.add(src.get(i));`:

```
CopyList.java:19: error: no suitable method found for add(Object)  
        dest.add(src.get(i));  
        ^
```

```

method List.add(int,CAP#1) is not applicable
  (actual and formal argument lists differ in length)
method List.add(CAP#1) is not applicable
  (actual argument Object cannot be converted to CAP#1 by method
  invocation conversion)
where CAP#1 is a fresh type-variable:
  CAP#1 extends Object from capture of ?

```

1 error

This error message assumes that `copyList()` is part of a class named `CopyList`. Although it appears to be incomprehensible, the message basically means that the `dest.add(src.get(i))` method call violates type safety. Because `?` implies that any type of object can serve as a list's element type, it's possible that the destination list's element type is incompatible with the source list's element type.

For example, suppose you create a `List<String>` as the source list and a `List<Employee>` as the destination list. Attempting to add the source list's elements to the destination list, which expects `Employees`, violates type safety. If this copy operation were allowed, a `ClassCastException` instance would be thrown when trying to obtain the destination list's elements.

The problem of copying lists of arbitrary element types to other lists can be solved through the use of a *generic method* (a class or instance method with a type-generalized implementation). Generic methods are syntactically expressed as follows:

`<formal_type_parameter_list> return_type identifier(parameter_list)`

The `formal_type_parameter_list` is the same as when specifying a generic type: it consists of type parameters with optional bounds. A type parameter can appear as the method's `return_type`, and type parameters can appear in the `parameter_list`. The compiler infers the actual type arguments from the context in which the method is invoked.

You'll discover many examples of generic methods in the Collections Framework. For example, its `java.util.Collections` class provides a public static `<T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)` method for returning the minimum element in the given collection according to the natural ordering of its elements.

You can easily convert `copyList()` into a generic method by prefixing the return type with `<T>` and replacing each wildcard with `T`. The resulting method header is `<T> void`

`copyList(List<T> src, List<T> dest)`, and Listing 6-18 presents its source code as part of an application that copies a List of Circle to another List of Circle.

Listing 6-18. Declaring and Using a `copyList()` Generic Method

```
import java.util.ArrayList;
import java.util.List;

public class Circle {
    private double x, y, radius;

    public Circle(double x, double y, double radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ", " + radius + ")";
    }
}

public class CopyList {
    public static void main(String[] args) {
        List<String> ls = new ArrayList<String>();
        ls.add("A");
        ls.add("B");
        ls.add("C");
        outputList(ls);
        List<String> lsCopy = new ArrayList<String>();
        copyList(ls, lsCopy);
        outputList(lsCopy);
        List<Circle> lc = new ArrayList<Circle>();
        lc.add(new Circle(10.0, 20.0, 30.0));
        lc.add(new Circle (5.0, 4.0, 16.0));
        outputList(lc);
        List<Circle> lcCopy = new ArrayList<Circle>();
```

```

        copyList(lc, lcCopy);
        outputList(lcCopy);
    }

    static <T> public void copyList(List<T> src, List<T> dest) {
        for (int i = 0; i < src.size(); i++)
            dest.add(src.get(i));
    }

    static public void outputList(List<?> list) {
        for (int i = 0; i < list.size(); i++)
            System.out.println(list.get(i));
        System.out.println();
    }
}

```

The compiler uses a *type inference algorithm* to infer a generic method's type arguments from the context in which the method was invoked. For example, the compiler determines that `copyList(ls, lsCopy);` copies a `List` of `String` to another `List` of `String`. Similarly, it determines that `copyList(lc, lcCopy);` copies a `List` of `Circle` to another `List` of `Circle`. Without this algorithm, you would have to specify these arguments, as in `CopyList.<String>copyList(ls, lsCopy);` and `CopyList.<Circle>copyList(lc, lcCopy);`.

Compile Listing 6-18 (`javac CopyList.java`) and run this application (`java CopyList`). You should observe the following output:

```

A
B
C

A
B
C

(10.0, 20.0, 30.0)
(5.0, 4.0, 16.0)

(10.0, 20.0, 30.0)
(5.0, 4.0, 16.0)

```

GENERIC CONSTRUCTORS

Generic and nongeneric classes can declare *generic constructors* in which a constructor has a formal type parameter list. For example, you could declare the following nongeneric class with a generic constructor:

```
public class GenericConstructorDemo {
    <T> GenericConstructorDemo(T type){
        System.out.println(type);
    }

    public static void main(String[] args){
        GenericConstructorDemo gcd = new GenericConstructorDemo("ABC");
        gcd = new GenericConstructorDemo(new Integer(100));
    }
}
```

Compile this class (`javac GenericConstructorDemo.java`) and run the resulting application (`java GenericConstructorDemo`). It outputs ABC on one line followed by 100 on the next line.

Arrays and Generics

After presenting Listing 6-14's `Queue<E>` generic type, we mentioned that we would explain why we specified `elements = (E[]) new Object[size];` instead of the more compact `elements = new E[size];` expression. Because of Java's generics implementation, it isn't possible to specify array-creation expressions that involve type parameters (such as `new E[size]` or `new List<E>[50]`) or actual type arguments (such as `new Queue<String>[15]`). If you attempt to do so, the compiler will report a generic array creation error message.

Before we present an example that demonstrates why allowing array-creation expressions that involve type parameters or actual type arguments is dangerous, you need to understand reification and covariance in the context of arrays, and erasure, which is at the heart of how generics are implemented.

Reification is representing the abstract as if it was concrete—for example, making a memory address available for direct manipulation by other language constructs. Java arrays are reified in that they're aware of their element types (an element type is

stored internally) and can enforce these types at runtime. Attempting to store an invalid element in an array causes the virtual machine to throw an instance of the `java.lang.ArrayStoreException` class.

[Listing 6-19](#) teaches you how array manipulation can lead to an `ArrayStoreException`.

Listing 6-19. How an `ArrayStoreException` Arises

```
public class Point {
    public int x, y;
}

public class ColoredPoint extends Point {
    public int color;
}

public class ReificationDemo {
    public static void main(String[] args) {
        // ColoredPoint[] cptArray = new Point[1]; // illegal
        ColoredPoint[] cptArray = new ColoredPoint[1];
        Point[] ptArray = cptArray;
        ptArray[0] = new Point();
    }
}
```

[Listing 6-19](#)'s `main()` method first instantiates a `ColoredPoint` array that can store one element. In contrast to this legal assignment (the types are compatible), specifying `ColoredPoint[] cptArray = new Point[1];` is illegal (and won't compile) because it would result in a `ClassCastException` at runtime—the array knows that the assignment is illegal.

Note If it's not obvious, `ColoredPoint[] cptArray = new Point[1];` is illegal because `Point` instances have fewer members (only `x` and `y`) than `ColoredPoint` instances (`x`, `y`, and `color`). Attempting to access a `Point` instance's nonexistent `color` field from its entry in the `ColoredPoint` array would result in a memory violation (because no memory has been assigned to `color`) and ultimately crash the virtual machine.

The second line (`Point[] ptArray = cptArray;`) is legal because of *covariance* (an array of supertype references is a supertype of an array of subtype references). In this case, an array of `Point` references is a supertype of an array of `ColoredPoint` references. The nonarray analogy is that a subtype is also a supertype. For example, a `java.lang.Throwable` instance is a kind of `Object` instance.

Covariance is dangerous when abused. For example, the third line (`ptArray[0] = new Point();`) results in `ArrayStoreException` at runtime because a `Point` instance is not a `ColoredPoint` instance. Without this exception, an attempt to access the nonexistent member `color` crashes the virtual machine.

Unlike with arrays, a generic type's type parameters are not reified. They're not available at runtime because they're thrown away after the source code is compiled. This "throwing away of type parameters" is a result of *erasure*, which also involves inserting casts to appropriate types when the code isn't type correct, and replacing type parameters by their upper bounds (such as `Object`).

Note The compiler performs erasure to let generic code interoperate with legacy (nongeneric) code. It transforms generic source code into nongeneric runtime code. One consequence of erasure is that you cannot use the `instanceof` operator with parameterized types apart from unbounded wildcard types. For example, it's illegal to specify `List<Employee> le = null; if (le instanceof ArrayList<Employee>) {}`. Instead, you must change the `instanceof` expression to `le instanceof ArrayList<?>` (unbounded wildcard) or `le instanceof ArrayList` (raw type, which is the preferred use).

Suppose you could specify an array-creation expression involving a type parameter or an actual type argument. Why would this be bad? For an answer, consider the following example, which should generate an `ArrayStoreException` instead of a `ClassCastException` but doesn't do so:

```
List<Employee>[] empListArray = new ArrayList<Employee>[1];
List<String> strList = new ArrayList<String>();
strList.add("string");
Object[] objArray = empListArray;
objArray[0] = strList;
Employee e = empListArray[0].get(0);
```

Assume that the first line, which creates a one-element array where this element stores a `List` of `Employee`, is legal. The second line creates a `List` of `String`, and the third line stores a single `String` object in this list.

The fourth line assigns `empListArray` to `objArray`. This assignment is legal because arrays are covariant and erasure converts `List<Employee>[]` to the `List` runtime type and `List` subtypes `Object`.

Because of erasure, the virtual machine doesn't throw `ArrayStoreException` when it encounters `objArray[0] = strList;`. After all, you're assigning a `List` reference to a `List[]` array at runtime. However, this exception would be thrown if generic types were reified because you'd then be assigning a `List<String>` reference to a `List<Employee>[]` array.

However, there is a problem. A `List<String>` instance has been stored in an array that can only hold `List<Employee>` instances. When the compiler-inserted cast operator attempts to cast `empListArray[0].get(0)`'s return value ("string") to `Employee`, the cast operator throws a `ClassCastException` object.

Mastering Enums

An *enumerated type* is a type that specifies a named sequence of related constants as its legal values. The months in a calendar, the coins in a currency, and the days of the week are examples of enumerated types.

Java developers have traditionally used sets of named integer constants to represent enumerated types. Because this form of representation has proven to be problematic, Java 5 introduced the `enum` alternative.

In this section, we will introduce you to enums. After discussing the problems with traditional enumerated types, we will present the `enum` alternative. We will then introduce you to the `Enum` class, from which enums originate.

The Trouble with Traditional Enumerated Types

Listing 6-20 declares a `Coin` enumerated type whose set of constants identifies different kinds of coins in a currency.

Listing 6-20. An Enumerated Type Identifying Coins

```
public class Coin {
    final static int PENNY = 0;
    final static int NICKEL = 1;
    final static int DIME = 2;
    final static int QUARTER = 3;
}
```

Listing 6-21 declares a Weekday enumerated type whose constants identify the days of the week.

Listing 6-21. An Enumerated Type Identifying Weekdays

```
public class Weekday {
    final static int SUNDAY = 0;
    final static int MONDAY = 1;
    final static int TUESDAY = 2;
    final static int WEDNESDAY = 3;
    final static int THURSDAY = 4;
    final static int FRIDAY = 5;
    final static int SATURDAY = 6;
}
```

Listings 6-20 and 6-21's approach to representing an enumerated type is problematic, where the biggest problem is the lack of compile-time type safety. For example, you can pass a coin to a method that requires a weekday and the compiler will not complain.

You can also compare coins to weekdays, as in `Coin.NICKEL == Weekday.MONDAY`, and specify even more meaningless expressions, such as `Coin.DIME + Weekday.FRIDAY - 1 / Coin.QUARTER`. The compiler doesn't complain because it only sees ints.

Applications that depend upon enumerated types are brittle. Because the type's constants are compiled into an application's class files, changing a constant's int value requires you to recompile dependent applications or risk them behaving erratically.

Another problem with enumerated types is that int constants cannot be translated into meaningful string descriptions. For example, what does the number 4 mean when debugging a faulty application? Being able to see THURSDAY instead of 4 would be more helpful.

Note You could circumvent the previous problem by using String constants. For example, you might specify `public final static String THURSDAY = "THURSDAY";`. Although the constant value is more meaningful, String-based constants can impact performance because you cannot use `==` to efficiently compare just any old strings (as you will discover in Chapter 7). Other problems related to String-based constants include hard-coding the constant's value ("THURSDAY") instead of the constant's name (THURSDAY) into source code, which makes it very difficult to change the constant's value at a later time, and misspelling a hard-coded constant ("THURZDAY"), which compiles correctly but is problematic at runtime.

The Enum Alternative

Java 5 introduced enums as a better alternative to traditional enumerated types. An *enum* is an enumerated type that is expressed via reserved word `enum`. The following example uses `enum` to declare Listings 6-20 and 6-21's enumerated types:

```
public enum Coin { PENNY, NICKEL, DIME, QUARTER }
public enum Weekday { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY }
```

Despite their similarity to the `int`-based enumerated types found in C++ and other languages, this example's enums are classes. Each constant is a `public static final` field that represents an instance of its enum class.

Because constants are final, and because you cannot call an enum's constructors to create more constants, you can use `==` to compare constants efficiently and (unlike string constant comparisons) safely. For example, you can specify `c == Coin.NICKEL`.

Enums promote compile-time type safety by preventing you from comparing constants in different enums. For example, the compiler will report an error when it encounters `Coin.PENNY == Weekday.SUNDAY`.

The compiler also frowns on passing a constant of the wrong enum kind to a method. For example, you cannot pass `Weekday.FRIDAY` to a method whose parameter type is `Coin`.

Applications depending on enums are not brittle because the enum's constants are not compiled into an application's class files. Also, the enum provides a `toString()` method for returning a more useful description of a constant's value.

Because enums are so useful, Java 5 enhanced the switch statement to support them. Listing 6-22 demonstrates this statement switching on one of the constants in the previous example's `Coin` enum.

Listing 6-22. Using the Switch Statement with an Enum

```
public class EnhancedSwitch {  
    public enum Coin { PENNY, NICKEL, DIME, QUARTER }  
  
    public static void main(String[] args) {  
        Coin coin = Coin.NICKEL;  
        switch (coin) {  
            case PENNY : System.out.println("1 cent"); break;  
            case NICKEL : System.out.println("5 cents"); break;  
            case DIME : System.out.println("10 cents"); break;  
            case QUARTER: System.out.println("25 cents"); break;  
        }  
    }  
}
```

Listing 6-22 demonstrates switching on an enum's constants. This enhanced statement only allows you to specify the name of a constant as a case label. If you prefix the name with the enum, as in `case Coin.DIME`, the compiler reports an error.

Enhancing an Enum

You can add fields, constructors, and methods to an enum—you can even have the enum implement interfaces. For example, Listing 6-23 adds a field, a constructor, and two methods to `Coin` to associate a denomination value with a `Coin` constant (such as 1 for penny and 5 for nickel) and convert pennies to the denomination.

Listing 6-23. Enhancing the Coin Enum

```
public enum Coin {  
    PENNY(1),  
    NICKEL(5),  
    DIME(10),  
    QUARTER(25);  
  
    private final int denomValue;  
  
    public Coin(int denomValue) {  
        this.denomValue = denomValue;  
    }  
  
    public int denomValue() {  
        return denomValue;  
    }  
  
    public int toDenomination(int numPennies) {  
        return numPennies / denomValue;  
    }  
}
```

Listing 6-23’s constructor accepts a denomination value, which it assigns to a private blank final field named `denomValue`—all fields should be declared `final` because constants are immutable. Notice that this value is passed to each constant during its creation (e.g., `PENNY(1)`).

Caution When the comma-separated list of constants is followed by anything other than an enum’s closing brace, you must terminate the list with a semicolon or the compiler will report an error.

Furthermore, this listing’s `denomValue()` method returns `denomValue`, and its `toDenomination()` method returns the number of coins of that denomination that are contained within the number of pennies passed to this method as its argument. For example, 3 nickels are contained in 16 pennies.

Listing 6-24 shows how to use the enhanced Coin enum.

Listing 6-24. Exercising the Enhanced Coin Enum

```

public class Coins {
    public static void main(String[] args) {
        if (args.length == 1) {
            int numPennies = Integer.parseInt(args[0]);
            System.out.println(numPennies + " pennies is equivalent to:");
            int numQuarters = Coin.QUARTER.toDenomination(numPennies);
            System.out.println(numQuarters + " " + Coin.QUARTER.toString() +
                (numQuarters != 1 ? "s," : ","));
            numPennies -= numQuarters * Coin.QUARTER.denomValue();
            int numDimes = Coin.DIME.toDenomination(numPennies);
            System.out.println(numDimes + " " + Coin.DIME.toString() +
                (numDimes != 1 ? "s, " : ","));
            numPennies -= numDimes * Coin.DIME.denomValue();
            int numNickels = Coin.NICKEL.toDenomination(numPennies);
            System.out.println(numNickels + " " + Coin.NICKEL.toString() +
                (numNickels != 1 ? "s, " : ", and"));
            numPennies -= numNickels * Coin.NICKEL.denomValue();
            System.out.println(numPennies + " " + Coin.PENNY.toString() +
                (numPennies != 1 ? "s" : ""));
        }
        System.out.println();
        System.out.println("Denomination values:");
        for (int i = 0; i < Coin.values().length; i++)
            System.out.println(Coin.values()[i].denomValue());
    }
}

```

Listing 6-24 describes an application that converts its solitary “pennies” command-line argument to an equivalent amount expressed in quarters, dimes, nickels, and pennies. In addition to calling a `Coin` constant’s `denomValue()` and `toDenomValue()` methods, the application calls `toString()` to output a string representation of the coin.

Another called enum method is `values()`. This method returns an array of all `Coin` constants that are declared in the `Coin` enum (`value()`’s return type, in this example, `Coin[]`). This array is useful when you need to iterate over these constants. For example, Listing 6-24 calls this method to output each coin’s denomination.

When you run this application with 119 as its command-line argument (`java Coins 119`), it generates the following output:

119 pennies is equivalent to:

4 QUARTERs,
1 DIME,
1 NICKEL, and
4 PENNYs

Denomination values:

1
5
10
25

The output shows that `toString()` returns a constant's name. It is sometimes useful to override this method to return a more meaningful value. For example, a method that extracts *tokens* (named character sequences) from a string might use a `Token` enum to list token names and, via an overriding `toString()` method, values—see Listing 6-25.

Listing 6-25. Overriding `toString()` to Return a Token Constant's Value

```
public enum Token {
    IDENTIFIER("ID"),
    INTEGER("INT"),
    LPAREN("("),
    RPAREN(")"),
    COMMA(",");
    private final String tokValue;
    public Token(String tokValue) {
        this.tokValue = tokValue;
    }
    @Override
    public String toString() {
        return tokValue;
    }
}
```

```

public static void main(String[] args) {
    System.out.println("Token values:");
    for (int i = 0; i < Token.values().length; i++)
        System.out.println(Token.values()[i].name() + " = " +
                           Token.values()[i]);
}
}

```

Listing 6-25's `main()` method calls `values()` to return the array of `Token` constants. For each constant, it calls the constant's `name()` method to return the constant's name and implicitly calls `toString()` to return the constant's value. If you were to run this application, you would observe the following output:

```

Token values:
IDENTIFIER = ID
INTEGER = INT
LPAREN = (
RPAREN = )
COMMA = ,

```

Another way to enhance an enum is to assign a different behavior to each constant. You can accomplish this task by introducing an abstract method into the enum and overriding this method in an anonymous subclass of the constant. Listing 6-26's `TempConversion` enum demonstrates this technique.

Listing 6-26. Using Anonymous Subclasses to Vary the Behaviors of Enum Constants

```

public enum TempConversion {
    C2F("Celsius to Fahrenheit") {
        @Override
        public double convert(double value) {
            return value * 9.0 / 5.0 + 32.0;
        }
    },
}

```

```

F2C("Fahrenheit to Celsius") {
    @Override
    public double convert(double value) {
        return (value - 32.0) * 5.0 / 9.0;
    }
};

public TempConversion(String desc) {
    this.desc = desc;
}

private String desc;

@Override
public String toString() {
    return desc;
}

abstract public double convert(double value);

public static void main(String[] args) {
    System.out.println(C2F + " for 100.0 degrees = " + C2F.
convert(100.0));
    System.out.println(F2C + " for 98.6 degrees = " + F2C.convert(98.6));
}
}

```

When you run this application, it generates the following output:

```

Celsius to Fahrenheit for 100.0 degrees = 212.0
Fahrenheit to Celsius for 98.6 degrees = 37.0

```

The Enum Class

The compiler regards enum as syntactic sugar. When it encounters an enum type declaration (enum Coin {}), it generates a class whose name (Coin) is specified by the declaration, which also subclasses the abstract `Enum` class (in the `java.lang` package), the common base class of all Java language-based enumeration types.

If you examine `Enum`'s Java documentation, you will discover that it overrides `Object`'s `clone()`, `equals()`, `finalize()`, `hashCode()`, and `toString()` methods.

- `clone()` is overridden to prevent constants from being cloned so that there is never more than one copy of a constant; otherwise, constants could not be compared via `==`.
- `equals()` is overridden to compare constants via their references—constants with the same identities (`==`) must have the same contents (`equals()`), and different identities imply different contents.
- `finalize()` is overridden to ensure that constants cannot be finalized.
- `hashCode()` is overridden because `equals()` is overridden.
- `toString()` is overridden to return the constant's name.

Except for `toString()`, all of the overriding methods are declared `final` so that they cannot be overridden in a subclass.

`Enum` also provides its own methods. These methods include the `final` `compareTo()` (`Enum` implements `Comparable`), `getDeclaringClass()`, `name()`, and `ordinal()` methods.

- `compareTo()` compares the current constant with the constant passed as an argument to see which constant precedes the other constant in the enum and returns a value indicating their order. This method makes it possible to sort an array of unsorted constants.
- `getDeclaringClass()` returns the `Class` object corresponding to the current constant's enum. For example, the `Class` object for `Coin` is returned when calling `Coin.PENNY.getDeclaringClass()` for enum `Coin { PENNY, NICKEL, DIME, QUARTER}`. Also, `TempConversion` is returned when calling `TempConversion.C2F.getDeclaringClass()` for Listing 6-26's `TempConversion` enum. The `compareTo()` method uses `Class`'s `getClass()` method and `Enum`'s `getDeclaringClass()` method to ensure that only constants belonging to the same enum are compared. Otherwise, a `ClassCastException` is thrown.
- `name()` returns the constant's name. Unless overridden to return something more descriptive, `toString()` also returns the constant's name.

- `ordinal()` returns a zero-based *ordinal*, an integer that identifies the position of the constant within the enum type. `compareTo()` compares ordinals.

Enum also provides the `public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)` method for returning the enum constant from the specified enum with the specified name.

- `enumType` identifies the `Class` object of the enum from which to return a constant.
- `name` identifies the name of the constant to return.

For example, `Coin penny = Enum.valueOf(Coin.class, "PENNY");` assigns the `Coin` constant whose name is `PENNY` to `penny`.

You will not discover a `values()` method in `Enum`'s Java documentation because the compiler *synthesizes* (manufactures) this method while generating the class.

Extending the `Enum` Class

`Enum`'s generic type is `Enum<E extends Enum<E>>`. Although the formal type parameter list looks ghastly, it's not that hard to understand. But first, take a look at Listing 6-27.

Listing 6-27. The `Coin` Class As It Appears from the Perspective of Its Class file

```
public final class Coin extends Enum<Coin> {
    public static final Coin PENNY = new Coin("PENNY", 0);
    public static final Coin NICKEL = new Coin("NICKEL", 1);
    public static final Coin DIME = new Coin("DIME", 2);
    public static final Coin QUARTER = new Coin("QUARTER", 3);
    private static final Coin[] $VALUES = { PENNY, NICKEL, DIME, QUARTER };

    public static Coin[] values() {
        return Coin.$VALUES.clone();
    }

    public static Coin valueOf(String name) {
        return Enum.valueOf(Coin.class, "Coin");
    }
}
```

```
private Coin(String name, int ordinal) {
    super(name, ordinal);
}
```

Behind the scenes, the compiler converts `enum Coin { PENNY, NICKEL, DIME, QUARTER}` into a class declaration that is similar to Listing 6-27.

The following rules show you how to interpret `Enum<E extends Enum<E>>` in the context of `Coin extends Enum<Coin>`:

- Any subclass of `Enum` must supply an actual type argument to `Enum`. For example, `Coin`'s header specifies `Enum<Coin>`.
- The actual type argument must be a subclass of `Enum`. For example, `Coin` is a subclass of `Enum`.
- A subclass of `Enum` (such as `Coin`) must follow the idiom that it supplies its own name (`Coin`) as an actual type argument.

The third rule allows `Enum` to declare methods—`compareTo()`, `getDeclaringClass()`, and `valueOf()`—whose parameter and/or return types are specified in terms of the subclass (`Coin`) and not in terms of `Enum`. The rationale for doing this is to avoid having to specify casts. For example, you don't need to cast `valueOf()`'s return value to `Coin` in `Coin penny = Enum.valueOf(Coin.class, "PENNY");`.

Note You cannot compile Listing 6-27 because the compiler will not compile any class that extends `Enum`. It will also complain about `super(name, ordinal);`.

EXERCISES

The following exercises are designed to test your understanding of Chapter 6's content:

1. Define annotation.
2. What kinds of application elements can be annotated?
3. Identify the three compiler-supported annotation types.
4. How do you declare an annotation type?

5. What is a marker annotation?
6. What is an element?
7. How do you assign a default value to an element?
8. What is a meta-annotation?
9. Identify Java's four meta-annotation types.
10. Define generics.
11. Why would you use generics?
12. What is the difference between a generic type and a parameterized type?
13. Which one of the nonstatic member class, local class, and anonymous class inner class categories cannot be generic?
14. Identify the five kinds of actual type arguments.
15. True or false: You cannot specify the name of a primitive type (such as double or int) as an actual type argument.
16. What is a raw type?
17. When does the compiler report an unchecked warning message and why?
18. How do you suppress an unchecked warning message?
19. True or false: List<E>'s E type parameter is unbounded.
20. How do you specify a single upper bound?
21. What is a recursive type bound?
22. Why are wildcard type arguments necessary?
23. What is a generic method?
24. In Listing 6-28, which overloaded method does the methodCaller() generic method call?

Listing 6-28. Which someOverloadedMethod() Is Called?

```
import java.util.Date;  
  
public class CallOverloadedNGMethodFromGMethod {
```

```
public static void someOverloadedMethod(Object o) {  
    System.out.println("call to someOverloadedMethod(Object o)");  
}  
public static void someOverloadedMethod(Date d) {  
    System.out.println("call to someOverloadedMethod(Date d)");  
}  
public static <T> void methodCaller(T t) {  
    someOverloadedMethod(t);  
}  
public static void main(String[] args) {  
    methodCaller(new Date());  
}
```

25. What is reification?
26. True or false: Type parameters are reified.
27. What is erasure?
28. Define enumerated type.
29. Identify three problems that can arise when you use enumerated types whose constants are int-based.
30. What is an enum?
31. How do you use the switch statement with an enum?
32. In what ways can you enhance an enum?
33. What is the purpose of the abstract Enum class?
34. What is the difference between Enum's name() and toString() methods?
35. True or false: Enum's generic type is Enum<E extends Enum<E>>.
36. Declare a ToDo marker annotation type that annotates only type elements and that also uses the default retention policy.
37. Rewrite the StubFinder application to work with Listing 6-6's Stub annotation type (with appropriate @Target and @Retention annotations) and Listing 6-7's Deck class.

38. Implement a `Stack<E>` generic type in a manner that is similar to Listing 6-14's `Queue` class. `Stack` must declare `push()`, `pop()`, and `isEmpty()` methods (it could also declare an `isFull()` method, but that method is not necessary in this exercise); `push()` must throw a `StackFullException` instance when the stack is full; and `pop()` must throw a `StackEmptyException` instance when the stack is empty. (You must create your own `StackFullException` and `StackEmptyException` helper classes because they are not provided for you in the standard class library.) Declare a similar `main()` method, and insert two assertions into this method that validate your assumptions about the stack being empty immediately after being created and immediately after popping the last element.
 39. Declare a `Compass` enum with `NORTH`, `SOUTH`, `EAST`, and `WEST` members. Declare a `UseCompass` class whose `main()` method randomly selects one of these constants and then switches on that constant. Each of the switch statement's cases should output a message such as `heading north`.
-

Summary

Annotations are instances of annotation types and associate metadata with application elements. They are expressed in source code by prefixing their type names with @ symbols. For example, `@ReadOnly` is an annotation and `ReadOnly` is its type.

Java supplies a wide variety of annotation types, including the compiler-oriented `Override`, `Deprecated`, and `SuppressWarnings` types. However, you can also declare your own annotation types by using the `@interface` syntax.

Annotation types can be annotated with meta-annotations that identify the application elements they can target (such as constructors, methods, or fields), their retention policies, and other characteristics.

Annotations whose types are assigned a runtime retention policy via `@Retention` annotations can be processed at runtime using custom applications. (Java 5 introduced an `apt` tool for this purpose, but its functionality was largely integrated into the compiler starting with Java 6.)

Java 5 introduced generics, language features for declaring and using type-agnostic classes and interfaces. While working with Java's Collections Framework, these features help you avoid `ClassCastException`s.

A generic type is a class or interface that introduces a family of parameterized types by declaring a formal type parameter list. The type name that replaces a type parameter is known as an actual type argument.

There are five kinds of actual type arguments: concrete type, concrete parameterized type, array type, type parameter, and wildcard. Furthermore, a generic type also identifies a raw type, which is a generic type without its type parameters.

A generic method is a class or instance method with a type-generalized implementation, for example, `<T> void copyList(List<T> src, List<T> dest)`. The compiler infers the actual type argument from the context in which the method is invoked.

An enumerated type is a type that specifies a named sequence of related constants as its legal values. Java developers have traditionally used sets of named integer constants to represent enumerated types.

Because sets of named integer constants have proven to be problematic, Java 5 introduced the enum alternative. An enum is an enumerated type that is expressed in source code via reserved word enum.

You can add fields, constructors, and methods to an enum—you can even have the enum implement interfaces. Also, you can override `toString()` to provide a more useful description of a constant's value, and subclass constants to assign different behaviors.

The compiler regards enum as syntactic sugar for a class that subclasses Enum. This abstract class overrides various Object methods to provide default behaviors (usually for safety reasons) and provides additional methods for various purposes.

This chapter largely completes a tour of the Java language. In Chapter 7, we will begin to emphasize Java APIs by focusing on those basic APIs related to mathematics, string management, and more.

CHAPTER 7

Exploring the Basic APIs, Part 1

The standard class library's `java.lang` and other packages provide many basic APIs that can benefit your Android apps. For example, you can perform mathematics operations and manipulate strings.

Note We don't list all classes and all methods and fields of the corresponding Java standard packages. You are free to investigate the official API documentation at <https://docs.oracle.com/javase/8/docs/api>. It is important to know that Android does not support all APIs, though. You will be given notice about that where appropriate.

Exploring Math

The `java.lang.Math` class declares double constants `E` and `PI` that represent the natural logarithm base value (2.71828...) and the ratio of a circle's circumference to its diameter (3.14159...). `E` is initialized to 2.718281828459045 and `PI` is initialized to 3.141592653589793. `Math` also declares class methods that perform various mathematics operations. Besides, the class provides a lot of mathematical operations you can use everywhere in your applications.

Its `abs(double)` and `abs(float)` methods are useful for comparing double-precision floating-point and floating-point values safely. For example, `0.3 == 0.1 + 0.1 + 0.1` evaluates to false because 0.1 has no exact representation. However, you can compare these expressions with `abs()` and a tolerance value, which indicates an acceptable range of error. For example, `Math.abs(0.3 - (0.1 + 0.1 + 0.1)) < 0.1` returns true because the absolute difference between 0.3 and 0.1 + 0.1 + 0.1 is less than a 0.1 tolerance value.

`random()` (which returns a number that appears to be randomly chosen but is actually chosen by a predictable math calculation and hence is *pseudorandom*) is useful in simulations (as well as in games and wherever an element of chance is needed).

The `sin()` and `cos()` methods implement the sine and cosine trigonometric functions—see http://en.wikipedia.org/wiki/Trigonometric_functions. These functions have uses ranging from the study of triangles to modeling periodic phenomena (such as simple harmonic motion—see http://en.wikipedia.org/wiki/Simple_harmonic_motion).

Java's floating-point calculations are capable of returning `+infinity`, `-infinity`, `+0.0`, `-0.0`, and `NaN` because Java largely conforms to IEEE 754 (http://en.wikipedia.org/wiki/IEEE_754), a standard for floating-point calculations. The following are the circumstances under which these special values arise:

- `+infinity` returns from attempting to divide a positive number by `0.0`.
For example, `System.out.println(1.0 / 0.0);` outputs `Infinity`.
- `-infinity` returns from attempting to divide a negative number by `0.0`. For example, `System.out.println(-1.0 / 0.0);` outputs `-Infinity`.
- `NaN` returns from attempting to divide `0.0` by `0.0`, attempting to calculate the square root of a negative number, and attempting other strange operations. For example, `System.out.println(0.0 / 0.0);` and `System.out.println(Math.sqrt(-1.0));` each output `NaN`.
- `+0.0` results from attempting to divide a positive number by `+infinity`.
For example, `System.out.println(1.0 / (1.0 / 0.0));` outputs `0.0` (`+0.0` without the `+` sign).
- `-0.0` results from attempting to divide a negative number by `+infinity`.
For example, `System.out.println(-1.0 / (1.0 / 0.0));` outputs `-0.0`.

After an operation yields `+infinity`, `-infinity`, or `NaN`, the rest of the expression usually equals that special value. For example, `System.out.println(1.0 / 0.0 * 20.0);` outputs `Infinity`. Also, an expression that first yields `+infinity` or `-infinity` might devolve into `NaN`. For example, expression `1.0 / 0.0 * 0.0` first yields `+infinity` (`1.0 / 0.0`) and then yields `NaN` (`+infinity * 0.0`).

Another curiosity is `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Long.MAX_VALUE`, and `Long.MIN_VALUE`. Each of these constants identifies the maximum or minimum value that can be represented by the class's associated primitive type. (You'll learn more about these classes later in this chapter.)

Finally, you might wonder why the `abs()`, `max()`, and `min()` overloaded methods don't include `byte` and `short` versions, as in `byte abs(byte b)` and `short abs(short s)`. There is no need for these methods because the limited ranges of bytes and short integers make them unsuitable in calculations. If you need such a method, check out Listing 7-1.

Listing 7-1. Obtaining Absolute Values for Byte Integers and Short Integers

```
public class AbsByteShort {  
    static public byte abs(byte b) {  
        return (b < 0) ? (byte) -b : b;  
    }  
  
    static public short abs(short s) {  
        return (s < 0) ? (short) -s : s;  
    }  
  
    public static void main(String[] args) {  
        byte b = -2;  
        System.out.println(abs(b)); // Output: 2  
        short s = -3;  
        System.out.println(abs(s)); // Output: 3  
    }  
}
```

Listing 7-1's `(byte)` and `(short)` casts are necessary because `-b` converts `b`'s value from a `byte` to an `int`, and `-s` converts `s`'s value from a `short` to an `int`. In contrast, these casts are not needed with `(b < 0)` and `(s < 0)`, which automatically cast `b`'s and `s`'s values to an `int` before comparing them with `int`-based 0.

Exploring Number and Its Children

The abstract `java.lang.Number` class is the superclass of those classes representing numeric values that are convertible to the byte integer, double-precision floating-point, floating-point, integer, long integer, and short integer primitive types. This class offers the following conversion methods:

- `byte byteValue()`
- `double doubleValue()`
- `float floatValue()`
- `int intValue()`
- `long longValue()`
- `short shortValue()`

You typically don't work with `Number` directly, unless you've created a collection or array of `Number` subclass objects and plan to iterate over this collection/array, calling one of the conversion methods on each stored instance. Instead, you would typically work with one of the following subclasses:

- `java.util.concurrent.atomic.AtomicInteger`
- `java.util.concurrent.atomic.AtomicLong`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.lang.Byte`
- `java.lang.Double`
- `java.lang.Float`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Short`

We'll shortly discuss all of these types except for `AtomicInteger` and `AtomicLong`, which we'll discuss in Chapter 11.

BigDecimal

In Chapter 3, we introduced a `SavingsAccount` class with a `balance` field of type `int`. This field records the number of dollars in this account. Alternatively, it could represent the number of pennies that the account contains.

Perhaps you are wondering why we didn't declare `balance` to be of type `double` or `float`. That way, `balance` could store values such as 18.26 (18 dollars in the whole number part and 26 pennies in the fraction part). We didn't declare `balance` to be a `double` or `float` for the following reasons:

- Not all floating-point values that can represent monetary amounts (dollars and cents) can be stored exactly in memory. For example, 0.1 (which you might use to represent 10 cents) has no exact storage representation. If you executed `double total = 0.1; for (int i = 0; i < 50; i++) total += 0.1; System.out.println(total);`, you would observe `5.099999999999998` instead of the correct `5.1` as the output.
- The result of each floating-point calculation needs to be rounded to the nearest cent. Failure to do so introduces tiny errors that can cause the final result to differ from the correct result. Although `Math` supplies a pair of `round()` methods that you might consider using to round a calculation to the nearest cent, these methods round to the nearest integer (dollar).

Listing 7-2's `InvoiceCalc` application demonstrates both problems. However, the first problem isn't serious because it contributes very little to the inaccuracy. The more serious problem occurs from failing to round to the nearest cent after performing a calculation.

Listing 7-2. Floating-Point-Based Invoice Calculations Leading to Confusing Results

```
import java.text.NumberFormat;

public class InvoiceCalc {
    public final static double DISCOUNT_PERCENT = 0.1; // 10%
    public final static double TAX_PERCENT = 0.05; // 5%
```

```

public static void main(String[] args) {
    double invoiceSubtotal = 285.36;
    double discount = invoiceSubtotal * DISCOUNT_PERCENT;
    double subtotalBeforeTax = invoiceSubtotal - discount;
    double salesTax = subtotalBeforeTax * TAX_PERCENT;
    double invoiceTotal = subtotalBeforeTax + salesTax;
    NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();
    System.out.println("Subtotal: " + currencyFormat.format(invoiceSubtotal));
    System.out.println("Discount: " + currencyFormat.format(discount));
    System.out.println("SubTotal after discount: " +
                       currencyFormat.format(subtotalBeforeTax));
    System.out.println("Sales Tax: " + currencyFormat.format(salesTax));
    System.out.println("Total: " + currencyFormat.format(invoiceTotal));
}
}

```

Listing 7-2 performs several invoice-related calculations that result in an incorrect final total. After performing these calculations, it obtains a currency-based formatter for formatting double-precision floating-point values into string-based monetary amounts with a currency symbol (such as the dollar sign [\$]). The formatter is obtained by calling the `java.text.NumberFormat` class's `NumberFormat.getCurrencyInstance()` method. A value is then formatted into a currency string by passing this value as an argument to `NumberFormat`'s `String format(double value)` method.

When you run `InvoiceCalc`, you will discover the following output:

```

Subtotal: $285.36
Discount: $28.54
SubTotal after discount: $256.82
Sales Tax: $12.84
Total: $269.67

```

This output reveals the correct subtotal, discount, subtotal after discount, and sales tax. In contrast, it incorrectly gives 269.67 instead of 269.66 as the final total. The customer will probably not appreciate paying an extra penny, even though 269.67 is the correct value according to the floating-point calculations:

```
Subtotal: 285.36
Discount: 28.536
SubTotal after discount: 256.824
Sales Tax: 12.8412
Total: 269.6652
```

The problem arises from not rounding the result of each calculation to the nearest cent before performing the next calculation. As a result, the 0.024 in 256.824 and 0.0012 in 12.84 contribute to the final value, causing `NumberFormat's format()` method to round this value to 269.67.

Caution Never use `float` or `double` to represent monetary values.

Java provides a solution to both problems in the form of a `BigDecimal` class. This immutable class (a `BigDecimal` instance cannot be modified) represents a signed decimal number (such as 23.653) of arbitrary *precision* (number of digits) with an associated *scale* (an integer that specifies the number of digits after the decimal point).

`BigDecimal` declares three convenience constants: `ONE`, `TEN`, and `ZERO`. Each constant is the `BigDecimal` equivalent of 1, 10, and 0 with a zero scale.

Caution `BigDecimal` declares several `ROUND_-prefixed` constants. These constants are largely obsolete and should be avoided, along with the public `BigDecimal divide(BigDecimal divisor, int scale, int roundingMode)` and public `BigDecimal setScale(int newScale, int roundingMode)` methods, which are still present so that dependent legacy code continues to compile.

`BigDecimal` also declares a variety of useful constructors and methods which are listed in the API documentation.

The best way to get comfortable with `BigDecimal` is to try it out. Listing 7-3 uses this class to correctly perform the invoice calculations that were presented in Listing 7-2.

Listing 7-3. BigDecimal-Based Invoice Calculations Not Leading to Confusing Results

```
import java.math.BigDecimal;
import java.math.RoundingMode;

public class InvoiceCalc {
    public static void main(String[] args) {
        BigDecimal invoiceSubtotal = new BigDecimal("285.36");
        BigDecimal discountPercent = new BigDecimal("0.10");
        BigDecimal discount = invoiceSubtotal.multiply(discountPercent);
        discount = discount.setScale(2, RoundingMode.HALF_UP);
        BigDecimal subtotalBeforeTax = invoiceSubtotal.subtract(discount);
        subtotalBeforeTax = subtotalBeforeTax.setScale(2, RoundingMode.HALF_UP);
        BigDecimal salesTaxPercent = new BigDecimal("0.05");
        BigDecimal salesTax = subtotalBeforeTax.multiply(salesTaxPercent);
        salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
        BigDecimal invoiceTotal = subtotalBeforeTax.add(salesTax);
        invoiceTotal = invoiceTotal.setScale(2, RoundingMode.HALF_UP);
        System.out.println("Subtotal: " + invoiceSubtotal);
        System.out.println("Discount: " + discount);
        System.out.println("SubTotal after discount: " + subtotalBeforeTax);
        System.out.println("Sales Tax: " + salesTax);
        System.out.println("Total: " + invoiceTotal);
    }
}
```

Listing 7-3's `main()` method first creates `BigDecimal` objects `invoiceSubtotal` and `discountPercent` that are initialized to 285.36 and 0.10, respectively. It multiplies `invoiceSubtotal` by `discountPercent` and assigns the `BigDecimal` result to `discount`.

At this point, `discount` contains 28.5360. Apart from the trailing zero, this value is the same as that generated by `invoiceSubtotal * DISCOUNT_PERCENT` in Listing 7-2. The value that should be stored in `discount` is 28.54. To correct this problem before performing another calculation, `main()` calls `discount's setScale()` method with these arguments:

- 2: Two digits after the decimal point
- `RoundingMode.HALF_UP`: The conventional approach to rounding

After setting the scale and proper rounding mode, `main()` subtracts `discount` from `invoiceSubtotal` and assigns the resulting `BigDecimal` instance to `subtotalBeforeTax`. `main()` calls `setScale()` on `subtotalBeforeTax` to properly round its value before moving on to the next calculation.

`main()` next creates a `BigDecimal` object named `salesTaxPercent` that is initialized to `0.05`. It then multiplies `subtotalBeforeTax` by `salesTaxPercent`, assigning the result to `salesTax`, and calls `setScale()` on this `BigDecimal` object to properly round its value.

Moving on, `main()` adds `salesTax` to `subtotalBeforeTax`, saving the result in `invoiceTotal`, and rounds the result via `setScale()`. The values in these objects are sent to the standard output device via `System.out.println()`, which calls their `toString()` methods to return string representations of the `BigDecimal` values.

When you run this new version of `InvoiceCalc`, you will discover the following output:

Subtotal: 285.36

Discount: 28.54

SubTotal after discount: 256.82

Sales Tax: 12.84

Total: 269.66

Caution `BigDecimal` declares a `BigDecimal(double val)` constructor that you should avoid using if at all possible. This constructor initializes the `BigDecimal` instance to the value stored in `val`, making it possible for this instance to reflect an invalid representation when the double cannot be stored exactly. For example, `BigDecimal(0.1)` results in `0.100000000000000055511151231257827021181583404541015625` being stored in the instance. In contrast, `BigDecimal("0.1")` stores `0.1` exactly.

BigInteger

`BigDecimal` stores a signed decimal number as an unscaled value with a 32-bit integer scale. The unscaled value is stored in an instance of the `BigInteger` class.

`BigInteger` is an immutable class that represents a signed integer of arbitrary precision. It stores its value in *two's complement format* (all bits are flipped—1s to 0s and 0s to 1s—and 1 is added to the result to be compatible with the two's complement format used by Java's byte integer, short integer, integer, and long integer types).

Note Check out Wikipedia’s “Two’s complement” entry ([http://en.wikipedia.org/wiki/Two's_complement](http://en.wikipedia.org/wiki/Two%27s_complement)) to learn more about two’s complement.

`BigInteger` declares three convenience constants: `ONE`, `TEN`, and `ZERO`. Each constant is the `BigInteger` equivalent of 1, 10, and 0.

BigInteger also declares a variety of useful constructors and methods listed in the API documentation.

Note `BigInteger` also declares several bit-oriented methods, such as `BigInteger and(BigInteger val)`, `BigInteger flipBit(int n)`, and `BigInteger shiftLeft(int n)`. These methods are useful for when you need to perform low-level bit manipulation.

The best way to get comfortable with `BigInteger` is to try it out. Listing 7-4 uses this class in a `factorial()` method comparison context.

Listing 7-4. Comparing `factorial()` Methods

```
import java.math.BigInteger;

public class FactComp {
    public static void main(String[] args) {
        System.out.println(factorial(12));
        System.out.println();
        System.out.println(factorial(20L));
        System.out.println();
        System.out.println(factorial(170.0));
        System.out.println();
        System.out.println(factorial(new BigInteger("170"))));
        System.out.println();
        System.out.println(factorial(25.0));
        System.out.println();
        System.out.println(factorial(new BigInteger("25"))));
    }
}
```

```
public static int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n - 1);  
}  
  
public static long factorial(long n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n - 1);  
}  
  
public static double factorial(double n) {  
    if (n == 1.0)  
        return 1.0;  
    else  
        return n * factorial(n - 1);  
}  
  
public static BigInteger factorial(BigInteger n) {  
    if (n.equals(BigInteger.ZERO))  
        return BigInteger.ONE;  
    else  
        return n.multiply(factorial(n.subtract(BigInteger.ONE)));  
}  
}
```

Listing 7-4 compares four versions of the recursive `factorial()` method. This comparison reveals the largest argument that can be passed to each of the first three methods before the returned factorial value becomes meaningless because of limits on the range of values that can be accurately represented by the numeric type.

The first version is based on `int` and has a useful argument range of 0 through 12. Passing any argument greater than 12 results in a factorial that cannot be represented accurately as an `int`.

You can increase the useful range of `factorial()`, but not by much, by changing the parameter and return types to `long`. After making these changes, you will discover that the upper limit of the useful range is 20.

To further increase the useful range, you might create a version of `factorial()` whose parameter and return types are `double`. This is possible because whole numbers can be represented exactly as doubles. However, the largest useful argument that can be passed is 170.0. Anything higher than this value results in `factorial()` returning `+infinity`.

It's possible that you might need to calculate a higher factorial value, perhaps in the context of calculating a statistics problem involving combinations or permutations. The only way to accurately calculate this value is to use a version of `factorial()` based on `BigInteger`.

When you run the previous application, it generates the following output:

479001600

2432902008176640000

7.257415615307994E306

72574156153079989673967282111292631147169916812964513765435777989005618434
0170615785235074924261745951149099123783852077666602256544275302532890077
3207510902400430280058295603966612599658257104398558294257568966313439612
2625710949468067112055688804571933402126614528000000000000000000000000000000000000
000000000000000

1.5511210043330986E25

15511210043330985984000000

The first three values represent the highest factorials that can be returned by the `int`-based, `long`-based, and `double`-based `factorial()` methods. The fourth value represents the `BigInteger` equivalent of the highest `double` factorial.

Notice that the `double` method fails to accurately represent 170! (! is the math symbol for factorial). Its precision is simply too small. Although the method attempts to round the smallest digit, rounding doesn't always work—the number ends in 7994 instead of 7998. Rounding is only accurate up to argument 25.0, as the last two output lines reveal.

Note RSA encryption (http://en.wikipedia.org/wiki/RSA_algorithm) offers another use for BigInteger.

Primitive Type Wrapper Classes

Byte, Double, Float, Integer, Long, and Short along with Boolean and Character are known as *primitive type wrapper classes* or *value classes* because their instances wrap themselves around values of primitive types. Java provides these eight primitive type wrapper classes for two reasons:

- The Collections Framework (discussed in Chapter 9) provides lists, sets, and maps that can only store objects; they cannot store primitive-type values. You store a primitive-type value in a primitive type wrapper class instance and store the instance in the collection.
- These classes provide a good place to associate useful constants (such as MAX_VALUE and MIN_VALUE) and class methods (such as Integer's parseInt() methods and Character's isDigit(), isLetter(), and toUpperCase() methods) with the primitive types.

In this section, we will introduce you to each of these primitive type wrapper classes.

Boolean

Boolean is the smallest of the primitive type wrapper classes. This class declares three constants, including TRUE and FALSE, which denote precreated Boolean objects. It also declares a pair of constructors for initializing a Boolean object:

- Boolean(boolean value) initializes the Boolean object to value.
- Boolean(String s) converts s's text to a true or false value and stores this value in the Boolean object.

The second constructor compares s's value with true. Because the comparison is case insensitive, any uppercase/lowercase combination of these four letters (such as true, TRUE, or tRue) results in true being stored in the object. Otherwise, the constructor stores false in the object.

Note Boolean's constructors are complemented by boolean `booleanValue()`, which returns the wrapped Boolean value.

To learn about Boolean's methods, please consult the API documentation.

Caution Newcomers to the Boolean class often think that the `getBoolean()` method returns a Boolean object's true/false value. However, `getBoolean()` returns the value of a Boolean-based system property. We will discuss system properties later in this chapter. If you need to return a Boolean object's true/false value, use the `booleanValue()` method instead.

It's usually better to use `TRUE` and `FALSE` than to create Boolean objects.

Tip You should strive to create as few objects as possible. Not only will your applications have smaller memory footprints, they'll perform better because the garbage collector will not run as often.

Character

Character is the largest of the primitive type wrapper classes, containing many constants, a constructor, many methods, and a pair of nested classes (`Subset` and `UnicodeBlock`).

Note Character's complexity derives from Java's support for Unicode (<http://en.wikipedia.org/wiki/Unicode>). For brevity, we ignore much of Character's Unicode-related complexity, which is beyond the scope of this chapter.

Character declares a single `Character(char value)` constructor, which you use to initialize a Character object to `value`. This constructor is complemented by `char charValue()`, which returns the wrapped character value.

When you start writing applications, you might codify expressions such as `ch >= '0' && ch <= '9'` (test `ch` to see if it contains a digit) and `ch >= 'A' && ch <= 'Z'` (test `ch` to see if it contains an uppercase letter). You should avoid doing so for three reasons:

- It's too easy to introduce a bug into the expression. For example, `ch > '0' && ch <= '9'` introduces a subtle bug that doesn't include '0' in the comparison.
- The expressions are not very descriptive of what they are testing.
- The expressions are biased toward Latin digits (0–9) and letters (A–Z and a–z). They don't take into account digits and letters that are valid in other languages. For example, '\u0beb' is a character literal representing one of the digits in the Tamil language.

`Character` declares several comparison and conversion class methods that address these concerns. These methods include the following:

- `static boolean isDigit(char ch)` returns true when `ch` contains a digit (typically 0 through 9 but also digits in other alphabets).
- `static boolean isLetter(char ch)` returns true when `ch` contains a letter (typically A–Z or a–z but also letters in other alphabets).
- `static boolean isLetterOrDigit(char ch)` returns true when `ch` contains a letter or digit (typically A–Z, a–z, or 0–9 but also letters or digits in other alphabets).
- `static boolean isLowerCase(char ch)` returns true when `ch` contains a lowercase letter.
- `static boolean isUpperCase(char ch)` returns true when `ch` contains an uppercase letter.
- `static boolean isWhitespace(char ch)` returns true when `ch` contains a whitespace character (typically a space, a horizontal tab, a carriage return, or a line feed).
- `static char toLowerCase(char ch)` returns the lowercase equivalent of `ch`'s uppercase letter; otherwise, this method returns `ch`'s value.
- `static char toUpperCase(char ch)` returns the uppercase equivalent of `ch`'s lowercase letter; otherwise, this method returns `ch`'s value.

For example, `isDigit(ch)` is preferable to `ch >= '0' && ch <= '9'` because it avoids a source of bugs, is more readable, and returns true for non-Latin digits (such as '`\u0beb`') and Latin digits.

Float and Double

`Float` and `Double` store floating-point and double-precision floating-point values in `Float` and `Double` objects, respectively. These classes declare the following constants:

- `MAX_VALUE` identifies the maximum value that can be represented as a `float` or `double`.
- `MIN_VALUE` identifies the minimum value that can be represented as a `float` or `double`.
- `NaN` represents `0.0F / 0.0F` as a `float` and `0.0 / 0.0` as a `double`.
- `NEGATIVE_INFINITY` represents `-infinity` as a `float` or `double`.
- `POSITIVE_INFINITY` represents `+infinity` as a `float` or `double`.

`Float` and `Double` also declare the following constructors for initializing their objects:

- `Float(float value)` initializes the `Float` object to `value`.
- `Float(double value)` initializes the `Float` object to the `float` equivalent of `value`.
- `Float(String s)` converts `s`'s text to a floating-point value and stores this value in the `Float` object.
- `Double(double value)` initializes the `Double` object to `value`.
- `Double(String s)` converts `s`'s text to a double-precision floating-point value and stores this value in the `Double` object.

`Float`'s constructors are complemented by `float floatValue()`, which returns the wrapped floating-point value. Similarly, `Double`'s constructors are complemented by `double doubleValue()`, which returns the wrapped double-precision floating-point value.

Float declares several utility methods in addition to floatValue(). These methods include the following:

- `static int floatToIntBits(float value)` converts value to a 32-bit integer.
- `static boolean isInfinite(float f)` returns true when f's value is +infinity or -infinity. A related boolean `isInfinite()` method returns true when the current Float object's value is +infinity or -infinity.
- `static boolean isNaN(float f)` returns true when f's value is NaN. A related boolean `isNaN()` method returns true when the current Float object's value is NaN.
- `static float parseFloat(String s)` parses s, returning the floating-point equivalent of s's textual representation of a floating-point value or throwing NumberFormatException when this representation is invalid (e.g., contains letters).

Double declares several utility methods as well as doubleValue(). These methods include the following:

- `static long doubleToLongBits(double value)` converts value to a long integer.
- `static boolean isInfinite(double d)` returns true when d's value is +infinity or -infinity. A related boolean `isInfinite()` method returns true when the current Double object's value is +infinity or -infinity.
- `static boolean isNaN(double d)` returns true when d's value is NaN. A related public boolean `isNaN()` method returns true when the current Double object's value is NaN.
- `static double parseDouble(String s)` parses s, returning the double-precision floating-point equivalent of s's textual representation of a double-precision floating-point value or throwing NumberFormatException when this representation is invalid.

The `floatToIntBits()` and `doubleToIntBits()` methods are used in implementations of the `equals()` and `hashCode()` methods that must take `float` and `double` fields into account. `floatToIntBits()` and `doubleToIntBits()` allow `equals()` and `hashCode()` to respond properly to the following situations:

- `equals()` must return true when `f1` and `f2` contain `Float.NaN` (or `d1` and `d2` contain `Double.NaN`). If `equals()` was implemented in a manner similar to `f1.floatValue() == f2.floatValue()` (or `d1.doubleValue() == d2.doubleValue()`), this method would return false because `NaN` is not equal to anything, including itself.
- `equals()` must return false when `f1` contains `+0.0` and `f2` contains `-0.0` (or vice versa), or `d1` contains `+0.0` and `d2` contains `-0.0` (or vice versa). If `equals()` was implemented in a manner similar to `f1.floatValue() == f2.floatValue()` (or `d1.doubleValue() == d2.doubleValue()`), this method would return true because `+0.0 == -0.0` returns true.

These requirements are needed for hash-based collections (discussed in Chapter 9) to work properly. Listing 7-5 shows how they impact `Float`'s and `Double`'s `equals()` methods.

Listing 7-5. Demonstrating `Float`'s `equals()` Method in a `NaN` Context and `Double`'s `equals()` Method in a `+/-0.0` Context

```
public class FloatDoubleDemo {
    public static void main(String[] args) {
        Float f1 = new Float(Float.NaN);
        System.out.println(f1.floatValue());
        Float f2 = new Float(Float.NaN);
        System.out.println(f2.floatValue());
        System.out.println(f1.equals(f2));
        System.out.println(Float.NaN == Float.NaN);
        System.out.println();
        Double d1 = new Double(+0.0);
        System.out.println(d1.doubleValue());
        Double d2 = new Double(-0.0);
        System.out.println(d2.doubleValue());
```

```

        System.out.println(d1.equals(d2));
        System.out.println(+0.0 == -0.0);
    }
}

```

Compile Listing 7-5 (`javac FloatDoubleDemo.java`) and run this application (`java FloatDoubleDemo`). The following output proves that `Float`'s `equals()` method properly handles NaN and `Double`'s `equals()` method properly handles `+/-0.0`:

```

NaN
NaN
true
false

0.0
-0.0
false
true

```

Tip When you want to test a `float` or `double` value for equality with `+infinity` or `-infinity` (but not both), don't use `isInfinite()`. Instead, compare the value with `NEGATIVE_INFINITY` or `POSITIVE_INFINITY` via `==`. For example, `f == Float.NEGATIVE_INFINITY`.

You will find `parseFloat()` and `parseDouble()` useful in many contexts. For example, Listing 7-6 uses `parseDouble()` to parse command-line arguments into doubles.

Listing 7-6. Parsing Command-Line Arguments into Double-Precision Floating-Point Values

```

public class Calc {
    public static void main(String[] args) {
        if (args.length != 3) {
            System.err.println("usage: java Calc value1 op value2");
            System.err.println("op is one of +, -, *, or /");
            return;
        }
    }
}

```

```

try {
    double value1 = Double.parseDouble(args[0]);
    double value2 = Double.parseDouble(args[2]);
    if (args[1].equals("+"))
        System.out.println(value1 + value2);
    else
        if (args[1].equals("-"))
            System.out.println(value1 - value2);
        else if (args[1].equals("x"))
            System.out.println(value1 * value2);
        else if (args[1].equals("/"))
            System.out.println(value1 / value2);
        else
            System.err.println("invalid operator: " + args[1]);
} catch (NumberFormatException nfe) {
    System.err.println("Bad number format: " + nfe.getMessage());
}
}
}
}

```

Specify `java Calc 10E+3 + 66.0` to try out the `Calc` application. This application responds by outputting `10066.0`. If you specified `java Calc 10E+3 + A` instead, you would observe `Bad number format: For input string: "A"` as the output, which is in response to the second `parseDouble()` method call's throwing of a `NumberFormatException` object.

Although `NumberFormatException` describes an unchecked exception, and although unchecked exceptions are often not handled because they represent coding mistakes, `NumberFormatException` doesn't fit this pattern in this example. The exception doesn't arise from a coding mistake; it arises from someone passing an illegal numeric argument to the application, which cannot be avoided through proper coding. Perhaps `NumberFormatException` should have been implemented as a checked exception.

Integer, Long, Short, and Byte

`Integer`, `Long`, `Short`, and `Byte` store 32-bit, 64-bit, 16-bit, and 8-bit integer values in `Integer`, `Long`, `Short`, and `Byte` objects, respectively.

Each class declares `MAX_VALUE` and `MIN_VALUE` constants that identify the maximum and minimum values that can be represented by its associated primitive type. For all constructors and methods, please consult the API documentation.

Exploring String, StringBuffer, and StringBuilder

Many computer languages implement the concept of a *string*, a sequence of characters treated as a single unit (and not as individual characters). For example, the C language implements a string as an array of characters terminated by the null character ('`\0`'). In contrast, Java implements a string via the `java.lang.String` class.

`String` objects are immutable: you cannot modify a `String` object's string. The various `String` methods that appear to modify the `String` object actually return a new `String` object with modified string content instead. Because returning new `String` objects is often wasteful, Java provides the `java.lang.StringBuffer` and equivalent `java.lang.StringBuilder` classes as a workaround.

This section introduces you to `String`, `StringBuffer`, and `StringBuilder`.

String

`String` represents a string as a sequence of characters. In contrast to strings in the C language, this sequence is not terminated by a null character. Instead, its length is stored separately.

You typically obtain a `String` by assigning a string literal to a variable of `String` type, for example, `String favLanguage = "Java";`. You can also obtain a `String` by calling a `String` constructor, for example, `String favLanguage = new String("Java");`.

After obtaining a `String`, you can invoke methods to accomplish various tasks. For example, you can obtain a string's length by invoking the `length()` method, for example, `favLanguage.length()`.

The API documentation describes all of `String`'s constructors and methods for initializing `String` objects and working with strings.

The API reveals a couple of interesting items about `String`. First, this class's `String(String s)` constructor doesn't initialize a `String` object to a string literal, as in `new String("Java")`. Instead, it behaves similarly to the C++ copy constructor by initializing the `String` object to the contents of another `String` object. This behavior suggests that a string literal is more than it appears to be.

In reality, a string literal is a `String` object. You can prove this to yourself by executing `System.out.println("abc".length());` and `System.out.println("abc" instanceof String);`. The first method call outputs 3, which is the length of the "abc" `String` object's string, and the second method call outputs true ("abc" is a `String` object).

Note String literals are stored in a class file data structure known as the *constant pool*. When a class is loaded, a `String` object is created for each literal and is stored in an internal table of `String` objects.

The second interesting item is the `intern()` method, which *interns* (stores a unique copy of) a `String` object in an internal table of `String` objects. `intern()` makes it possible to compare strings via their references and `==` or `!=`. These operators are the fastest way to compare strings, which is especially valuable when sorting a huge number of strings.

Listing 7-7 demonstrates these concepts.

Listing 7-7. Demonstrating a Couple of Interesting Things About Strings

```
public class StringDemo {
    public static void main(String[] args) {
        System.out.println("abc".length());
        System.out.println("abc" instanceof String);
        System.out.println("abc" == "a" + "bc");
        System.out.println("abc" == new String("abc"));
        System.out.println("abc" == new String("abc").intern());
    }
}
```

Compile Listing 7-7 (`javac StringDemo.java`) and run this application (`java StringDemo`). You should observe the following output:

```
3
true
true
false
true
```

By default, `String` objects denoted by literal strings ("abc") and string-valued constant expressions ("a" + "bc") are interned, which is why `System.out.println("abc" == "a" + "bc")`; outputs true. However, `String` objects created via `String` constructors are not interned, which is why `System.out.println("abc" == new String("abc"))`; outputs false. In contrast, `System.out.println("abc" == new String("abc").intern())`; outputs true.

Caution Be careful with this string comparison technique (which only compares references) because you can easily introduce a bug when one of the strings being compared has not been interned. To make your intents clear, use the `equals()` or `equalsIgnoreCase()` method. For example, each of `"abc".equals(new String("abc"))` and `"abc".equalsIgnoreCase(new String("ABC"))` returns true.

The documentation also reveals the `charAt()` method, which is useful for extracting a string's characters. Listing 7-8 offers a demonstration.

Listing 7-8. Iterating over a String

```
public class StringDemo {
    public static void main(String[] args) {
        String s = "abc";
        for (int i = 0; i < s.length(); i++)
            System.out.println(s.charAt(i));
    }
}
```

Compile Listing 7-8 and run this application. You will observe that `for (int i = 0; i < s.length(); i++) System.out.println(s.charAt(i));` returns each of s's a, b, and c characters and outputs it on a separate line.

Finally, `String` has `split()`, a method that we employed in Chapter 6's `StubFinder` application to split a string's comma-separated list of values into an array of `String` objects. This method uses a regular expression that identifies a sequence of characters around which the string is split. (We will discuss regular expressions in Chapter 14.)

StringBuffer and StringBuilder

`String` objects are immutable: you cannot modify a `String` object's string. The `String` methods that appear to modify the `String` object (such as `replace()`) actually return a new `String` object with modified string content instead. Because returning new `String` objects is often wasteful, Java provides the `StringBuffer` and `StringBuilder` classes as a workaround.

`StringBuffer` and `StringBuilder` are identical apart from the fact that `StringBuilder` offers better performance than `StringBuffer` but cannot be used in the context of multiple threads without explicit synchronization. (We discuss threads and synchronization later in this chapter.)

Tip Use `StringBuffer` in a multithreaded context (for safety) and `StringBuilder` in a single-threaded context (for performance).

`StringBuffer` and `StringBuilder` provide an internal character array for building a string efficiently. After creating a `StringBuffer/StringBuilder` object, you call various methods to append, delete, and insert the character representations of various values to, from, and into the array. You then call `toString()` to convert the array's content to a `String` object and return this object.

The API documentation describes all `StringBuffer`'s constructors and methods for initializing `StringBuffer` objects and working with string buffers.

A `StringBuffer` or `StringBuilder` object's internal array is associated with the concepts of capacity and length. *Capacity* refers to the maximum number of characters that can be stored in the array before the array grows to accommodate additional characters. *Length* refers to the number of characters that are already stored in the array.

Listing 7-9 demonstrates `StringBuffer()`, various `append()` methods, and `toString()`.

Listing 7-9. Demonstrating `StringBuffer`

```
public class StringBufferDemo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello,");
        sb.append(' ');
        sb.append("world. ");
```

```

        sb.append(args.length);
        sb.append(" argument(s) have been passed to this method.");
        String s = sb.toString();
        System.out.println(s);
    }
}

```

After creating a `StringBuffer` object initialized to the contents of the interned "Hello," `String` object, `main()` appends a character, another `String` object, an integer (identifying the number of command-line arguments passed to this application), and yet another `String` object to this buffer. It then converts the `StringBuffer`'s contents to a `String` object and prints this object's content on the standard output device.

Compile Listing 7-9 (`javac StringBufferDemo.java`) and run this application (`java StringBufferDemo`). You should observe the following output:

```
Hello, world. 0 argument(s) have been passed to this method.
```

Rerun this application with one or more command-line arguments and the number will change to reflect how many command-line arguments were passed.

Consider a scenario where you've written code to format an integer value into a string. As part of the formatter, you need to prepend a specific number of leading spaces to the integer. You decide to use the following initialization code and loop to build a `spacesPrefix` string with three leading spaces:

```

int numLeadingSpaces = 3; // default value
String spacesPrefix = "";
for (int j = 0; j < numLeadingSpaces; j++)
    spacesPrefix += "0";

```

This loop is inefficient because each of the iterations creates a `StringBuilder` object and a `String` object. The compiler transforms this code fragment into the following fragment:

```

int numLeadingSpaces = 3; // default value
String spacesPrefix = "";
for (int j = 0; j < numLeadingSpaces; j++)
    spacesPrefix = new StringBuilder().append(spacesPrefix).append("0").
        toString();

```

A more efficient way to code the previous loop involves creating a `StringBuilder/StringBuffer` object prior to entering the loop, calling the appropriate `append()` method in the loop, and calling `toString()` after the loop. The following code fragment demonstrates this more efficient scenario:

```
int numLeadingSpaces = 3; // default value
StringBuilder sb = new StringBuilder();
for (int j = 0; j < numLeadingSpaces; j++)
    sb.append('0');
String spacesPrefix = sb.toString();
```

Caution Avoid using the string concatenation operator in a lengthy loop because it results in the creation of many unnecessary `StringBuilder` and `String` objects.

Exploring System

The `java.lang.System` utility class declares class methods that provide access to the current time (in milliseconds), system property values, environment variable values, and other kinds of system information. Furthermore, it declares class methods that support the system tasks of copying one array to another array, requesting garbage collection, and so on.

Note `System` declares `SecurityManager getSecurityManager()` and `void setSecurityManager(SecurityManager sm)` methods that are not supported by Android. On an Android device, the former method always returns `null`, and the latter method always throws an instance of the `java.lang.SecurityException` class. Regarding the latter method, its documentation states that “security managers do not provide a secure environment for executing untrusted code and are unsupported on Android. Untrusted code cannot be safely isolated within a single virtual machine on Android.”

Listing 7-10 demonstrates the `arraycopy()`, `currentTimeMillis()`, and `getProperty()` methods.

Listing 7-10. Experimenting with System Methods

```
public class SystemDemo {  
    public static void main(String[] args) {  
        int[] grades = { 86, 92, 78, 65, 52, 43, 72, 98, 81 };  
        int[] gradesBackup = new int[grades.length];  
        System.arraycopy(grades, 0, gradesBackup, 0, grades.length);  
        for (int i = 0; i < gradesBackup.length; i++)  
            System.out.println(gradesBackup[i]);  
        System.out.println("Current time: " + System.currentTimeMillis());  
        String[] propNames = {  
            "file.separator",  
            "java.class.path",  
            "java.home",  
            "java.io.tmpdir",  
            "java.library.path",  
            "line.separator",  
            "os.arch",  
            "os.name",  
            "path.separator",  
            "user.dir"  
        };  
        for (int i = 0; i < propNames.length; i++)  
            System.out.println(propNames[i] + ": " +  
                System.getProperty(propNames[i]));  
    }  
}
```

Listing 7-10's `main()` method begins by demonstrating `arraycopy()`. It uses this method to copy the contents of a `grades` array to a `gradesBackup` array.

Tip The `arraycopy()` method is the fastest portable way to copy one array to another. Also, when you write a class whose methods return a reference to an internal array, you should use `arraycopy()` to create a copy of the array and then return the copy's reference. That way, you prevent clients from directly manipulating (and possibly screwing up) the internal array.

`main()` next calls `currentTimeMillis()` to return the current time as a milliseconds value. Because this value is not human-readable, you might want to use the `java.util.Date` class. The `Date()` constructor calls `currentTimeMillis()` and its `toString()` method converts this value to a readable date and time.

`main()` concludes by demonstrating `getProperty()` in a for loop. This loop iterates over some property names, outputting each name and value.

Compile Listing 7-10 (`javac SystemDemo.java`) and run this application (`java SystemDemo`). When we run this application on our platform, it generates the following output:

```
86
92
78
65
52
43
72
98
81
Current time: 1353115138889
file.separator: \
java.class.path: .;C:\Program Files (x86)\QuickTime\QTSystem\QTJava.zip
java.home: C:\Program Files\Java\jre11
java.io.tmpdir: C:\Users\Owner\AppData\Local\Temp\
java.library.path: C:\Windows\system32;C:\Windows\Sun\Java\bin;C:\Windows\system32;C:\Windows;c:\Program Files (x86)\AMD APP\bin\x86_64;c:\Program Files (x86)\AMD APP\bin\x86;c:\Program Files\Common Files\Microsoft Shared\Windows Live;c:\Program Files (x86)\Common Files\Microsoft Shared\Windows
```

```
Live;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-Static;C:\Program Files (x86)\Windows Live\Shared;C:\Program Files\java\jdk1.11.0_06\bin;C:\Program Files (x86)\Borland\BCC55\bin;C:\android;C:\android\tools;C:\android\platform-tools;C:\Program Files (x86)\apache-ant-1.9.14\bin;C:\Program Files (x86)\QuickTime\QTSystem\;.
```

line.separator:

```
os.arch: amd64  
os.name: Windows 10  
path.separator: ;  
user.dir: C:\prj\dev\ljfad2\ch08\code\SystemDemo
```

Note `line.separator` stores the actual line separator character/characters, not its/their representation (such as `\r\n`), which is why a blank line appears after `line.separator`:

Exploring Threads

Applications execute via *threads*, which are independent paths of execution through an application’s code. When multiple threads are executing, each thread’s path can differ from other thread paths. For example, a thread might execute one of a switch statement’s cases, and another thread might execute another of this statement’s cases.

Note Applications use threads to improve performance. Some applications can get by with only the *default main thread* (the thread that executes the `main()` method) to carry out their tasks, but other applications need additional threads to perform time-intensive tasks in the background, so that they remain responsive to their users.

The virtual machine gives each thread its own method-call stack to prevent threads from interfering with each other. Separate stacks let threads keep track of their next instructions to execute, which can differ from thread to thread. The stack also provides a thread with its own copy of method parameters, local variables, and return value.

Java supports threads via its Threads API. This API largely consists of one interface (`Runnable`) and four classes (`Thread`, `ThreadGroup`, `ThreadLocal`, and `InheritableThreadLocal`) in the `java.lang` package. After talking about `Runnable` and `Thread` (and mentioning `ThreadGroup` during that), we explore synchronization, `ThreadLocal`, and `InheritableThreadLocal`.

Note Java 5 introduced the `java.util.concurrent` package as a high-level alternative to the low-level Threads API. (We will discuss this package in Chapter 11.) Although `java.util.concurrent` is the preferred API for working with threads, you should also be somewhat familiar with Threads because it's helpful in simple threading scenarios. Also, you might have to analyze someone else's source code that depends on Threads.

Runnable and Thread

Java's `Runnable` interface identifies those objects that supply code for threads to execute via this interface's solitary `void run()` method—a thread receives no arguments and returns no value. In the following code fragment, an anonymous class implements `Runnable`:

```
Runnable r = new Runnable() {  
    @Override  
    public void run() {  
        // perform some work  
    }  
};
```

Java's `Thread` class provides a consistent interface to the underlying operating system's threading architecture. (The operating system is typically responsible for creating and managing threads.) A single operating system thread is associated with a `Thread` object.

`Thread` declares several constructors for initializing `Thread` objects. Some of these constructors take `Runnable` arguments. For example, `Thread(Runnable runnable)` initializes a new `Thread` object to the specified `runnable` whose code is to be executed, which the following code fragment demonstrates:

```
Thread t = new Thread(r);
```

Other constructors don't take `Runnable` arguments. For example, `Thread()` doesn't initialize `Thread` to a `Runnable` argument. You must extend `Thread` and override its `run()` method to supply the code to run, which the following code fragment accomplishes:

```
class MyThread extends Thread {
    @Override
    public void run() {
    }
}
```

In the absence of an explicit name argument, each constructor assigns a unique default name (starting with `Thread-`) to the `Thread` object. Names make it possible to differentiate threads. In contrast to the previous two constructors, which choose default names, `Thread(String threadName)` lets you specify your own thread name.

`Thread` also declares methods for starting and managing threads. The API documentation describes all those methods.

[Listing 7-11](#) introduces you to the Threads API via a `main()` method that demonstrates `Runnable`, `Thread(Runnable runnable)`, `currentThread()`, `getName()`, and `start()`.

Listing 7-11. A Pair of Counting Threads

```
public class CountingThreads {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (true)
                    System.out.println(name + ": " + count++);
            }
        };
        Thread thdA = new Thread(r);
        Thread thdB = new Thread(r);
```

```
    thdA.start();
    thdB.start();
}
}
```

According to Listing 7-11, the default main thread that executes `main()` first instantiates an anonymous class that implements `Runnable`. It then creates two `Thread` objects, initializing each object to the runnable, and calls `Thread`'s `start()` method to create and start both threads. After completing these tasks, the main thread exits `main()` and dies.

Each of the two started threads executes the runnable's `run()` method. It calls `Thread`'s `currentThread()` method to obtain its associated `Thread` instance, uses this instance to call `Thread`'s `getName()` method to return its name, initializes `count` to 0, and enters an infinite loop where it outputs name and `count`, and increments `count` on each iteration.

Tip To stop an application that doesn't end, press the Ctrl and C keys simultaneously on a Windows or Linux platform, or do the equivalent on other platforms.

We observed both threads alternating in their execution when we ran this application on the Windows platform. Partial output from one run appears here:

```
Thread-0: 0
Thread-0: 1
Thread-1: 0
Thread-0: 2
Thread-1: 1
Thread-0: 3
Thread-1: 2
Thread-0: 4
Thread-1: 3
Thread-0: 5
Thread-1: 4
Thread-0: 6
Thread-1: 5
```

```
Thread-0: 7
Thread-1: 6
Thread-1: 7
Thread-1: 8
Thread-1: 9
Thread-1: 10
Thread-1: 11
Thread-1: 12
```

Note Although the output shows that the first thread (Thread-0) starts executing, never assume that the thread associated with the Thread object whose start() method is called first will execute first.

When a computer has enough processors and/or processor cores, the computer's operating system assigns a separate thread to each processor or core so the threads execute simultaneously. When a computer doesn't have enough processors and/or cores, various threads must wait their turns to use the shared processors/cores.

Listing 7-12 refactors Listing 7-11's main() method to give each thread a nondefault name and to put each thread to sleep after outputting name and count.

Listing 7-12. A Pair of Counting Threads Revisited

```
public class CountingThreads {
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (true) {
                    System.out.println(name + ": " + count++);
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException ie) {
                    }
                }
            }
        };
        r.run();
    }
}
```

```
        }
    }
};

Thread thdA = new Thread(r);
thdA.setName("A");
Thread thdB = new Thread(r);
thdB.setName("B");
thdA.start();
thdB.start();
}
}
```

Listing 7-12 reveals that threads A and B execute `Thread.sleep(100)`; to sleep for 100 milliseconds. This sleep results in each thread executing more frequently, as the following partial output reveals:

```
A: 0
B: 0
A: 1
B: 1
A: 2
B: 2
A: 2
B: 3
A: 3
B: 4
A: 4
B: 5
A: 5
B: 6
A: 6
B: 7
A: 7
```

A thread will occasionally start another thread to perform a lengthy calculation, download a large file, or perform some other time-consuming activity. After finishing its other tasks, the thread that started the *worker thread* is ready to process the results of the worker thread and waits for the worker thread to finish and die.

It's possible to wait for the worker thread to die by using a while loop that repeatedly calls Thread's `isAlive()` method on the worker thread's `Thread` object and sleeps for a certain length of time when this method returns true. However, Listing 7-13 demonstrates a more clever alternative: the `join()` method.

Listing 7-13. Joining the Default Main Thread with a Background Thread

```
public class JoinDemo {  
    public static void main(String[] args) {  
        Runnable r = new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("Worker thread is simulating " +  
                    "work by sleeping for 5  
                    seconds.");  
                try {  
                    Thread.sleep(5000);  
                } catch (InterruptedException ie) {  
                }  
                System.out.println("Worker thread is dying");  
            }  
        };  
        Thread thd = new Thread(r);  
        thd.start();  
        System.out.println("Default main thread is doing work.");  
        try {  
            Thread.sleep(2000);  
        } catch (InterruptedException ie) {  
        }  
        System.out.println("Default main thread has finished its work.");  
        System.out.println("Default main thread is waiting for worker thread " +  
            "to die.");  
    }  
}
```

```

try {
    thd.join();
} catch (InterruptedException ie) {
}
System.out.println("Main thread is dying");
}
}

```

Listing 7-13 demonstrates the default main thread starting a worker thread, performing some work, and then waiting for the worker thread to die by calling `join()` via the worker thread's `thd` object. When you run this application, you will discover output similar to the following (message order might differ somewhat):

```

Default main thread is doing work.
Worker thread is simulating work by sleeping for 5 seconds.
Default main thread has finished its work.
Default main thread is waiting for worker thread to die.
Worker thread is dying
Main thread is dying

```

Inside the `Runnable`'s implementation you should take care of proper exception handling. Only in case you don't want that for improved readability reasons, you can also install an exception handler via the method `setDefaultUncaughtExceptionHandler()` of class `Thread`.

Synchronization

Throughout its execution, each thread is isolated from other threads because it has been given its own method-call stack. However, threads can still interfere with each other when they access and manipulate shared data. This interference can corrupt the shared data, and this corruption can cause an application to fail.

For example, consider a checking account in which a Husband and Wife have joint access. Suppose that the Husband and Wife decide to empty this account at the same time without knowing that the other is doing the same thing. Listing 7-14 demonstrates this scenario.

Listing 7-14. A Problematic Checking Account

```

public class CheckingAccount {
    private int balance;

    public CheckingAccount(int initialBalance) {
        balance = initialBalance;
    }

    public boolean withdraw(int amount) {
        if (amount <= balance) {
            try {
                // This is important to show the point; here
                // the threads interfere
                Thread.sleep((int) (Math.random() * 200));
            } catch (InterruptedException ie) {
            }
            balance -= amount;
            return true;
        }
        return false;
    }

    public static void main(String[] args) {
        final CheckingAccount ca = new CheckingAccount(100);
        Runnable r = new Runnable() {
            public void run() {
                String name = Thread.currentThread().getName();
                for (int i = 0; i < 10; i++)
                    System.out.println (name + " withdraws $10: " +
                        ca.withdraw(10));
            }
        };
        Thread thdHusband = new Thread(r);
        thdHusband.setName("Husband");
        Thread thdWife = new Thread(r);
        thdWife.setName("Wife");
    }
}

```

```
    thdHusband.start();
    thdWife.start();
}
}
```

This application lets more money be withdrawn than is available in the account. For example, the following output reveals \$110 being withdrawn when only \$100 is available:

```
Wife withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Wife withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: false
Wife withdraws $10: true
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
```

The reason why more money is withdrawn than is available for withdrawal is that a race condition exists between the Husband and Wife threads.

Note A *race condition* is a scenario in which multiple threads are accessing shared data and the final result of these accesses is dependent on the timing of how the threads are scheduled. Race conditions can lead to bugs that are hard to find and results that are unpredictable.

In this example, there is a race condition: between checking the amount for withdrawal to ensure that it is less than what appears in the balance and deducting the amount from the balance. The race condition exists because these actions are not *atomic* (indivisible) operations.

Consider the following scenario:

- The Husband thread executes `withdraw()`'s amount `<= balance` expression, which returns true. The scheduler then suspends the Husband thread and executes the Wife thread.
- The Wife thread executes `withdraw()`'s amount `<= balance` expression, which returns true.
- The Wife thread performs the withdrawal. The scheduler suspends the Wife thread and resumes the Husband thread.
- The Husband thread performs the withdrawal.

This problem can be corrected by *synchronizing* access to `withdraw()` so that only one thread at a time can execute inside this method. You can synchronize access to this method by adding reserved word `synchronized` to the method header prior to the method's return type, for example, `synchronized boolean withdraw(int amount)`.

When you run the modified `CheckingAccount` application, you will observe that the account withdrawables will not overflow any longer.

As we will demonstrate later, you can also synchronize access to a block of statements by specifying the following syntax:

```
synchronized(object) {
    /* statements */
}
```

According to this syntax, *object* is an arbitrary object reference.

Mutual Exclusion, Monitors, and Locks

Whether you are synchronizing access to a method or a block of statements, no thread can enter the synchronized region until a thread that's already executing inside that region leaves it. This property of synchronization is known as *mutual exclusion*.

Mutual exclusion is implemented in terms of monitors and locks. A *monitor* is a concurrency construct for controlling access to a *critical section*, a region of code that must execute atomically. It is identified at the source code level as a synchronized method or a synchronized block.

A *lock* is a token that a thread must acquire before a monitor allows that thread to execute inside a monitor's critical section. The token is released automatically when the thread exits the monitor, to give another thread an opportunity to acquire the token and enter the monitor.

Note A thread that has acquired a lock doesn't release this lock when it calls one of Thread's sleep() methods.

A thread entering a synchronized instance method acquires the lock associated with the object on which the method is called. A thread entering a synchronized class method acquires the lock associated with the class's java.lang.Class object. Finally, a thread entering a synchronized block acquires the lock associated with the block's controlling object.

Tip Thread declares a static boolean holdsLock(Object o) method that returns true when the calling thread holds the monitor lock on object o. You will find this method handy in assertion statements, such as assert Thread.holdsLock(o);.

Visibility

For performance reasons, each thread can have its own copy of a shared variable stored in a local *cache* (localized high-speed memory). Without synchronization, one thread's write to its copy will not be visible to other thread's copies. Ideally, when a thread updates a shared variable, this update should be made to the copy stored in main memory so that other threads can see these updates.

For this aim, and in case you don't need mutual exclusion, you can take advantage of Java's volatile reserved word, which supports visibility only, and which Listing 7-15 demonstrates.

Listing 7-15. The volatile Alternative to Synchronization

```

public class ThreadStopping {
    public static void main(String[] args) {
        class StoppableThread extends Thread {
            private volatile boolean stopped = false;

            @Override
            public void run() {
                while(!stopped)
                    System.out.println("running");
            }

            void stopThread() {
                stopped = true;
            }
        }
        StoppableThread thd = new StoppableThread();
        thd.start();
        try {
            Thread.sleep(1000); // sleep for 1 second
        } catch (InterruptedException ie) {
        }
        thd.stopThread();
    }
}

```

Listing 7-15 declares `stopped` to be `volatile`. Threads that access this field will always access a single shared copy (not cached copies on multiprocessor/multicore machines).

When a field is declared `volatile`, it cannot also be declared `final`. If you're depending on the *semantics* (meaning) of volatility, you still get those from a `final` field. For more information, check out Brian Goetz's "Java theory and practice: Fixing the Java Memory Model, Part 2" article (www.ibm.com/developerworks/library/j-jtp03304/).

Caution Use `volatile` only in a thread communication context. Also, you can only use this reserved word in the context of field declarations. Although you can declare `double` and `long` fields `volatile`, you should avoid doing so on 32-bit virtual machines because it takes two operations to access a `double` or `long` variable's value, and mutual exclusion (via synchronization) is required to access their values safely.

Waiting and Notification

The `java.lang.Object` class provides `wait()`, `notify()`, and `notifyAll()` methods to support a form of thread communication where a thread voluntarily waits for some *condition* (a prerequisite for continued execution) to arise, at which time another thread notifies the waiting thread that it can continue. `wait()` causes its calling thread to wait on an object's monitor, and `notify()` and `notifyAll()` wake up one or all threads waiting on the monitor.

Caution Because the `wait()`, `notify()`, and `notifyAll()` methods depend on a lock, they cannot be called from outside of a synchronized method or synchronized block. If you fail to heed this warning, you will encounter a thrown instance of the `java.lang.IllegalMonitorStateException` class. Also, a thread that has acquired a lock releases this lock when it calls one of `Object`'s `wait()` methods.

A classic example of thread communication involving conditions is the relationship between a producer thread and a consumer thread. The producer thread produces data items to be consumed by the consumer thread. Each produced data item is stored in a shared variable.

Imagine that the threads are running at different speeds. The producer might produce a new data item and record it in the shared variable before the consumer retrieves the previous data item for processing. Also, the consumer might retrieve the contents of the shared variable before a new data item is produced.

To overcome those problems, the producer thread must wait until it is notified that the previously produced data item has been consumed, and the consumer thread must

wait until it is notified that a new data item has been produced. Listing 7-16 shows you how to accomplish this task via `wait()` and `notify()`.

Listing 7-16. The Producer-Consumer Relationship, Version 1

```
public class PC {  
    public static void main(String[] args) {  
        Shared s = new Shared();  
        new Producer(s).start();  
        new Consumer(s).start();  
    }  
}  
  
public class Shared {  
    private char c = '\u0000';  
    private boolean writeable = true;  
  
    synchronized public void setSharedChar(char c) {  
        while (!writeable)  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        this.c = c;  
        writeable = false;  
        notify();  
    }  
  
    synchronized public char getSharedChar() {  
        while (writeable)  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        writeable = true;  
        notify();  
        return c;  
    }  
}
```

CHAPTER 7 EXPLORING THE BASIC APIs, PART 1

```
public class Producer extends Thread {  
    private Shared s;  
  
    public Producer(Shared s) {  
        this.s = s;  
    }  
  
    @Override  
    public void run() {  
        for (char ch = 'A'; ch <= 'Z'; ch++) {  
            s.setSharedChar(ch);  
            System.out.println(ch + " produced by producer.");  
        }  
    }  
}  
  
public class Consumer extends Thread {  
    private Shared s;  
  
    public Consumer(Shared s) {  
        this.s = s;  
    }  
  
    @Override  
    public void run() {  
        char ch;  
        do {  
            ch = s.getSharedChar();  
            System.out.println(ch + " consumed by consumer.");  
        } while (ch != 'Z');  
    }  
}
```

This application creates a Shared object and two threads that get a copy of the object's reference. The producer calls the object's setSharedChar() method to save each of 26 uppercase letters; the consumer calls the object's getSharedChar() method to acquire each letter.

The `writeable` instance field tracks two conditions: the producer waiting on the consumer to consume a data item and the consumer waiting on the producer to produce a new data item. It helps coordinate execution of the producer and consumer. The following scenario, where the consumer executes first, illustrates this coordination:

1. The consumer executes `s.getSharedChar()` to retrieve a letter.
2. Inside of that synchronized method, the consumer calls `wait()` because `writeable` contains true. The consumer now waits until it receives notification from the producer.
3. The producer eventually executes `s.setSharedChar(ch);`.
4. When the producer enters that synchronized method (which is possible because the consumer released the lock inside of the `wait()` method prior to waiting), the producer discovers `writeable`'s value to be true and doesn't call `wait()`.
5. The producer saves the character, sets `writeable` to false (which will cause the producer to wait on the next `setSharedChar()` call when the consumer has not consumed the character by that time), and calls `notify()` to awaken the consumer (assuming the consumer is waiting).
6. The producer exits `setSharedChar(char c)`.
7. The consumer wakes up (and reacquires the lock), sets `writeable` to true (which will cause the consumer to wait on the next `getSharedChar()` call when the producer has not produced a character by that time), notifies the producer to awaken that thread (assuming the producer is waiting), and returns the shared character.

Although the synchronization works correctly, you might observe output (on some platforms) that shows multiple producing messages before multiple consuming messages. For example, you might see A produced by producer., followed by B produced by producer., followed by A consumed by consumer., at the beginning of the application's output.

This strange output order is caused by the call to `setSharedChar()` followed by its companion `System.out.println()` method call not being atomic, and by the call to `getSharedChar()` followed by its companion `System.out.println()` method call not

being atomic. The output order can be corrected by wrapping each of these method call pairs in a synchronized block that synchronizes on the Shared object referenced by s. Listing 7-17 presents this enhancement.

Listing 7-17. The Producer-Consumer Relationship, Version 2

```
public class PC {  
    public static void main(String[] args) {  
        Shared s = new Shared();  
        new Producer(s).start();  
        new Consumer(s).start();  
    }  
}  
  
public class Shared {  
    private char c = '\u0000';  
    private boolean writeable = true;  
  
    public synchronized void setSharedChar(char c) {  
        while (!writeable)  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        this.c = c;  
        writeable = false;  
        notify();  
    }  
  
    public synchronized char getSharedChar() {  
        while (writeable)  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        writeable = true;  
        notify();  
        return c;  
    }  
}
```

```
public class Producer extends Thread {  
    private Shared s;  
  
    public Producer(Shared s){  
        this.s = s;  
    }  
  
    @Override  
    public void run() {  
        for (char ch = 'A'; ch <= 'Z'; ch++) {  
            synchronized(s) {  
                s.setSharedChar(ch);  
                System.out.println(ch + " produced by producer.");  
            }  
        }  
    }  
  
    public class Consumer extends Thread {  
        private Shared s;  
  
        public Consumer(Shared s) {  
            this.s = s;  
        }  
  
        @Override  
        public void run() {  
            char ch;  
            do {  
                synchronized(s) {  
                    ch = s.getSharedChar();  
                    System.out.println(ch + " consumed by consumer.");  
                }  
            } while (ch != 'Z');  
        }  
    }  
}
```

Compile Listing 7-17 (`javac PC.java`) and run this application (`java PC`). Its output should always appear in the same alternating order as shown next (only the first few lines are shown for brevity):

- A produced by producer.
- A consumed by consumer.
- B produced by producer.
- B consumed by consumer.
- C produced by producer.
- C consumed by consumer.
- D produced by producer.
- D consumed by consumer.

Caution Never call `wait()` outside of a loop. The loop tests the condition (`!writeable` or `writeable` in the previous example) before and after the `wait()` call. Testing the condition before calling `wait()` ensures *liveness*. If this test was not present, and if the condition held and `notify()` had been called prior to `wait()` being called, it is unlikely that the waiting thread would ever wake up. Retesting the condition after calling `wait()` ensures *safety*. If retesting didn't occur, and if the condition didn't hold after the thread had awakened from the `wait()` call (perhaps another thread called `notify()` accidentally when the condition didn't hold), the thread would proceed to destroy the lock's protected invariants.

Deadlock

Too much synchronization can be problematic. If you are not careful, you might encounter a situation where locks are acquired by multiple threads, neither thread holds its own lock but holds the lock needed by some other thread, and neither thread can enter and later exit its critical section to release its held lock because some other thread holds the lock to that critical section. Listing 7-18's atypical example demonstrates this scenario, which is known as *deadlock*.

Listing 7-18. A Pathological Case of Deadlock

```
public class DeadlockDemo {  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void instanceMethod1() {  
        synchronized(lock1) {  
            synchronized(lock2) {  
                System.out.println("first thread in instanceMethod1");  
                // critical section guarded first by  
                // lock1 and then by lock2  
            }  
        }  
    }  
  
    public void instanceMethod2() {  
        synchronized(lock2) {  
            synchronized(lock1) {  
                System.out.println("second thread in instanceMethod2");  
                // critical section guarded first by  
                // lock2 and then by lock1  
            }  
        }  
    }  
  
    public static void main(String[] args) {  
        final DeadlockDemo dld = new DeadlockDemo();  
        Runnable r1 = new Runnable() {  
            @Override  
            public void run() {  
                while(true) {  
                    dld.instanceMethod1();  
                    try {  
                        Thread.sleep(50);  
                    } catch (InterruptedException ie){  
                    }  
                }  
            }  
        };  
        Thread t1 = new Thread(r1);  
        t1.start();  
        Thread t2 = new Thread(r1);  
        t2.start();  
    }  
}
```

```

        }
    }
};

Thread thdA = new Thread(r1);
Runnable r2 = new Runnable() {
    @Override
    public void run() {
        while(true) {
            dld.instanceMethod2();
            try {
                Thread.sleep(50);
            } catch (InterruptedException ie) {
            }
        }
    }
};

Thread thdB = new Thread(r2);
thdA.start();
thdB.start();
}
}

```

Listing 7-18's thread A and thread B call `instanceMethod1()` and `instanceMethod2()`, respectively, at different times. Consider the following execution sequence:

1. Thread A calls `instanceMethod1()`, obtains the lock assigned to the `lock1`-referenced object, and enters its outer critical section (but has not yet acquired the lock assigned to the `lock2`-referenced object).
2. Thread B calls `instanceMethod2()`, obtains the lock assigned to the `lock2`-referenced object, and enters its outer critical section (but has not yet acquired the lock assigned to the `lock1`-referenced object).

3. Thread A attempts to acquire the lock associated with `lock2`.
The virtual machine forces the thread to wait outside of the inner critical section because thread B holds that lock.
4. Thread B attempts to acquire the lock associated with `lock1`. The virtual machine forces the thread to wait outside of the inner critical section because thread A holds that lock.
5. Neither thread can proceed because the other thread holds the needed lock. You have a deadlock situation and the program (at least in the context of the two threads) freezes up.

Although the previous example clearly identifies a deadlock state, it's often not that easy to detect deadlock. For example, your code might contain the following circular relationship among various classes (in several source files):

- Class A's synchronized method calls class B's synchronized method.
- Class B's synchronized method calls class C's synchronized method.
- Class C's synchronized method calls class A's synchronized method.

If thread A calls class A's synchronized method and thread B calls class C's synchronized method, thread B will block when it attempts to call class A's synchronized method and thread A is still inside of that method. Thread A will continue to execute until it calls class C's synchronized method, and then block. Deadlock is the result.

Note Neither the Java language nor the virtual machine provides a way to prevent deadlock, and so the burden falls on you. The simplest way to prevent deadlock from happening is to avoid having either a synchronized method or a synchronized block call another synchronized method/block. Although this advice prevents deadlock from happening, it is impractical because one of your synchronized methods/blocks might need to call a synchronized method in a Java API, and the advice is overkill because the synchronized method/block being called might not call any other synchronized method/block, so deadlock would not occur.

Thread-Local Variables

You will sometimes want to associate per-thread data (such as a user ID) with a thread. Although you can accomplish this task with a local variable, you can only do so while the local variable exists. You could use an instance field to keep this data around longer, but then you would have to deal with synchronization. Thankfully, Java supplies `ThreadLocal` as a simple (and very handy) alternative.

Each instance of the `ThreadLocal` class describes a *thread-local variable*, which is a variable that provides a separate storage slot to each thread that accesses the variable. You can think of a thread-local variable as a multislotted variable in which each thread can store a different value in the same variable. Each thread sees only its value and is unaware of other threads having their own values in this variable.

`ThreadLocal` is generically declared as `ThreadLocal<T>`, where `T` identifies the type of value that is stored in the variable. This class declares the following constructor and methods:

- `ThreadLocal()` creates a new thread-local variable.
- `T get()` returns the value in the calling thread's storage slot. If an entry doesn't exist when the thread calls this method, `get()` calls `initialValue()`.
- `T initialValue()` creates the calling thread's storage slot and stores an initial (default) value in this slot. The initial value defaults to null. You must subclass `ThreadLocal` and override this protected method to provide a more suitable initial value.
- `void remove()` removes the calling thread's storage slot. If this method is followed by `get()` with no intervening `set()`, `get()` calls `initialValue()`.
- `void set(T value)` sets the value of the calling thread's storage slot to `value`.

Listing 7-19 shows how to use `ThreadLocal` to associate different user IDs with two threads.

Listing 7-19. Different User IDs for Different Threads

```

public class ThreadLocalDemo {
    private static volatile ThreadLocal<String> userID =
        new ThreadLocal<String>();

    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                if (name.equals("A"))
                    userID.set("foxtrot");
                else
                    userID.set("charlie");
                System.out.println(name + " " + userID.get());
            }
        };
        Thread thdA = new Thread(r);
        thdA.setName("A");
        Thread thdB = new Thread(r);
        thdB.setName("B");
        thdA.start();
        thdB.start();
    }
}

```

After instantiating `ThreadLocal` and assigning the reference to a `volatile` class field named `userID` (the field is `volatile` because it is accessed by different threads, which might execute on a multiprocessor/multicore machine), the default main thread creates two more threads that store different `String` objects in `userID` and output their objects.

When you run this application, you will observe the following output (possibly not in this order):

```
A foxtrot
B charlie
```

Values stored in thread-local variables are not related. When a new thread is created, it gets a new storage slot containing `initialValue()`'s value. Perhaps you would prefer to pass a value from a *parent thread*, a thread that creates another thread, to a *child thread*, the created thread. You accomplish this task with `InheritableThreadLocal`.

Note For more insight into `ThreadLocal` and how it is implemented, check out Patson Luk's "A Painless Introduction to Java's ThreadLocal Storage" blog post (<http://java.dzone.com/articles/painless-introduction-javas-threadlocal-storage>).

EXERCISES

The following exercises are designed to test your understanding of Chapter 7's content:

1. What constants does `Math` declare?
2. Why is `Math.abs(Integer.MIN_VALUE)` equal to `Integer.MIN_VALUE`?
3. What does `Math's random()` method accomplish?
4. Identify the five special values that can arise during floating-point calculations.
5. What is `BigDecimal` and why might you use this class?
6. Which `RoundingMode` constant describes the form of rounding commonly taught at school?
7. What is `BigInteger`?
8. What is a primitive type wrapper class?
9. Identify Java's primitive type wrapper classes.
10. Why does Java provide primitive type wrapper classes?
11. True or false: `Byte` is the smallest of the primitive type wrapper classes.
12. Why should you use `Character` class methods instead of expressions such as `ch >= '0' && ch <= '9'` to determine whether or not a character is a digit, a letter, and so on?

13. How do you determine whether or not double variable d contains +infinity or -infinity?
14. Identify the class that is the superclass of Byte, Character, and the other primitive type wrapper classes.
15. True or false: A string literal is a String object.
16. What is the purpose of String's intern() method?
17. How do String and StringBuffer differ?
18. How do StringBuffer and StringBuilder differ?
19. What System method do you invoke to copy an array to another array?
20. What System method do you invoke to obtain the current time in milliseconds?
21. Define thread.
22. What is the purpose of the Runnable interface?
23. What is the purpose of the Thread class?
24. True or false: A Thread object associates with multiple threads.
25. Define race condition.
26. What is synchronization?
27. How is synchronization implemented?
28. How does synchronization work?
29. True or false: Variables of type long or double are not atomic on 32-bit virtual machines.
30. What is the purpose of reserved word volatile?
31. True or false: Object's wait() methods can be called from outside of a synchronized method or block.
32. Define deadlock.
33. What is the purpose of the ThreadLocal class?
34. A *prime number* is a positive integer greater than 1 that is evenly divisible only by 1 and itself. Create a PrimeNumberTest application that determines if its solitary integer argument is prime or not prime, and outputs a suitable

message. For example, `java PrimeNumberTest 289` should output the message `289 is not prime`. A simple way to check for primality is to loop from 2 through the square root of the integer argument, and use the remainder operator in the loop to determine if the argument is divided evenly by the loop index. For example, because `6 % 2` yields a remainder of 0 (2 divides evenly into 6), integer 6 is not a prime number.

35. Create a `MultiPrint` application that takes two arguments: text and an integer value that represents a count. This application should print count copies of the text, one copy per line.
36. Rewrite the following inefficient loop to use `StringBuffer`. The resulting loop should minimize object creation:

```
String[] imageNames = new String[NUM_IMAGES];
for (int i = 0; i < imageNames.length; i++)
    imageNames[i] = new String("image" + i + ".png");
```

37. Create a `DigitsToWords` application that accepts a single integer-based command-line argument. This application converts this argument to an `int` value (via `Integer.parseInt(args[0])`) and then passes the result to a `String convertDigitsToWords(int integer)` class method that returns a string containing a textual representation of that number. For example, 1 converts to one, 16 converts to sixteen, 69 converts to sixty-nine, 123 converts to one hundred and twenty-three, and 2938 converts to two thousand nine hundred and thirty-eight. Throw `java.lang.IllegalArgumentException` when the value passed to `integer` is less than 0 or greater than 9999. Use the `StringBuffer` class to serve as a repository for the generated text. Usage example: `java DigitsToWords 2938`.
38. Create an `EVDump` application that dumps all environment variables (not system properties) to the standard output.
39. Modify Listing 7-11's `CountingThreads` application by marking the two started threads as daemon threads. What happens when you run the resulting application?

40. Modify Listing 7-11's CountingThreads application by adding logic to stop both counting threads when the user presses the Return/Enter key. The default main thread of the new StopCountingThreads application should call `System.in.read()` before terminating, and assign `true` to a variable named `stopped` after this method call returns. At each loop iteration start, each counting thread should test this variable to see if it contains `true`, and only continue the loop when the variable contains `false`.
-

Summary

The standard class library offers many basic APIs via its `java.lang` and other packages. For example, the `Math` class supplements the basic math operations (+, -, *, /, and %) with advanced operations (such as trigonometry).

The abstract `java.lang.Number` class is the superclass of those classes representing numeric values that are convertible to the byte integer, double-precision floating-point, floating-point, integer, long integer, and short integer primitive types.

Money must never be represented by floating-point and double-precision floating-point variables because not all monetary values can be represented exactly. In contrast, `Number`'s `BigDecimal` subclass lets you accurately represent and manipulate these values.

`BigDecimal` relies on `Number`'s `BigInteger` subclass for representing its unscaled value. A `BigInteger` instance describes an integer value that can be of arbitrary length (subject to the limits of the virtual machine's memory).

`Number`'s `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short` subclasses are known as *primitive type wrapper classes* or *value classes* because their instances wrap themselves around values of primitive types.

Java provides these eight primitive type wrapper classes so that primitive-type values can be stored in collections, such as lists, sets, and maps. Furthermore, these classes provide a good place to associate useful constants and class methods with the primitive types.

`String` represents a string as a sequence of characters. Because `String` instances are immutable, Java provides `StringBuffer` and `StringBuilder` for building strings more efficiently. The former class is used in multithreaded contexts; the latter (and more performant) class is for single-threaded use.

Applications execute via threads that serve as independent paths of execution through an application's code. The virtual machine gives each thread its own method-call stack to prevent threads from interfering with each other.

Java supports threads via its Threads API. This API largely consists of one interface (`Runnable`) and four classes (`Thread`, `ThreadGroup`, `ThreadLocal`, and `InheritableThreadLocal`) in the `java.lang` package.

Throughout its execution, each thread is isolated from other threads because it has been given its own method-call stack. However, threads can still interfere with each other when they access and manipulate shared data. This interference can corrupt the shared data, causing an application to fail.

Corruption can be avoided by using synchronization so that only one thread at a time can execute inside a critical section, a region of code that must execute atomically. It is identified at the source code level as a synchronized method or a synchronized block.

You synchronize access at the method level by adding reserved word `synchronized` to the method header prior to the method's return type. You can also synchronize access to a block of statements by specifying a synchronized block via the following syntax:
`synchronized(object) { /* statements */ }.`

Synchronization supports mutual exclusion and is implemented in terms of monitors and locks. A monitor controls access to a critical section and a lock must be acquired by a thread before the monitor will allow the thread to execute inside the monitor's critical section.

Synchronization also supports visibility in which a thread always sees the main memory value and not a cached value of a shared variable. There exists an alternative to synchronization when only visibility is required. This alternative is reserved word `volatile`.

`Object`'s `wait()`, `notify()`, and `notifyAll()` methods support a form of thread communication where a thread voluntarily waits for some condition to arise, at which time another thread notifies the waiting thread that it can continue. `wait()` causes its calling thread to wait on an object's monitor, and `notify()` and `notifyAll()` wake up one or all threads waiting on the monitor.

Too much synchronization can be problematic. If you are not careful, you might encounter a situation where locks are acquired by multiple threads, neither thread holds its own lock but holds the lock needed by some other thread, and neither thread can

enter and later exit its critical section to release its held lock because some other thread holds the lock to that critical section. This scenario is known as deadlock.

You will sometimes want to associate per-thread data with a thread. Although you can accomplish this task with a local variable, you can only do so while the local variable exists. You could use an instance field to keep this data around longer, but then you would have to deal with synchronization. Java supplies the `ThreadLocal` class as a simple (and very handy) alternative.

Each `ThreadLocal` instance describes a thread-local variable that provides a separate storage slot to each thread that accesses the variable. Think of a thread-local variable as a multislots variable in which each thread can store a different value in the same variable. Each thread sees only its value and is unaware of other threads having their own values in this variable.

Values stored in thread-local variables are not related. When a new thread is created, it gets a new storage slot containing `initialValue()`'s value. However, you can pass a value from a parent thread to a child thread by working with the `InheritableThreadLocal` class.

Chapter 8 continues to explore the basic APIs by focusing on Random, References, Reflection, StringTokenizer, and Timer and TimerTask.

CHAPTER 8

Exploring the Basic APIs, Part 2

There are more basic APIs in the `java.lang` package and also in `java.lang.ref`, `java.lang.reflect`, and `java.util` to consider for your Android apps. For example, you can add timers to your games.

Exploring Random

In Chapter 7, we formally introduced you to the `java.lang.Math` class's `random()` method. If you were to investigate this method's source code from the perspective of Java 8 or later (at least up to Java 12), you would encounter the following implementation:

```
public static double random() {  
    return  
    RandomNumberGeneratorHolder.randomNumberGenerator.nextDouble();  
}  
...  
// class Math  
private static final class RandomNumberGeneratorHolder {  
    static final Random randomNumberGenerator = new Random();  
}
```

This code excerpt shows you that `Math`'s `random()` method is implemented in terms of a class named `Random`, which is located in the `java.util` package. `Random` instances generate sequences of random numbers and are known as *random number generators*.

Note These numbers are not truly random because they are generated from a mathematical algorithm. As a result, they are often referred to as *pseudorandom numbers*. However, it is often convenient to drop the “pseudo” prefix and refer to them as random numbers. Also, delaying object creation (e.g., new Random()) until the first time the object is needed is known as *lazy initialization*.

Random generates its sequence of random numbers by starting with a special 48-bit value that is known as a *seed*. This value is subsequently modified by a mathematical algorithm, which is known as a *linear congruential generator*.

Note Check out Wikipedia’s “linear congruential generator” entry (http://en.wikipedia.org/wiki/Linear_congruential_generator) to learn about this algorithm for generating random numbers.

Random declares a pair of constructors:

- Random() creates a new random number generator. This constructor sets the seed of the random number generator to a value that is very likely to be distinct from any other call to this constructor.
- Random(long seed) creates a new random number generator using its seed argument. This argument is the initial value of the internal state of the random number generator, which is maintained by the protected int next(int bits) method.

Because Random() doesn’t take a seed argument, the resulting random number generator always generates a different sequence of random numbers. This explains why Math.random() generates a different sequence each time an application starts running.

Tip Random(long seed) gives you the opportunity to reuse the same seed value, allowing the same sequence of random numbers to be generated. You will find this capability useful when debugging a faulty application that involves random numbers.

`Random(long seed)` calls the `void setSeed(long seed)` method to set the seed to the specified value. If you call `setSeed()` after instantiating `Random`, the random number generator is reset to the state that it was in immediately after calling `Random(long seed)`.

The previous code excerpt demonstrates `Random`'s `double nextDouble()` method, which returns the next pseudorandom, uniformly distributed double-precision floating-point value between 0.0 and 1.0 in this random number generator's sequence.

`Random` also declares the following methods for returning other kinds of values:

- `boolean nextBoolean()` returns the next pseudorandom, uniformly distributed Boolean value in this random number generator's sequence. Values true and false are generated with (approximately) equal probability.
- `void nextBytes(byte[] bytes)` generates pseudorandom byte integer values and stores them in the `bytes` array. The number of generated bytes is equal to the length of the `bytes` array.
- `float nextFloat()` returns the next pseudorandom, uniformly distributed floating-point value between 0.0 and 1.0 in this random number generator's sequence.
- `double nextGaussian()` returns the next pseudorandom, *Gaussian* (“normally”) distributed double-precision floating-point value with mean 0.0 and standard deviation 1.0 in this random number generator's sequence.
- `int nextInt()` returns the next pseudorandom, uniformly distributed integer value in this random number generator's sequence. All 4,294,967,296 possible integer values are generated with (approximately) equal probability.
- `int nextInt(int n)` returns a pseudorandom, uniformly distributed integer value between 0 (inclusive) and the specified value (exclusive) drawn from this random number generator's sequence. All `n` possible integer values are generated with (approximately) equal probability.

- long `nextLong()` returns the next pseudorandom, uniformly distributed long integer value in this random number generator's sequence. Because `Random` uses a seed with only 48 bits, this method will not return all possible 64-bit long integer values.

The `java.util.Collections` class declares a pair of `shuffle()` methods for shuffling the contents of a list. In contrast, the `java.util.Arrays` class doesn't declare a `shuffle()` method for shuffling the contents of an array. Listing 8-1 addresses this omission.

Listing 8-1. Shuffling an Array of Integers

```
import java.util.Random;

public class Shuffle {
    public static void main(String[] args) {
        Random r = new Random();
        int[] array = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        for (int i = 0; i < array.length; i++) {
            int n = r.nextInt(array.length);
            // swap array[i] with array[n]
            int temp = array[i];
            array[i] = array[n];
            array[n] = temp;
        }
        for (int i = 0; i < array.length; i++)
            System.out.print(array[i] + " ");
        System.out.println();
    }
}
```

Listing 8-1 presents a simple recipe for shuffling an array of integers—this recipe could be generalized. For each array entry from the start of the array to the end of the array, this entry is swapped with another entry whose index is chosen by `int nextInt(int n)`.

When you run this application, you will observe a shuffled sequence of integers that is similar to the following sequence that we observed:

9 0 5 6 2 3 8 4 1 7

Exploring Reflection

Chapter 4 presented two forms of runtime type identification (RTTI). Java’s Reflection API offers a third RTTI form in which applications can dynamically load and learn about loaded classes and other reference types. The API also lets applications instantiate classes, call methods, access fields, and perform other tasks reflectively. This form of RTTI is known as *reflection*.

Caution Reflection should not be used indiscriminately. Application performance suffers because it takes longer to perform operations with reflection than without reflection. Also, reflection-oriented code can be harder to read and might jeopardize the original application architecture, and the absence of compile-time type checking can result in runtime failures.

Chapter 6 presented a StubFinder application that used part of the Reflection API to load a class and identify all of the loaded class’s public methods that are annotated with @Stub annotations. This tool is one example where using reflection is beneficial. Another example is the *class browser*, a tool that enumerates the members of a class.

The Class Entry Point

The `java.lang` package’s `Class` class is the entry point into the Reflection API, whose types are stored mainly in the `java.lang.reflect` package. `Class` is generically declared as `Class<T>`, where `T` identifies the class, interface, enum, or annotation type that’s being modeled by the `Class` object. `T` can be replaced by `?` (as in `Class<?>`) when the type being modeled is unknown.

The API documentation at <https://docs.oracle.com/javase/8/docs/api> lists and describes all of `Class`’s methods.

Obtaining a Class Object

The `forName()` method of class `Class` reveals one way to obtain a `Class` object. This method loads, links, and initializes a class or interface that isn’t in memory and returns a `Class` object representing the class or interface. Listing 8-2 demonstrates `forName()` and some additional methods of use for this class.

Listing 8-2. Using Reflection to Investigate a Type

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

public class Investigator {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage: java Investigator classname");
            return;
        }
        Try {
            investigateClass(Class.forName(args[0]), 0);
        } catch (ClassNotFoundException cnfe) {
            System.err.println("could not locate " + args[0]);
        }
    }

    public static void investigateClass(Class<?> clazz, int indentLevel) {
        indent(indentLevel * 3);
        System.out.print(Modifier.toString(clazz.getModifiers()) + " ");
        if (clazz.isEnum())
            System.out.println("enum " + clazz.getName());
        else if (clazz.isInterface())
            if (clazz.isAnnotation())
                System.out.print("@");
            System.out.println(clazz.getName());
        } else
            System.out.println(clazz);
        indent(indentLevel * 3);
        System.out.println("{");
        Field[] fields = clazz.getDeclaredFields();
        for (int i = 0; i < fields.length; i++) {
            indent(indentLevel * 3);
            System.out.println("    " + fields[i]);
        }
    }
}
```

```
}

Constructor[] constructors = clazz.getDeclaredConstructors();
if (constructors.length != 0 && fields.length != 0)
    System.out.println();
for (int i = 0; i < constructors.length; i++) {
    indent(indentLevel * 3);
    System.out.println("    "+constructors[i]);
}
Method[] methods = clazz.getDeclaredMethods();
if (methods.length != 0 &&
    (fields.length != 0 || constructors.length != 0))
    System.out.println();
for (int i = 0; i < methods.length; i++) {
    indent(indentLevel * 3);
    System.out.println("    "+methods[i]);
}
Method[] methodsAll = clazz.getMethods();
if (methodsAll.length != 0 &&
    (fields.length != 0 || constructors.length != 0 ||
     methods.length != 0))
    System.out.println();
if (methodsAll.length != 0) {
    indent(indentLevel * 3);
    System.out.println("    ALL PUBLIC METHODS");
    System.out.println();
}
for (int i = 0; i < methodsAll.length; i++){
    indent(indentLevel * 3);
    System.out.println("    "+methodsAll[i]);
}
Class<?>[] members = clazz.getDeclaredClasses();
if (members.length != 0 && (fields.length != 0 ||
    constructors.length != 0 || methods.length != 0 ||
    methodsAll.length != 0))
    System.out.println();
```

```

        for (int i = 0; i < members.length; i++) {
            if (clazz != members[i]) {
                investigateClass(members[i], indentLevel + 1);
                if (i != members.length - 1)
                    System.out.println();
            }
            indent(indentLevel * 3);
            System.out.println("}");
        }
    }

    static void indent(int numSpaces) {
        for (int i = 0; i < numSpaces; i++)
            System.out.print(' ');
    }
}

```

Listing 8-2 presents the source code to an investigation tool that uses reflection to obtain information about the command-line argument, which must be a Java reference type (e.g., a class). The investigator outputs the type and name information for a class's fields, constructors, methods, and nested types. It also outputs the members of interfaces, enums, and annotation types.

After verifying that one command-line argument has been passed to this application, `main()` calls `forName()` to return a `Class` object representing the class or interface identified by this argument.

`forName()` throws an instance of the checked `ClassNotFoundException` class when it cannot locate the class's class file (perhaps the class file was erased before executing the application). It also throws `LinkageError` when a class's class file is malformed and `ExceptionInInitializerError` when a class's static initialization fails.

Note `ExceptionInInitializerError` is often thrown as the result of a class initializer throwing an unchecked exception. For example, the class initializer in the following `FailedInitialization` class results in `ExceptionInInitializerError` because `someMethod()` throws `java.lang.NullPointerException`:

```
public class FailedInitialization {
    static {
        ... some code which for example throws a NullPointerException
    }
}
```

Assuming that `forName()` is successful, the returned object's reference is passed to `investigateClass()`, which introspects the type. (`investigateClass()` is recursive in that it invokes itself for every encountered nested type.)

The `investigateClass()` method invokes `Class`'s `getModifiers()`, `isEnum()`, `getName()`, `isInterface()`, `isAnnotation()`, `getDeclaredFields()`, `getDeclaredConstructors()`, `getDeclaredMethods()`, `getMethods()`, and `getDeclaredClasses()` methods to return different pieces of information about the loaded class or interface, which is subsequently output.

Much of the printing code focuses on making the output look nice as if it was a source code listing. The code manages indentation and only allows a newline character to be output to separate one section from another; a newline character isn't output unless content appears before and after the newline.

Compile Listing 8-2 (`javac Investigator.java`) and run this application, for example, with `java.lang.Byte` as the solitary command-line argument (`java Investigator java.lang.Byte`).

Another way to obtain a `Class` object is to call `Object`'s `getClass()` method on an object reference, for example, `Employee e = new Employee(); Class<? extends Employee> clazz = e.getClass();`. The `getClass()` method doesn't throw an exception because the class from which the object was created is already present in memory.

There is one more way to obtain a `Class` object, and that is to employ a *class literal*, which is an expression consisting of a class name, followed by a period separator that's followed by reserved word `class`. Examples of class literals include `Class<Employee> clazz = Employee.class;` and `Class<String> clazz = String.class.`

Perhaps you're wondering about how to choose between `forName()`, `getClass()`, and a class literal. To help you make your choice, the following list compares each competitor:

- `forName()` is very flexible in that you can dynamically specify any reference type by its package-qualified name. If the type isn't in memory, it's loaded, linked, and initialized. However, lack of compile-time type safety can lead to runtime failures.
- `getClass()` returns a `Class` object describing the type of its referenced object. When called on a superclass variable containing a subclass instance, a `Class` object representing the subclass type is returned. Because the class is in memory, type safety is assured.
- A class literal returns a `Class` object representing its specified class. Class literals are compact and the compiler enforces type safety by refusing to compile the source code when it cannot locate the literal's specified class.

Note You can use class literals with primitive types, including `void`. Examples include `int.class`, `double.class`, and `void.class`. The returned `Class` object represents the class identified by a primitive type wrapper class's `TYPE` field or `java.lang.Void.TYPE`. For example, each of `int.class == java.lang.Integer.TYPE` and `void.class == Void.TYPE` evaluates to true.

You can also use class literals with primitive type-based arrays. Examples include `int[].class` and `double[].class`. For these examples, the returned `Class` objects represent `Class<int[]>` and `Class<double[]>`.

Instantiating a Dynamically Loaded Class

One of the `Class`' API not yet further investigated is `newInstance()`, which is useful for instantiating a dynamically loaded class provided that the class has a noargument constructor. The following code fragment demonstrates this:

```
try {
    Class<?> clazz = Class.forName("codecs.AVI");
    Codec codec = (Codec) clazz.newInstance();
    VideoPlayer player = new VideoPlayer(codec);
    player.play("movie.avi");
```

```

} catch( ClassNotFoundException
        | IllegalAccessException
        | InstantiationException e) {
    e.printStackTrace(System.err);
}

```

This code fragment uses `Class.forName()` to attempt to load a hypothetical Audio Video Interleave (AVI) compressor/decompressor (codec), which is stored as `AVI.class` in the `codecs` package. If successful, the codec is instantiated via the `newInstance()` method. A hypothetical `VideoPlayer` class is instantiated and its instance is initialized to the codec, and the player is told to play the contents of `movie.avi`, which was encoded via this codec.

`forName()` throws `ClassNotFoundException` when it cannot find `AVI.class`. `newInstance()` throws `IllegalAccessException` when it cannot locate a noargument constructor in `AVI.class` and `InstantiationException` when instantiation fails (perhaps the class file describes an interface or an abstract class).

Constructor, Field, and Method

Instances of `java.lang.reflect.Constructor`, `Field`, and `Method` (same package) represent a class's constructors and a class's or an interface's fields and methods.

`Constructor` represents a constructor and is generically declared as `Constructor<T>` where `T` identifies the class in which the constructor represented by `Constructor` is declared. `Constructor` declares various methods you can learn about by looking at the API documentation.

Tip If you want to instantiate a class via a constructor that takes arguments, you cannot use `Class`'s `newInstance()` method. Instead, you must use `Constructor`'s `T newInstance(Object... initargs)` method to perform this task. Unlike `Class`'s `newInstance()` method, which bypasses the compile-time exception checking that would otherwise be performed by the compiler, `Constructor`'s `newInstance()` method avoids this problem by wrapping any exception thrown by the constructor in an instance of the `java.lang.reflect.InvocationTargetException` class.

`Field` represents a field and declares various methods which allow to investigate fields.

Method `get()` returns the value of any type of field. In contrast, the other `get*()` methods return the values of specific types of fields. These methods throw a `NullPointerException` instance when `object` is `null` and the field is an instance field, an `IllegalArgumentException` instance when `object` is not an instance of the class or interface declaring the underlying field (or not an instance of a subclass or interface implementor), and an `IllegalAccessException` instance when the underlying field cannot be accessed (perhaps the field is private).

[Listing 8-3](#) demonstrates `Field`'s `getInt(Object)` and `getDouble(Object)` methods along with their `void setInt(Object obj, int i)` and `void setDouble(Object obj, double d)` counterparts.

Listing 8-3. Reflectively Getting and Setting the Values of Instance and Class Fields

```
import java.lang.reflect.Field;

class X {
    public int i = 10;
    public static final double PI = 3.14;
}

public class FieldAccessDemo {
    public static void main(String[] args) {
        try {
            Class<?> clazz = Class.forName("X");
            X x = (X) clazz.newInstance();
            Field f = clazz.getField("i");
            System.out.println(f.getInt(x)); // Output: 10
            f.setInt(x, 20);
            System.out.println(f.getInt(x)); // Output: 20
            f = clazz.getField("PI");
            System.out.println(f.getDouble(null)); // Output: 3.14
            f.setDouble(x, 20);
            System.out.println(f.getDouble(null)); // Never executed
        } catch (Exception e) {
```

```

        System.err.println(e);
    }
}
}

```

FieldAccessDemo's main() method first attempts to load X and then tries to instantiate this class via newInstance(). If successful, the instance is assigned to reference variable x.

main() next invokes Class's Field getField(String name) method to return a Field instance that represents the public field identified by name, which happens to be i (in the first case) and PI (in the second case). This method throws java.lang.NoSuchFieldException when the named field doesn't exist.

Continuing, main() invokes Field's getInt() and setInt() methods (with an object reference) to get the instance field's initial value, change this value to another value, and get the new value. The initial and new values are output.

At this point, main() demonstrates class field access in a similar manner. However, it passes null to getInt() and setInt() because an object reference isn't required to access a class field. Because PI is declared final, the call to setInt() results in a thrown instance of the IllegalAccessException class.

Note For simplicity we've specified catch(Exception e) to avoid having to specify multiple catch blocks.

Method represents a method and declares various methods which we can use to learn about a method.

Listing 8-4 demonstrates Method's invoke(Object, Object...) method.

Listing 8-4. Reflectively Invoking Instance and Class Methods

```

import java.lang.reflect.Method;

class X {
    public void objectMethod(String arg) {
        System.out.println("Instance method: " + arg);
    }
}

```

```

public static void classMethod() {
    System.out.println("Class method");
}

}

public class MethodInvocationDemo {
    public static void main(String[] args) {
        try {
            Class<?> clazz = Class.forName("X");
            X x = (X) clazz.newInstance();
            Class[] argTypes = { String.class };
            Method method = clazz.getMethod("objectMethod", argTypes);
            Object[] data = { "Hello" };
            method.invoke(x, data); // Output: Instance method: Hello
            method = clazz.getMethod("classMethod", (Class<?>[]) null);
            method.invoke(null, (Object[]) null); // Output: Class method
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}

```

Listing 8-4 declares classes X and MethodInvocationDemo. MethodInvocationDemo's main() method first attempts to load X and then tries to instantiate this class via newInstance(). When successful, the instance is assigned to reference variable x.

main() next creates a one-element Class array that describes the types of objectMethod()'s parameter list. This array is used in the subsequent call to Class's Method getMethod(String name, Class<?>... parameterTypes) method to return a Method object for invoking a public method named objectMethod with this parameter list. This method throws java.lang.NoSuchMethodException when the named method doesn't exist.

Continuing, main() creates an Object array that specifies the data to be passed to the method's parameters; in this case, the array consists of a single String argument. It then reflectively invokes objectMethod() by passing this array along with the object reference stored in x to the invoke() method.

At this point, `main()` shows you how to reflectively invoke a class method. The `(Class<?>[])` and `(Object[])` casts are used to suppress warning messages that have to do with variable numbers of arguments and null references. Notice that the first argument passed to `invoke()` is null when invoking a class method.

Accessibility of Objects

The `java.lang.reflect.AccessibleObject` class is the superclass of `Constructor`, `Field`, and `Method`. This superclass provides annotation-related methods, and methods for reporting a constructor's, field's, or method's accessibility (is it private?) and making an inaccessible constructor, field, or method accessible. Please look at the API documentation for all `AccessibleObject`'s methods.

Package

Another class of the reflection API is `Package`, a class that provides information about a package (see Chapter 5 for an introduction to packages). This information includes version details about the implementation and specification of a Java package, the name of the package, and an indication of whether or not the package has been *sealed* (all classes that are part of the package are archived in the same JAR file).

We have created a `PackageInfo` application that demonstrates many of the `Package` class's methods. Listing 8-5 presents this application's source code.

Listing 8-5. Obtaining Information About a Package

```
public class PackageInfo {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.err.println("usage: java PackageInfo packageName
                [version]");
            return;
        }
        Package pkg = Package.getPackage(args[0]);
        if (pkg == null) {
            System.err.println(args[0] + " not found");
            return;
        }
    }
}
```

```

System.out.println("Name: " + pkg.getName());
System.out.println("Implementation title: " +
                   pkg.getImplementationTitle());
System.out.println("Implementation vendor: " +
                   pkg.getImplementationVendor());
System.out.println("Implementation version: " +
                   pkg.getImplementationVersion());
System.out.println("Specification title: " +
                   pkg.getSpecificationTitle());
System.out.println("Specification vendor: " +
                   pkg.getSpecificationVendor());
System.out.println("Specification version: " +
                   pkg.getSpecificationVersion());
System.out.println("Sealed: " + pkg.isSealed());
if (args.length > 1)
    System.out.println("Compatible with " + args[1] + ": " +
                       pkg.isCompatibleWith(args[1]));
}
}

```

After compiling Listing 8-5 (`javac PackageInfo.java`), specify at least a package name on the command line. For example, `java PackageInfo java.lang` returns the following output under Java 8:

```

Name: java.lang
Implementation title: Java Runtime Environment
Implementation vendor: Oracle Corporation
Implementation version: 1.8.0_101
Specification title: Java Platform API Specification
Specification vendor: Oracle Corporation
Specification version: 1.8
Sealed: false

```

`PackageInfo` also lets you determine if the package's specification is compatible with a specific version number. A package is compatible with its predecessors.

For example, `java PackageInfo java.lang 1.6` outputs `Compatible with 1.6: true`, whereas `java PackageInfo java.lang 1.8` outputs `Compatible with 1.8: false`.

You can also use `PackageInfo` with your own packages, which you learned to create in Chapter 5. For example, that chapter presented a logging package.

Copy `PackageInfo.class` into the directory containing the logging package directory (which contains the compiled class files), and execute the following command:

```
java PackageInfo logging
```

`PackageInfo` responds by displaying the following output:

```
logging not found
```

This error message is presented because `getPackage()` requires at least one class file to be loaded from the package before it returns a `Package` object describing that package.

The only way to eliminate the previous error message is to load a class from the package. Accomplish this task by merging the following code fragment into Listing 8-5:

```
if (args.length == 3) try {
    Class.forName(args[2]);
} catch (ClassNotFoundException cnfe) {
    System.err.println("cannot load " + args[2]);
    return;
}
```

This code fragment, which must precede `Package pkg = Package.getPackage(args[0]);`, loads the class file named by the revised `PackageInfo` application's third command-line argument.

Run the new `PackageInfo` application via `java PackageInfo logging 1.5 logging.File` and you will observe the following output, provided that `File.class` exists (you need to compile this package before specifying this command line)—this command line identifies `logging`'s `File` class as the class to load:

```
Name: logging
Implementation title: null
Implementation vendor: null
Implementation version: null
Specification title: null
Specification vendor: null
Specification version: null
Sealed: false
```

```
Exception in thread "main" java.lang.NumberFormatException: Empty version
string
```

```
  at java.lang.Package.isCompatibleWith(Unknown Source)
  at PackageInfo.main(PackageInfo.java:41)
```

It's not surprising to see all of these null values because no package information has been added to the logging package. Also, `NumberFormatException` is thrown from `isCompatibleWith()` because the logging package doesn't contain a specification version number in dotted form (it is null).

Perhaps the simplest way to place package information into the logging package is to create a `logging.jar` file in a similar manner to the example shown in Chapter 5. But first, you must create a small text file that contains the package information. You can choose any name for the file. Listing 8-6 reveals our choice of `manifest.mf`.

Listing 8-6. `manifest.mf` Containing the Package Information

```
Implementation-Title: Logging Implementation
Implementation-Vendor: Jeff Friesen
Implementation-Version: 1.0a
Specification-Title: Logging Specification
Specification-Vendor: Jeff "JavaJeff" Friesen
Specification-Version: 1.0
Sealed: true
```

Note Make sure to press the Return/Enter key at the end of the final line (`Sealed: true`). Otherwise, you will probably observe `Sealed: false` in the output because this entry will not be stored in the logging package by the JDK's `jar` tool; `jar` is a bit quirky.

Execute the following command line to create a JAR file that includes logging and its files, and whose *manifest*, a special file named `MANIFEST.MF` that stores information about the contents of a JAR file, contains the contents of Listing 8-6:

```
jar cfm logging.jar manifest.mf logging
```

Alternatively, specify one of the following slightly longer command lines, which are equivalent to the former command line:

```
jar cfm logging.jar manifest.mf logging\*.class  
jar cfm logging.jar manifest.mf logging/*.class
```

Either command line creates a JAR file named `logging.jar` (via the `c` [create] and `f` [file] options). It also merges the contents of `manifest.mf` (via the `m` [manifest] option) into `MANIFEST.MF`, which is stored in the package's/JAR's file `META-INF` directory.

Note To learn more about a JAR file's manifest, read the “JAR Manifest” section of the JDK documentation's “JAR File Specification” page (http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html#JAR_Manifest).

Assuming that the `jar` tool presents no error messages, execute the following Windows-oriented command line (or a command line suitable for your platform) to run `PackageInfo` and extract the package information from the `logging` package:

```
java -cp logging.jar; . PackageInfo logging 1.0 logging.File
```

The `-cp` command-line option lets you specify the classpath, which consists of `logging.jar` and the current directory (represented by the dot `[.]` character). Fail to specify the dot and `java` outputs an error message complaining that it cannot locate `PackageInfo.class`.

This time, you should see the following output:

```
Name: logging  
Implementation title: Logging Implementation  
Implementation vendor: Jeff Friesen (IV)  
Implementation version: 1.0a  
Specification title: Logging Specification  
Specification vendor: Jeff Friesen (SV)  
Specification version: 1.0  
Sealed: true  
Compatible with 1.0: true
```

Array

The `java.lang.reflect` package also includes an `Array` class whose class methods make it possible to reflectively create and access Java arrays. Listing 8-7 provides a demonstration.

Listing 8-7. Reflectively Creating and Accessing an Array

```
import java.lang.reflect.Array;

public class ArrayDemo {
    public static void main(String[] args) {
        String[] argsCopy = (String[]) Array.newInstance(
            String.class, args.length);
        for (int i = 0; i < args.length; i++)
            Array.set(argsCopy, i, args[i]);
        for (int i = 0; i < args.length; i++)
            System.out.println(Array.get(argsCopy, i));
    }
}
```

Listing 8-7 first invokes `Array`'s `Object newInstance(Class<?> componentType, int length)` class method to create an array that can store `String` objects. It then copies all passed `String` arguments to this array, invoking `Array`'s `void set(Object array, int index, Object value)` class method to store each object. Finally, it retrieves each stored object by invoking `Array`'s `Object get(Object array, int index)` class method.

Compile Listing 8-7 (`javac ArrayDemo.java`) and run this application. For example, consider the following command:

```
java ArrayDemo a b c
```

This command generates the following output:

```
a  
b  
c
```

Exploring StringTokenizer

You'll occasionally want to extract a string's individual components. For example, you have the string "int x = 4;" and want to extract int, x, =, 4, and ; separately. Alternatively, you might have a comma-separated values (CSV) file where each line consists of multiple data items separated by commas and you want to read each value separately.

Note The procedure described in this section is not able to handle general CSV files with escaped delimiters. For that aim you can use one of the generally available CSV parsing libraries.

The task of breaking up a string into its individual components is known as *parsing* or *tokenizing*. The components themselves are known as *tokens*. (Technically, a token is a category, such as identifier, and the component is known as a *lexeme*, such as int.)

Java 1.0 introduced the `java.util.StringTokenizer` class to let applications tokenize strings. `StringTokenizer` lets you specify a string to be tokenized and a set of *delimiters* that separate successive tokens. This class declares three constructors for specifying these items:

- `StringTokenizer(String str)` constructs a string tokenizer for the specified string. The tokenizer uses the default delimiter set, which is "\t\n\r\f": the space character, the tab character, the newline character, the carriage return character, and the form feed character. Delimiter characters themselves won't be treated as tokens. This constructor throws `NullPointerException` when you pass `null` to `str`.
- `StringTokenizer(String str, String delim)` constructs a string tokenizer for the specified string. The characters in the `delim` argument are the delimiters for separating tokens. Delimiter characters themselves won't be treated as tokens. This constructor throws `NullPointerException` when you pass `null` to `str`. Although it doesn't throw an exception when you pass `null` to `delim`, trying

to invoke other methods on the resulting StringTokenizer instance may result in NullPointerException.

- StringTokenizer(String str, String delim, boolean returnDelims) constructs a string tokenizer for the specified string. All characters in the `delim` argument are the delimiters for separating tokens. When the `returnDelims` flag is true, the delimiter characters are also returned as tokens. Each delimiter is returned as a string of length one. When `returnDelims` is false, the delimiter characters are skipped and only serve as separators between tokens. This constructor throws NullPointerException when you pass null to `str`. Although it doesn't throw an exception when you pass null to `delim`, invoking other methods on the resulting StringTokenizer instance may result in NullPointerException.

For all the methods the StringTokenizer provides, please consult the API documentation.

You would typically use StringTokenizer in a loop context, as follows:

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens())
    System.out.println(st.nextToken());
```

This loop uses the standard set of delimiters. It generates the following output:

```
this
is
a
test
```

Alternatively, you could specify the following loop context:

```
StringTokenizer st = new StringTokenizer("this is a test");
Enumeration<String> e = (Enumeration) st;
while (e.hasMoreElements())
    System.out.println(e.nextElement());
```

Note StringTokenizer implements Enumeration so that you can create common code for enumerating tokens and legacy vector/hashtable (see Chapter 9) content.

Although you'll find StringTokenizer easy to use in your Android apps or non-Android Java programs, Java provides a more powerful alternative in the form of regular expressions (discussed in Chapter 14). To give you a taste for the power of regular expressions, you can easily bypass the previous loops for obtaining tokens by employing the following code fragment:

```
String[] values = "this is a test".split("\\s");
```

The String class declares String[] split(String regex) and String[] split(String regex, int limit) methods that let you split a string into components that are separated by delimiters identified by the specified regular expression (regex). For example, \\s represents the \s regular expression (backslashes must be doubled when placed in string literals), which stands for whitespace character. The string is split around whitespace character delimiters.

If you were to specify the following loop

```
for (int i = 0; i < values.length; i++)
    System.out.println(values[i]);
```

you would observe the same output as for the preceding StringTokenizer code.

Exploring Timer and TimerTask

It's often necessary to schedule a *task* (a unit of work) for *one-shot execution* (the task runs only once) or for repeated execution at regular intervals. The java.util.Timer and java.util.TimerTask classes provide a way to accomplish task scheduling.

Note Android apps can use Timer and TimerTask, but for many scheduling tasks in Android, it is better to use a Handler or the AlarmManager. Here we describe Java's Timer and TimerTask in case the Android classes don't suit your needs.

Timer provides a facility for scheduling TimerTasks for future execution on a background thread. Timer tasks may be scheduled for one-shot execution or for repeated execution at regular intervals. Before delving into the internals of these classes, check out Listing 8-8.

Listing 8-8. Displaying the Current Millisecond Value at Approximately 1-Second Intervals

```
import java.util.Timer;
import java.util.TimerTask;

public class TimerDemo {
    public static void main(String[] args) {
        TimerTask task = new TimerTask(){
            @Override
            public void run() {
                System.out.println(System.currentTimeMillis());
            }
        };
        Timer timer = new Timer();
        timer.schedule(task, 0, 1000);
    }
}
```

Listing 8-8 describes an application that outputs the current time (in milliseconds) approximately every second. It first instantiates a TimerTask subclass (in this case, an anonymous class is used) whose overriding run() method outputs the time. It then instantiates Timer and invokes its schedule() method with this task as the first argument. The second and third arguments indicate that the task is scheduled for repeated execution after no initial delay and every 1000 milliseconds.

Compile Listing 8-8 (`javac TimerDemo.java`) and run this application (`java TimerDemo`). You should observe output that's similar to the following truncated output:

```
1380933893664
1380933894666
1380933895668
1380933896668
1380933897670
1380933898672
```

For more details about `Timer` and `TimerTask`, please consult the API documentation. There we learn how to schedule one-shot executions, and executions with fixed delay or at fixed rate, and how to cancel times.

EXERCISES

The following exercises are designed to test your understanding of Chapter 8's content:

1. What does the `Random` class accomplish?
2. What are some of the capabilities offered by the Reflection API?
3. Reflection should not be used indiscriminately. Why not?
4. Identify the class that is the entry point into the Reflection API.
5. True or false: All of the Reflection API is contained in the `java.lang.reflect` package?
6. What are the three ways to obtain a `Class` object?
7. True or false: You can use class literals with primitive types.
8. How do you instantiate a dynamically loaded class?
9. What method do you invoke to obtain a constructor's parameter types?
10. What does `Class`'s `Field getField(String name)` method do when it cannot locate the named field?
11. How do you determine if a method is declared to receive a variable number of arguments?
12. True or false: You can reflectively make a private method accessible.
13. What is the purpose of `Package`'s `isSealed()` method?
14. True or false: `getPackage()` requires at least one class file to be loaded from the package before it returns a `Package` object describing that package.
15. How do you reflectively create and access a Java array?
16. What is the purpose of the `StringTokenizer` class?

17. Identify the standard class library's convenient and simpler alternative to the Threads API for scheduling task execution.
 18. Create a Guess application that uses Random to generate a random integer from 0 to 25. Then, the application repeatedly asks the user to guess which integer was chosen by entering a letter from a through z. The application continues by comparing the entered letter with the random integer (which is offset by lowercase letter a) to learn if the user's choice was too low, too high, or just right, and outputs an appropriate message.
 19. Create a Classify application that uses Class's `forName()` method to load its single command-line argument, which will represent a package-qualified annotation type, enum, interface, or class (the default). Use a chained if-else statement along with the appropriate Class methods and `System.out.println()` to identify the type and output Annotation, Enum, Interface, or Class.
 20. Create a Tokenize application that uses StringTokenizer to extract the month, day, year, hour, minute, and second fields from the string 03-12-2014 03:05:20, which are then output.
 21. Create a BackAndForth application that uses Timer and TimerTask to repeatedly move an asterisk forward 20 steps and then backward 20 steps. The asterisk is output via `System.out.print()`.
-

Summary

The Math class's `random()` method is implemented in terms of the Random class, whose instances are known as random number generators. Random generates a sequence of random numbers by starting with a special 48-bit seed. This value is subsequently modified via a mathematical algorithm that is known as a linear congruent generator.

Random declares a pair of constructors, a `setSeed()` method for setting the random number generator's seed, and several "next"-prefixed methods that return the next number in the sequence as a Boolean value, an integer, a long integer, a floating-point value, or a double-precision floating-point value. There is even a method for generating a sequence of random bytes.

Java's Reflection API offers a third RTTI form in which applications can dynamically load and learn about loaded classes and other reference types. The API also lets applications instantiate classes, call methods, access fields, and perform other tasks reflectively. This form of RTTI is known as reflection or introspection.

The `java.lang` package's `Class` class is the entry point into the Reflection API, whose types are stored mainly in the `java.lang.reflect` package. You can obtain a `Class` object by invoking `Class`'s `forName()` method, by invoking `Object`'s `getClass()` method, or by using a class literal, which is the name of a class followed by a `.class` suffix.

The `java.lang.reflect` package declares `Constructor`, `Field`, and `Method` classes that represent constructors, fields, and methods. Each of these classes extends the `AccessibleObject` class, which provides methods for determining if the constructor, field, or method is accessible, and for changing the constructor's, field's, or method's accessibility.

The `Package` class provides access to package information. This information includes version information about the implementation and specification of a Java package, the package's name, and an indication of whether the package is sealed or not. Also, the `Array` class provides class methods that make it possible to reflectively create and access Java arrays.

You'll occasionally want to extract a string's individual components. The task of breaking up a string into its individual components is known as parsing or tokenizing. The components themselves are known as tokens. Java 1.0 introduced the `StringTokenizer` class to let applications tokenize strings.

`StringTokenizer` provides constructors for specifying the string to be tokenized and the set of delimiters that separate successive tokens. The number of tokens can be obtained by invoking the `countTokens()` method.

The `hasMoreElements()` and `hasMoreTokens()` methods tell you whether there are more tokens to extract. The `nextToken()` and `nextElement()` methods return the next token. One of the `nextToken()` methods also lets you specify a new set of delimiters.

`Timer` provides a facility for scheduling `TimerTasks` for future execution on a background thread. Timer tasks may be scheduled for one-shot execution or for repeated execution at regular intervals. Corresponding to each `Timer` object is a single background thread that's used to execute all of the timer tasks sequentially.

This chapter completes our tour of Java's basic APIs. Chapter 9 explores Java's Collections Framework and classic collections APIs.

CHAPTER 9

Exploring the Collections Framework

Applications often must manage collections of objects. Although you can use arrays for this purpose, they are not always a good choice. For example, arrays have fixed sizes, making it tricky to determine an optimal size when you need to store a variable number of objects. Also, arrays can be indexed by integers only, making them unsuitable for mapping arbitrary objects to other objects.

The standard class library provides the Collections Framework and legacy utility APIs to manage collections on behalf of applications. In this chapter, we first present this framework and then introduce you to these legacy APIs (in case you encounter them in legacy code). As you will discover, some of the legacy APIs are still useful.

Note Java’s Concurrency Utilities API suite (discussed in Chapter 11) extends the Collections Framework.

Exploring Collections Framework Fundamentals

The *Collections Framework* is a group of types (mainly located in the `java.util` package) that offers a standard architecture for representing and manipulating *collections*, which are groups of objects stored in instances of classes designed for this purpose. This framework’s architecture is divided into three sections:

- *Core interfaces*: The framework provides core interfaces for manipulating collections independently of their implementations.

- *Implementation classes:* The framework provides classes that offer different core interface implementations to address performance and other requirements.
- *Utility classes:* The framework provides utility classes with methods for sorting arrays, obtaining synchronized collections, and more.

The core interfaces include `java.lang.Iterable`, `Collection`, `List`, `Set`, `SortedSet`, `NavigableSet`, `Queue`, `Deque`, `Map`, `SortedMap`, and `NavigableMap`. `Collection` extends `Iterable`; `List`, `Set`, and `Queue` each extend `Collection`; `SortedSet` extends `Set`; `NavigableSet` extends `SortedSet`; `Deque` extends `Queue`; `SortedMap` extends `Map`; and `NavigableMap` extends `SortedMap`.

Figure 9-1 illustrates the core interfaces hierarchy (arrows point to parent interfaces).

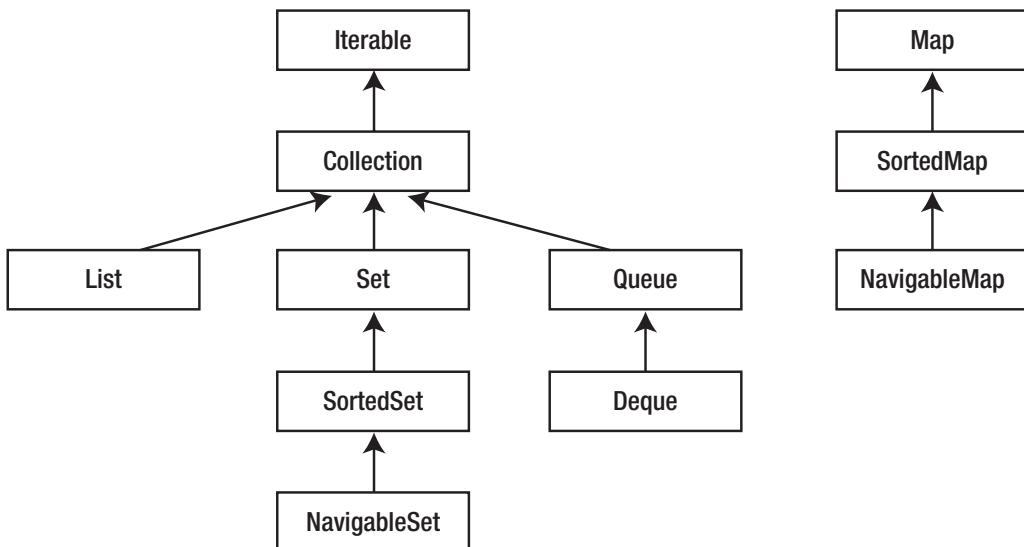


Figure 9-1. The Collections Framework is based on a hierarchy of core interfaces

The framework's implementation classes include `ArrayList`, `LinkedList`, `TreeSet`, `HashSet`, `LinkedHashSet`, `EnumSet`, `PriorityQueue`, `ArrayDeque`, `TreeMap`, `HashMap`, `LinkedHashMap`, `IdentityHashMap`, `WeakHashMap`, and `EnumMap`. The name of each concrete class ends in a core interface name, identifying the core interface on which it is based.

Note Additional implementation classes are part of the concurrency utilities.

The framework's implementation classes also include the abstract `AbstractCollection`, `AbstractList`, `AbstractSequentialList`, `AbstractSet`, `AbstractQueue`, and `AbstractMap` classes. These classes offer skeletal implementations of the core interfaces to facilitate the creation of concrete implementation classes.

Finally, the framework provides two utility classes: `Arrays` and `Collections`.

Comparable vs. Comparator

A collection implementation stores its elements in some *order* (arrangement). This order may be unsorted, or it may be sorted according to some criterion (such as alphabetical, numerical, or chronological).

A sorted collection implementation defaults to storing its elements according to their *natural ordering*. For example, the natural ordering of `java.lang.String` objects is *lexicographic* or *dictionary* (also known as alphabetical) order.

A collection cannot rely on `equals()` to dictate natural ordering because this method can only determine if two elements are equivalent. Instead, element classes must implement the `java.lang.Comparable<T>` interface and its `int compareTo(T o)` method.

Note According to Comparable's Oracle-based Java documentation, this interface is considered to be part of the Collections Framework even though it is a member of the `java.lang` package.

A sorted collection uses `compareTo()` to determine the natural ordering of this method's element argument `o` in a collection. `compareTo()` compares argument `o` with the current element (which is the element on which `compareTo()` was called) and does the following:

- It returns a negative value when the current element should precede `o`.
- It returns a zero value when the current element and `o` are the same.
- It returns a positive value when the current element should succeed `o`.

When you need to implement Comparable's `compareTo()` method, there are some rules that you must follow. The following rules are similar to those shown in Chapter 4 for implementing the `equals()` method:

- **`compareTo()` must be reflexive:** For any nonnull reference value x , $x.compareTo(x)$ must return 0.
- **`compareTo()` must be symmetric:** For any nonnull reference values x and y , $x.compareTo(y) == -y.compareTo(x)$ must hold.
- **`compareTo()` must be transitive:** For any nonnull reference values x , y , and z , if $x.compareTo(y) > 0$ is true, and if $y.compareTo(z) > 0$ is true, then $x.compareTo(z) > 0$ must also be true.

Also, `compareTo()` should throw `java.lang.NullPointerException` when the null reference is passed to this method. However, you don't need to check for null because this method throws `NullPointerException` when it attempts to access a null reference's nonexistent members.

You might occasionally need to store in a collection objects that are sorted in some order that differs from their natural ordering. In this case, you would supply a comparator to provide that ordering.

A *comparator* is an object whose class implements the `Comparator` interface. This interface, whose generic type is `Comparator<T>`, provides the following pair of methods:

- `int compare(T o1, T o2)` compares both arguments for order. This method returns 0 when $o1$ equals $o2$, a negative value when $o1$ is less than $o2$, and a positive value when $o1$ is greater than $o2$.
- `boolean equals(Object o)` returns true when o "equals" this `Comparator` in that o is also a `Comparator` and imposes the same ordering. Otherwise, this method returns false.

Note `Comparator` declares `equals()` because this interface places an extra condition on this method's contract. *Additionally, this method can return true only if the specified object is also a comparator and it imposes the same ordering as this comparator. You don't have to override `equals()`, but doing so may improve performance by allowing programs to determine that two distinct comparators impose the same order.*

Chapter 6 provided an example that illustrated implementing Comparable, and you will discover another example later in this chapter. Also, in this chapter, we will present examples of implementing Comparator.

Iterable and Collection

Most of the core interfaces are rooted in Iterable and its Collection subinterface. Their generic types are Iterable<T> and Collection<E>.

Iterable describes any object that can return its contained objects in some sequence. This interface declares an Iterator<T> iterator() method that returns an Iterator instance for iterating over all of the contained objects.

Collection represents a collection of objects that are known as *elements*. This interface provides methods that are common to the Collection subinterfaces on which many collections are based. Table 9-1 describes these methods. For details please consult the API documentation at <https://docs.oracle.com/javase/8/docs/api>.

The documentation reveals three exceptional things about various Collection methods. First, some methods can throw instances of the UnsupportedOperationException class. For example, add() throws UnsupportedOperationException when you attempt to add an object to an *immutable* (unmodifiable) collection (discussed later in this chapter).

Second, some of Collection's methods can throw instances of the ClassCastException class. For example, remove() throws ClassCastException when you attempt to remove an entry (also known as mapping) from a tree-based map whose keys are Strings, but specify a non-String key instead.

Finally, Collection's add() and addAll() methods throw IllegalArgumentException instances when some *property* (attribute) of the element to be added prevents it from being added to this collection. For example, a third-party collection class's add() and addAll() methods might throw this exception when they detect negative Integer values.

Note Perhaps you're wondering why remove() is declared to accept any Object argument instead of accepting only objects whose types are those of the collection. In other words, why is remove() not declared as boolean remove (E e)? Also, why are containsAll(), removeAll(), and retainAll() not declared with an argument of type Collection<? extends E> to ensure that

the collection argument only contains elements of the same type as the collection on which these methods are called? The answer to these questions is the need to maintain backward compatibility. The Collections Framework was introduced before Java 5 and its introduction of generics. To let legacy code written before version 5 continue to compile, these four methods were declared with weaker type constraints.

Iterator and the Enhanced For Loop Statement

By extending Iterable, Collection inherits that interface's iterator() method, which makes it possible to iterate over a collection. iterator() returns an instance of a class that implements the Iterator interface, whose generic type is expressed as Iterator<E> and which declares the following three methods:

- boolean hasNext() returns true when this Iterator instance has more elements to return; otherwise, this method returns false.
- E next() returns the next element from the collection associated with this Iterator instance or throws NoSuchElementException when there are no more elements to return.
- void remove() removes the last element returned by next() from the collection associated with this Iterator instance.
This method can be called only once per next() call. The behavior of an Iterator instance is unspecified when the underlying collection is modified while iteration is in progress in any way other than by calling remove(). This method throws UnsupportedOperationException when it is not supported by this Iterator, and IllegalStateException when remove() has been called without a previous call to next() or when multiple remove() calls occur with no intervening next() calls.

The following example shows you how to iterate over a collection after calling iterator() to return an Iterator instance:

```
Collection<String> col = Arrays.asList("John", "Linda"); // Some way to create
                                                       a collection
Iterator iter = col.iterator();
```

```
while (iter.hasNext())
    System.out.println(iter.next());
```

The while loop repeatedly calls the iterator's `hasNext()` method to determine whether or not iteration should continue, and (if it should continue) the `next()` method to return the next element from the associated collection.

Because this idiom is commonly used, Java 5 introduced syntactic sugar to the for loop statement to simplify iteration in terms of the idiom. This sugar makes this statement appear like the foreach statement found in languages such as Perl and is revealed in the following simplified equivalent of the previous example:

```
Collection<String> col = Arrays.asList("John", "Linda");
// Some way to create a collection
for (String s: col)
    System.out.println(s);
```

This sugar hides `col.iterator()`, a method call that returns an `Iterator` instance for iterating over `col`'s elements. It also hides calls to `Iterator`'s `hasNext()` and `next()` methods on this instance. You interpret this sugar to read as follows: “for each `String` object in `col`, assign this object to `s` at the start of the loop iteration.”

Note The enhanced for loop statement is also useful in an arrays context, in which it hides the array index variable. Consider the following example:

```
String[] verbs = { "run", "walk", "jump" };
for (String verb: verbs)
    System.out.println(verb);
```

The enhanced for loop statement is limited in that you cannot use this statement where access to the iterator is required to remove an element from a collection. Also, it's not usable where you must replace elements in a collection/array during a traversal, and it cannot be used where you must iterate over multiple collections or arrays in parallel.

Autoboxing and Unboxing

Developers who believe that Java should support only reference types have complained about Java's support for primitive types. One area where the dichotomy of Java's type system is clearly seen is the Collections Framework: you can store objects but not primitive type-based values in collections.

Although you cannot directly store a primitive type-based value in a collection, you can indirectly store this value by first wrapping it in an object created from a primitive type wrapper class such as `Integer` and then storing this primitive type wrapper class instance in the collection—see the following example:

```
Collection<Integer> col = new ArrayList<>();
    // Some way to create an empty collection
int x = 27;
col.add(new Integer(x));
    // Indirectly store int value 27 via an Integer object.
```

The reverse situation is also tedious. When you want to retrieve the `int` from `col`, you must invoke `Integer`'s `intValue()` method (which, if you recall, is inherited from `Integer`'s `java.lang.Number` superclass). Continuing on from this example, you would specify `int y = col.iterator().next().intValue();` to assign the stored 32-bit integer to `y`.

To alleviate this tedium, Java 5 introduced autoboxing and unboxing, a pair of complementary syntactic sugar-based language features that make primitive-type values appear more like objects. (This “sleight of hand” isn't complete because you cannot specify expressions such as `27.doubleValue()`.)

Autoboxing automatically *boxes* (wraps) a primitive-type value in an object of the appropriate primitive type wrapper class whenever a primitive-type value is specified but a reference is required. For example, you could change the example's third line to `col.add(x);` and have the compiler box `x` into an `Integer` object.

Unboxing automatically *unboxes* (unwraps) a primitive-type value from its wrapper object whenever a reference is specified but a primitive-type value is required. For example, you could specify `int y = col.iterator().next();` and have the compiler unbox the returned `Integer` object to `int` value 27 prior to the assignment.

Although autoboxing and unboxing were introduced to simplify working with primitive-type values in a collections context, these language features can be used in other contexts; and this arbitrary use can lead to a problem that is difficult to understand

without knowledge of what is happening behind the scenes. Consider the following example: Integer i1 = 30000; Integer i2 = 30000. While 30000 == 30000 certainly is true, i1 == i2 doesn't have to be true, because those are possibly two different objects! Sometimes Java caches such objects, so i1 == i2 *might* be true. The point is: you just can't rely on it!

Caution Don't assume that autoboxing and unboxing are useful in the context of the == and != operators.

Exploring Lists

A *list* is an ordered collection, which is also known as a *sequence*. Elements can be stored in and accessed from specific locations via integer indexes. Some of these elements may be duplicates or null (when the list's implementation allows null elements). Lists are described by the List interface, whose generic type is List<E>.

List extends Collection and redeclares its inherited methods, partly for convenience. It also redeclares iterator(), add(), remove(), equals(), and hashCode() to place extra conditions on their contracts. For example, List's contract for add() specifies that it appends an element to the end of the list rather than just somehow adding the element to the collection.

The most important constructs you will use for lists are using a constructor to create a list given a collection, adding elements via add(), and iterating over list elements:

```
List<Integer> l = new ArrayList<>(); // or some other implementation  
// Initializing with some other collection:  
// List<Integer> l = new ArrayList<>(someOtherColl);  
  
l.add(42);  
l.add(-37_498);  
  
for(Integer i : l) {  
    System.out.println(i);  
}
```

For more details and all the other methods and their descriptions, please see the API documentation.

Note The Iterator and ListIterator instances that are returned by the iterator() and listIterator() methods in the ArrayList and LinkedList List implementation classes are *fail-fast*: when a list is structurally modified (e.g., by calling the implementation's add() method to add a new element) after the iterator is created, in any way except through the iterator's own add() and remove() methods, the iterator throws ConcurrentModificationException. Therefore, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, nondeterministic behavior at an undetermined time in the future.

ArrayList

The ArrayList class provides a versatile list implementation that is based on an internal array. As a result, access to the list's elements is fast. However, because elements must be moved to open a space for insertion or to close a space after deletion, insertions and deletions of elements are slow (although some internal algorithm makes sure this wouldn't happen too often).

ArrayList supplies three constructors:

- ArrayList() creates an empty array list with an initial *capacity* (storage space) of ten elements. Once this capacity is reached, a larger array is allocated, elements from the current array are copied into the larger array, and the larger array becomes the new current array. This process repeats as additional elements are added to the array list.
- ArrayList(Collection<? extends E> c) creates an array list containing c's elements in the order in which they are returned by c's iterator. NullPointerException is thrown when c contains the null reference.
- ArrayList(int initialCapacity) creates an empty array list with an initial capacity of initialCapacity elements. IllegalArgumentException is thrown when initialCapacity is negative.

Listing 9-1 demonstrates an array list.

Listing 9-1. A Demonstration of an Array-Based List

```
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;

public class ArrayListDemo {
    public static void main(String[] args) {
        // We must wrap the "Arrays.asList()" into a "new ArrayList()", otherwise
        // the remove() below won't work.
        List<String> ls = new ArrayList(
            Arrays.asList("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat") );
        System.out.println("ls: " + ls);
        ls.set(ls.indexOf("Wed"), "Wednesday");
        System.out.println("ls: " + ls);
        ls.remove(ls.lastIndexOf("Fri"));
        System.out.println("ls: " + ls);
    }
}
```

`ArrayListDemo` creates an array list and an array of short weekday names. It then dumps the list to standard output, changes one of the list entries, dumps the list again, removes a list entry, and dumps the list one last time. The `dump()` method's enhanced for loop statement uses `iterator()`, `hasNext()`, and `next()` behind the scenes.

When you run this application, it generates the following output:

```
ls: [ Sun, Mon, Tue, Wed, Thu, Fri, Sat ]
ls: [ Sun, Mon, Tue, Wednesday, Thu, Fri, Sat ]
ls: [ Sun, Mon, Tue, Wednesday, Thu, Sat ]
```

LinkedList

The `LinkedList` class provides a list implementation that is based on linked nodes (a node is either an element or a subtree in a tree representation). Because links must be traversed, access to the list's elements is slower compared to `ArrayList`. However,

because only node references need to be changed, insertions and deletions of elements are faster compared to `ArrayList`.

Exploring Sets

A *set* is a collection that contains no duplicate elements. In other words, a set contains no pair of elements *e1* and *e2* such that *e1.equals(e2)* returns true. Furthermore, a set can contain at most one null element. Sets are described by the `Set` interface, whose generic type is `Set<E>`.

`Set` extends `Collection` and redeclares its inherited methods, for convenience and also to add stipulations to the contracts for `add()`, `equals()`, and `hashCode()` to address how they behave in a set context.

TreeSet

The `TreeSet` class provides a set implementation that is based on a tree data structure. As a result, elements are stored in sorted order. However, accessing these elements is somewhat slower than with the other `Set` implementations (which are not sorted) because links must be traversed.

Note Check out Wikipedia's "tree (data structure)" entry ([http://en.wikipedia.org/wiki/Tree_\(data_structure\)](http://en.wikipedia.org/wiki/Tree_(data_structure))) to learn about trees.

`TreeSet` supplies four constructors:

- `TreeSet()` creates a new, empty tree set that is sorted according to the natural ordering of its elements. All elements inserted into the set must implement the `Comparable` interface.
- `TreeSet(Collection<? extends E> c)` creates a new tree set containing *c*'s elements, sorted according to the natural ordering of its elements. All elements inserted into the new set must implement the `Comparable` interface. This constructor throws `ClassCastException` when *c*'s elements don't implement `Comparable` or are not mutually comparable and `NullPointerException` when *c* contains the null reference.

- `TreeSet(Comparator<? super E> comparator)` creates a new, empty tree set that is sorted according to the specified comparator. Passing `null` to `comparator` implies that natural ordering will be used.
- `TreeSet(SortedSet<E> ss)` creates a new tree set containing the same elements and using the same ordering as `ss`. (We discuss sorted sets later in this chapter.) This constructor throws `NullPointerException` when `ss` contains the null reference.

Listing 9-2 demonstrates a tree set.

Listing 9-2. A Demonstration of a Tree Set with String Elements Sorted According to Their Natural Ordering

```
import java.util.Set;
import java.util.TreeSet;
import java.util.Arrays;

public class TreeSetDemo {
    public static void main(String[] args) {
        Set<String> ss = new TreeSet<>(Arrays.asList(
            "apples", "pears", "grapes", "bananas", "kiwis"
        ));
        System.out.println("ss: " + ss);
    }
}
```

`TreeSetDemo` creates a tree set and an array of fruit names. It then dumps the set to standard output. Because `String` implements `Comparable`, it's legal for this application to use strings as elements.

When you run this application, it generates the following output:

```
ss: [ apples, bananas, grapes, kiwis, pears ]
```

HashSet

The HashSet class provides a set implementation that is backed by a hashtable data structure (implemented as a HashMap instance, discussed later, which provides a quick way to determine if an element has already been stored in this structure). Although this class provides no ordering guarantees for its elements, HashSet is much faster than TreeSet. Furthermore, HashSet permits the null reference to be stored in its instances.

Note Check out Wikipedia’s “Hash table” entry (http://en.wikipedia.org/wiki/Hash_table) to learn about hashtables.

For HashSet’s constructors and methods, see the API documentation. The identity of an element in a HashSet is closely connected to its hashCode() result. So if you want to store your own classes in a HashSet, you must make sure they obey the usual hashCode() contract.

A LinkedHashSet is a subclass of HashSet that uses a linked list to store its elements. As a result, LinkedHashSet’s iterator returns elements in the order in which they were inserted. LinkedHashSet offers slower writing performance than HashSet and faster writing performance than TreeSet.

EnumSet

The EnumSet class provides a Set implementation that is based on a bit vector (bits in a long or int indicating whether an element is present or not). Operations are extremely fast compared to the other set implementations. See Listing 9-3 for an example.

Listing 9-3. Creating the EnumSet Equivalent of DAYS_OFF

```
import java.util.EnumSet;
import java.util.Iterator;
import java.util.Set;

enum Weekday {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

```

public class EnumSetDemo {
    public static void main(String[] args) {
        Set<Weekday> daysOff = EnumSet.of(Weekday.SUNDAY, Weekday.MONDAY);
        for(Weekday wd : daysOff)
            System.out.println(wd);
    }
}

```

Exploring Sorted Sets

`TreeSet` is an example of a *sorted set*, which is a set that maintains its elements in ascending order, sorted according to their natural ordering or according to a comparator that is supplied when the sorted set is created. Sorted sets are described by the `SortedSet` interface.

`SortedSet`, whose generic type is `SortedSet<E>`, extends `Set`. With two exceptions, the methods it inherits from `Set` behave identically on sorted sets as on other sets:

- The `Iterator` instance returned from `iterator()` traverses the sorted set in ascending element order.
- The array returned by `toArray()` contains the sorted set's elements in order.

Note Although not guaranteed, the `toString()` methods of `SortedSet` implementations in the Collections Framework (such as `TreeSet`) return a string containing all of the sorted set's elements in order.

The set-based range views returned from the `headSet()`, `subSet()`, and `tailSet()` methods are analogous to the list-based range view returned from `List`'s `subList()` method except that a set-based range view remains valid even when the backing sorted set is modified. As a result, a set-based range view can be used for a lengthy period of time.

Note Unlike a list-based range view whose endpoints are elements in the backing list, the endpoints of a set-based range view are absolute points in element space, allowing a set-based range view to serve as a window onto a portion of the set's element space. Any changes made to the set-based range view are written back to the backing sorted set and vice versa.

[Listing 9-4](#) demonstrates a sorted set based on a tree set.

Listing 9-4. A Sorted Set of Fruit and Vegetable Names

```
import java.util.SortedSet;
import java.util.TreeSet;
import java.util.Arrays;

public class SortedSetDemo {
    public static void main(String[] args) {
        SortedSet<String> sss = new TreeSet<String>();
        String[] fruitAndVeg = {
            "apple", "potato", "turnip", "banana", "corn", "carrot", "cherry",
            "pear", "mango", "strawberry", "cucumber", "grape", "banana",
            "kiwi", "radish", "blueberry", "tomato", "onion", "raspberry",
            "lemon", "pepper", "squash", "melon", "zucchini", "peach", "plum",
            "turnip", "onion", "nectarine"
        };
        System.out.println("Array size = " + fruitAndVeg.length);

        sss.addAll(Arrays.asList(fruitAndVeg));
        System.out.println("sss: " + sss);
        System.out.println("Sorted set size = " + sss.size());
        System.out.println("First element = " + sss.first());
        System.out.println("Last element = " + sss.last());
        System.out.println("Comparator = " + sss.comparator());
        System.out.println("hs: " + sss.headSet("n"));
        System.out.println("ts: " + sss.tailSet("n"));
        System.out.println("Count of p-named fruits & vegetables = " +
                           sss.subSet("p", "q").size());
    }
}
```

```

        System.out.println("Incorrect count of c-named fruits & vegetables = " +
                           sss.subSet("carrot", "cucumber").size());
        System.out.println("Correct count of c-named fruits & vegetables = " +
                           sss.subSet("carrot", "cucumber\0").size());
    }
}

```

SortedSetDemo creates a sorted set and an array of fruit and vegetable names and then proceeds to populate the set from this array. After dumping out the set's contents, it outputs information about the set, including head and tail views of portions of the set.

When you run this application, it generates the following output:

```

Array size = 29
sss: [ apple, banana, blueberry, carrot, cherry, corn, cucumber, grape,
kiwi, lemon, mango, melon, nectarine, onion, peach, pear, pepper, plum,
potato, radish, raspberry, squash, strawberry, tomato, turnip, zucchini ]
Sorted set size = 26
First element = apple
Last element = zucchini
Comparator = null
hs: [ apple, banana, blueberry, carrot, cherry, corn, cucumber, grape,
kiwi, lemon, mango, melon ]
ts: [ nectarine, onion, peach, pear, pepper, plum, potato, radish,
raspberry, squash, strawberry, tomato, turnip, zucchini ]
Count of p-named fruits & vegetables = 5
Incorrect count of c-named fruits & vegetables = 3
Correct count of c-named fruits & vegetables = 4

```

This output reveals that the sorted set's size is less than the array's size because a set cannot contain duplicate elements: the duplicate banana, turnip, and onion elements are not stored in the sorted set.

The comparator() method returns null because the sorted set was not created with a comparator. Instead, the sorted set relies on the natural ordering of String elements to store them in sorted order.

The headSet() and tailSet() methods are called with argument "n" to return, respectively, a set of elements whose names begin with a letter that is strictly less than n and a letter that is greater than or equal to n.

Finally, the output shows you that you must be careful when passing an upper limit to `subSet()`. As you can see, `sss.subSet("carrot", "cucumber")` doesn't include `cucumber` in the returned range view because `cucumber` is `subSet()`'s high endpoint.

To include `cucumber` in the range view, you need to form a *closed range* or *closed interval* (both endpoints are included). With `String` objects, you accomplish this task by appending `\0` to the string. For example, `sss.subSet("carrot", "cucumber\0")` includes `cucumber` because it is less than `cucumber\0`.

This same technique can be applied wherever you need to form an *open range* or *open interval* (neither endpoint is included). For example, `sss.subSet("carrot\0", "cucumber")` doesn't include `carrot` because it is less than `carrot\0`. Furthermore, it doesn't include high endpoint `cucumber`.

Note When you want to create closed and open ranges for elements created from your own classes, you need to provide some form of `predecessor()` and `successor()` methods that return an element's predecessor and successor.

You need to be careful when designing classes that work with sorted sets. For example, the class must implement `Comparable` when you plan to store the class's instances in a sorted set where these elements are sorted according to their natural ordering.

Exploring Navigable Sets

`TreeSet` is an example of a *navigable set*, which is a sorted set that can be iterated over in descending order as well as ascending order and which can report closest matches for given search targets. Navigable sets are described by the `NavigableSet` interface, whose generic type is `NavigableSet<E>`, which extends `SortedSet`. Have a look at the API documentation to learn about its methods.

[Listing 9-5](#) demonstrates a navigable set based on a tree set.

Listing 9-5. Navigating a Set of Integers

```
import java.util.Iterator;
import java.util.NavigableSet;
import java.util.TreeSet;
import java.util.Arrays;
```

```

public class NavigableSetDemo {
    public static void main(String[] args) {
        NavigableSet<Integer> ns = new TreeSet<Integer>(
            Arrays.asList( 82, -13, 4, 0, 11, -6, 9 ) );
        System.out.print("Ascending order: ");
        Iterator iter = ns.iterator();
        while (iter.hasNext())
            System.out.print(iter.next() + " ");
        System.out.println();
        System.out.print("Descending order: ");
        iter = ns.descendingIterator();
        while (iter.hasNext())
            System.out.print(iter.next() + " ");
        System.out.println("\n");
        outputClosestMatches(ns, 4);
        outputClosestMatches(ns.descendingSet(), 12);
    }

    static void outputClosestMatches(NavigableSet<Integer> ns, int i) {
        System.out.println("Element < " + i + " is " + ns.lower(i));
        System.out.println("Element <= " + i + " is " + ns.floor(i));
        System.out.println("Element > " + i + " is " + ns.higher(i));
        System.out.println("Element >= " + i + " is " + ns.ceiling(i));
        System.out.println();
    }
}

```

Listing 9-5 creates a navigable set of Integer elements. It takes advantage of autoboxing to ensure that ints are converted to Integers.

When you run this application, it generates the following output:

Ascending order: -13 -6 0 4 9 11 82

Descending order: 82 11 9 4 0 -6 -13

Element < 4 is 0

Element <= 4 is 4

Element > 4 is 9

Element >= 4 is 4

```

Element < 12 is 82
Element <= 12 is 82
Element > 12 is 11
Element >= 12 is 11

```

The first four output lines beginning with Element pertain to an ascending-order set where the element being matched (4) is a member of the set. The second four Element-prefixed lines pertain to a descending-order set where the element being matched (12) is not a member.

As well as letting you conveniently locate set elements via its closest-match methods (`ceiling()`, `floor()`, `higher()`, and `lower()`), `NavigableSet` lets you return set views containing all elements within certain ranges, as demonstrated by the following examples:

- `ns.subSet(-13, true, 9, true)`: Returns all elements from -13 through 9.
- `ns.tailSet(-6, false)`: Returns all elements greater than -6.
- `ns.headSet(4, true)`: Returns all elements less than or equal to 4.

Finally, you can return and remove from the set the first (lowest) element by calling `pollFirst()` and the last (highest) element by calling `pollLast()`. For example, `ns.pollFirst()` removes and returns -13, and `ns.pollLast()` removes and returns 82.

Exploring Queues

A *queue* is a collection in which elements are stored and retrieved in a specific order. Most queues are categorized as one of the following:

- *First-in, first-out (FIFO) queue*: Elements are inserted at the queue's *tail* and removed at the queue's *head*.
- *Last-in, first-out (LIFO) queue*: Elements are inserted and removed at one end of the queue such that the last element inserted is the first element retrieved. This kind of queue behaves as a *stack*.
- *Priority queue*: Elements are inserted according to their natural ordering or according to a comparator that is supplied to the queue implementation.

Queue, whose generic type is `Queue<E>`, extends `Collection`, redeclaring `add()` to adjust its contract (insert the specified element into this queue if it's possible to do so immediately without violating capacity restrictions) and inheriting the other methods from `Collection`. The API documentation of interface `Queue` tells you the details of all methods.

PriorityQueue

The `PriorityQueue` class provides an implementation of a *priority queue*, which is a queue that orders its elements according to their natural ordering or by a comparator provided when the queue is instantiated. Priority queues don't permit null elements and don't permit insertion of non-`Comparable` objects when relying on natural ordering.

The element at the head of the priority queue is the least element with respect to the specified ordering. When multiple elements are tied for least element, one of those elements is arbitrarily chosen as the least element. Similarly, the element at the tail of the priority queue is the greatest element, which is arbitrarily chosen when there is a tie.

Priority queues are unbounded but have a capacity that governs the size of the internal array that is used to store the priority queue's elements. The capacity value is at least as large as the queue's length and grows automatically as elements are added to the priority queue.

The API documentation shows you details about all the constructors and methods of class `PriorityQueue`.

Listing 9-6 demonstrates a priority queue.

Listing 9-6. Adding Randomly Generated Integers to a Priority Queue

```
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueDemo {
    public static void main(String[] args) {
        Queue<Integer> qi = new PriorityQueue<Integer>();
        for (int i = 0; i < 15; i++)
            qi.add((int) (Math.random() * 100));
```

```

        while (!qi.isEmpty())
            System.out.print(qi.poll() + " ");
        System.out.println();
    }
}

```

After creating a priority queue, `PriorityQueueDemo`'s main thread adds 15 randomly generated integers (ranging from 0 through 99) to this queue. It then enters a while loop that repeatedly polls the priority queue for the next element and outputs that element until the queue is empty.

When you run this application, it outputs a line of 15 integers in ascending numerical order from left to right. For example, we observed the following output from one run:

```
30 43 53 61 61 66 66 67 76 78 80 83 87 90 97
```

Suppose you want to reverse the order of the previous example's output so that the largest element appears on the left and the smallest element appears on the right. As Listing 9-7 demonstrates, you can achieve this task by passing a comparator to the appropriate `PriorityQueue` constructor.

Listing 9-7. Using a Comparator with a Priority Queue

```

import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueDemo {
    final static int NELEM = 15;

    public static void main(String[] args) {
        Comparator<Integer> cmp;
        cmp = new Comparator<Integer>() {
            @Override
            public int compare(Integer e1, Integer e2) {
                return e2 - e1;
            }
        };
    }
}

```

```

Queue<Integer> qi = new PriorityQueue<Integer>(NELEM, cmp);
for (int i = 0; i < NELEM; i++)
    qi.add((int) (Math.random() * 100));
while (!qi.isEmpty())
    System.out.print(qi.poll() + " ");
System.out.println();
}
}

```

Listing 9-7 is similar to Listing 9-6, but there are some differences. First, we have declared a constant named NELEM so that we can easily change both the priority queue's initial capacity and the number of elements inserted into the priority queue by specifying the new value in one place.

Second, Listing 9-7 declares and instantiates an anonymous class that implements Comparator. Its compareTo() method subtracts element e2 from element e1 to achieve descending numerical order. The compiler handles the task of unboxing e2 and e1 by converting e2 - e1 to e2.intValue() - e1.intValue().

Finally, Listing 9-7 passes an initial capacity of NELEM elements and the instantiated comparator to the PriorityQueue(int initialCapacity, Comparator<? super E> comparator) constructor. The priority queue will use this comparator to order these elements.

Run this application and you will now see a single output line of 15 integers shown in descending numerical order from left to right. For example, we observed this output line:

```
97 72 70 70 67 64 56 43 36 22 9 5 3 2 1
```

Exploring Deques

A *deque* (pronounced deck) is a double-ended queue in which element insertion or removal occurs at its *head* or *tail*. Deques can be used as queues or stacks.

Deque, whose generic type is Deque<E>, extends Queue in which the inherited add(E e) method inserts e at the deque's tail.

As the documentation reveals, Deque declares methods to access elements at both ends of the deque. Methods are provided to insert, remove, and examine the element. Each of these methods exists in two forms: one throws an exception when the operation fails and the other returns a special value (either null or false, depending on the

operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Deque implementations; in most implementations, insert operations cannot fail.

When a deque is used as a queue, you observe FIFO behavior. Elements are added at the end of the deque and removed from the beginning. The methods inherited from the Queue interface are precisely equivalent to the Deque methods as indicated in Table 9-1.

Table 9-1. Queue and Equivalent Deque Methods

Queue Method	Equivalent Deque Method
<code>add(e)</code>	<code>addLast(e)</code>
<code>offer(e)</code>	<code>offerLast(e)</code>
<code>remove()</code>	<code>removeFirst()</code>
<code>poll()</code>	<code>pollFirst()</code>
<code>element()</code>	<code>getFirst()</code>
<code>peek()</code>	<code>peekFirst()</code>

Finally, deques can also be used as LIFO stacks. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque. Because a stack's `push(e)` method would be equivalent to Deque's `addFirst(e)` method, its `pop()` method would be equivalent to Deque's `removeFirst()` method, and its `peek()` method would be equivalent to Deque's `peekFirst()` method, Deque declares the `E peek(), E pop(), and void push(E e)` stack-oriented convenience methods.

ArrayDeque

The `ArrayDeque` class provides a resizable-array implementation of the `Deque` interface. It prohibits null elements from being added to a deque, and its `iterator()` method returns fail-fast iterators.

Listing 9-8 demonstrates an array deque.

Listing 9-8. Using an Array Deque as a Stack

```

import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Arrays;

public class ArrayDequeDemo {
    public static void main(String[] args) {

        Deque<String> stack = new ArrayDeque<>( Arrays.asList(
            "Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday" ) );
        while (stack.peek() != null)
            System.out.println(stack.pop());
    }
}

```

ArrayDequeDemo creates a deque for use as a stack and an array of weekday names. It then pops them, outputting the names in reverse order.

When you run this application, it generates the following output:

```

Saturday
Friday
Thursday
Wednesday
Tuesday
Monday
Sunday

```

Exploring Maps

A *map* is a group of key/value pairs (also referred to as *entries*). Because the *key* identifies an entry, a map cannot contain duplicate keys. Furthermore, each key can map to at most one value. Maps are described by the Map interface, which has no parent interface, and whose generic type is Map<K,V> (K is the key's type; V is the value's type).

The API documentation describes Map's methods. You'll most often use `put(K key, V value)` to add an entry, `containsKey(K key)` to check whether some key exists, `get(K key)` to retrieve a value, and `for(Map.Entry<K,V> me : map.entrySet()) { ... }` to iterate through a map.

Unlike List, Set, and Queue, Map doesn't extend Collection. However, it is possible to view a map as a Collection instance by calling Map's `keySet()`, `values()`, and `entrySet()` methods, which respectively return a Set of keys, a Collection of values, and a Set of key/value pair entries.

Note The `values()` method returns Collection instead of Set because multiple keys can map to the same value, and `values()` would then return multiple copies of the same value.

The Collection views returned by these methods (recall that a Set is a Collection because Set extends Collection) provide the only means to iterate over a Map. For example, suppose you declare Listing 9-9's Color enum with its three Color constants, RED, GREEN, and BLUE.

Listing 9-9. A Colorful enum

```
enum Color {  
    RED(255, 0, 0),  
    GREEN(0, 255, 0),  
    BLUE(0, 0, 255);  
  
    private int r, g, b;  
  
    private Color(int r, int g, int b) {  
        this.r = r;  
        this.g = g;  
        this.b = b;  
    }  
}
```

```

@Override
public String toString() {
    return "r = " + r + ", g = " + g + ", b = " + b;
}
}

```

The following example declares a map of `String` keys and `Color` values, adds several entries to the map, and iterates over the keys and values:

```

Map<String, Color> colorMap = new HashMap<>();
    // an example of a Map implementation
colorMap.put("red", Color.RED);
colorMap.put("blue", Color.BLUE);
colorMap.put("green", Color.GREEN);
colorMap.put("RED", Color.RED);
for (String colorKey: colorMap.keySet())
    System.out.println(colorKey);
Collection<Color> colorValues = colorMap.values();
for (Iterator<Color> it = colorValues.iterator(); it.hasNext();)
    System.out.println(it.next());

```

When running this code fragment against a hashmap implementation (discussed later) of `colorMap`, you should observe output similar to the following:

```

red
blue
green
RED
r = 255, g = 0, b = 0
r = 0, g = 0, b = 255
r = 0, g = 255, b = 0
r = 255, g = 0, b = 0

```

The first four output lines identify the map's keys; the second four output lines identify the map's values.

The `entrySet()` method returns a `Set` of `Map.Entry` objects. Each of these objects describes a single entry as a key/value pair and is an instance of a class that implements the `Map.Entry` interface, where `Entry` is a nested interface of `Map`.

The following example shows you how you might iterate over the previous example's map entries:

```
for (Map.Entry<String, Color> colorEntry: colorMap.entrySet())
    System.out.println(colorEntry.getKey() + ":" + colorEntry.getValue());
```

When running this example against the previously mentioned hashmap implementation, you would observe the following output:

```
red: r = 255, g = 0, b = 0
blue: r = 0, g = 0, b = 255
green: r = 0, g = 255, b = 0
RED: r = 255, g = 0, b = 0
```

TreeMap

The TreeMap class provides a map implementation that is based on a red-black tree. As a result, entries are stored in sorted order of their keys. However, accessing these entries is somewhat slower than with the other Map implementations (which are not sorted) because links must be traversed.

Note Check out Wikipedia's "Red-black tree" entry (http://en.wikipedia.org/wiki/Red-black_tree) to learn about red-black trees.

For TreeMap's constructors and additional methods, see the API documentation.

HashMap

The HashMap class provides a map implementation that is based on a hashtable data structure. This implementation supports all Map operations and permits null keys and null values. It makes no guarantees on the order in which entries are stored.

A hashtable internally maps keys to integer values with the help of a *hash function*. Java provides this function in the form of Object's hashCode() method, which classes override to provide appropriate hash codes.

A *hash code* identifies one of the hashtable's array elements, which is known as a *bucket* or *slot*. For some hashtables, the bucket may store the value that is associated with the key. Figure 9-2 illustrates this kind of hashtable.

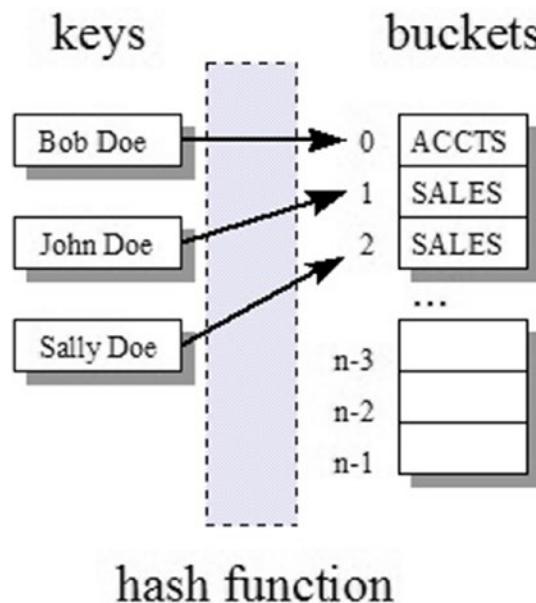
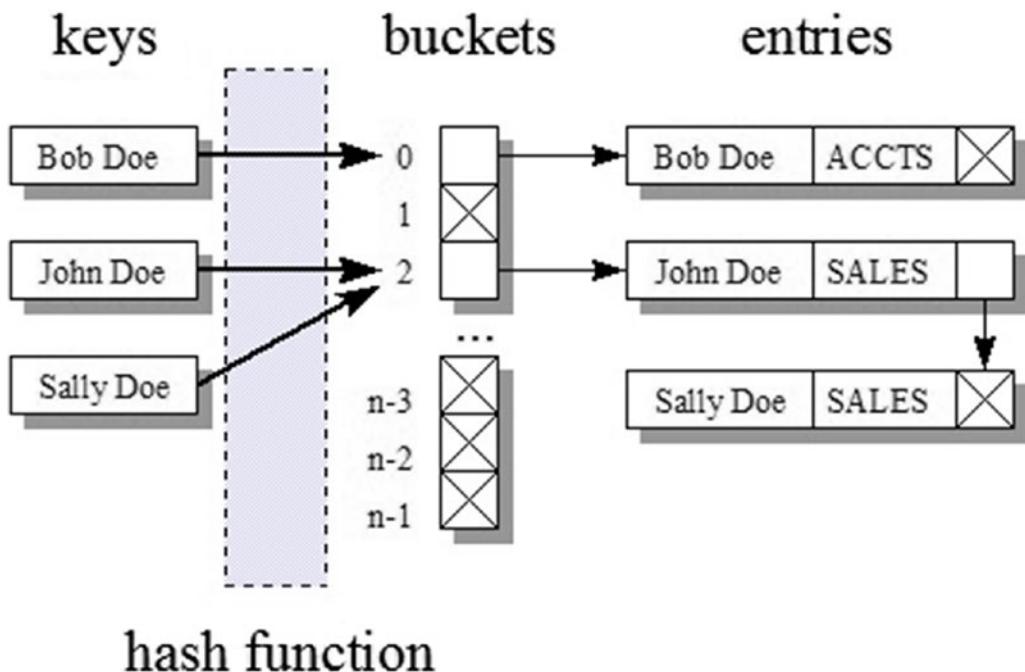


Figure 9-2. A simple hashtable maps keys to buckets that store values associated with those keys

The hash function hashes Bob Doe to 0, which identifies the first bucket. This bucket contains ACCTS, which is Bob Doe's employee type. The hash function also hashes John Doe and Sally Doe to 1 and 2 (respectively) whose buckets contain SALES.

A perfect hash function hashes each key to a unique integer value. However, this ideal is very difficult to meet. In practice, some keys will hash to the same integer value. This nonunique mapping is referred to as a *collision*.

To address collisions, most hashtables associate a linked list of entries with a bucket. Instead of containing a value, the bucket contains the address of the first node in the linked list, and each node contains one of the colliding entries. See Figure 9-3.



hash function

Figure 9-3. A complex hashtable maps keys to buckets that store references to linked lists whose node values are hashed from the same keys

When storing a value in a hashtable, the hashtable uses the hash function to hash the key to its hash code and then searches the appropriate linked list to see if an entry with a matching key exists. If there is an entry, its value is updated with the new value. Otherwise, a new node is created, populated with the key and value, and appended to the list.

When retrieving a value from a hashtable, the hashtable uses the hash function to hash the key to its hash code and then searches the appropriate linked list to see if an entry with a matching key exists. If there is an entry, its value is returned. Otherwise, the hashtable may return a special value to indicate that there is no entry, or it might throw an exception.

The number of buckets is known as the hashtable's *capacity*. The ratio of the number of stored entries divided by the number of buckets is known as the hashtable's *load factor*. Choosing the right load factor is important for balancing performance with memory use.

- As the load factor approaches 1, the probability of collisions and the cost of handling them (by searching lengthy linked lists) increase.
- As the load factor approaches 0, the hashtable's size in terms of number of buckets increases with little improvement in search cost.
- For many hashtables, a load factor of 0.75 is close to optimal. This value is the default for HashMap's hashtable implementation.

For the constructors and all methods of HashMap, consult the API documentation.

[Listing 9-10](#) demonstrates a hashmap.

Listing 9-10. Using a Hashmap to Count Command-Line Arguments

```
import java.util.HashMap;
import java.util.Map;

public class HashMapDemo {
    public static void main(String[] args) {
        Map<String, Integer> argMap = new HashMap<>();
        for (String arg: args) {
            argMap.merge(arg, 1, Integer::sum);
        }
        System.out.println(argMap);
        System.out.println("Number of distinct arguments = " + argMap.size());
    }
}
```

HashMapDemo creates a hashmap of String keys and Integer values. Each key is one of the command-line arguments passed to this application, and its value is the number of occurrences of that argument on the command line. The `merge()` function uses a functional interface to either enter a 1 value if not yet present in the map, or increase by 1 otherwise (we talk about functional programming in Chapter 10).

For example, `java HashMapDemo how much wood could a woodchuck chuck if a woodchuck could chuck wood` generates the following output:

```
{wood=2, could=2, how=1, if=1, chuck=2, a=2, woodchuck=2, much=1}
```

Number of distinct arguments = 8
LinkedHashMap is a subclass of HashMap that uses a linked list to store its entries. As a result, LinkedHashMap's iterator returns entries in the order in which they were inserted. For example, if Listing 9-10 had specified `Map<String, Integer> argMap = new LinkedHashMap<>();`, the application's output for `java HashMapDemo` how much wood could a woodchuck chuck if a woodchuck could chuck wood would have been {how=1, much=1, wood=2, could=2, a=2, woodchuck=2, chuck=2, if=1} followed by number of distinct arguments = 8.

Because the String class overrides `equals()` and `hashCode()`, Listing 9-10 can use String objects as keys in a hashmap. When you create a class whose instances are to be used as keys, you must ensure that you override both methods.

IdentityHashMap

The IdentityHashMap class provides a Map implementation that uses reference equality (`==`) instead of object equality (`equals()`) when comparing keys and values. This is an intentional violation of Map's general contract, which mandates the use of `equals()` when comparing elements.

IdentityHashMap obtains hash codes via System's `int identityHashCode(Object x)` class method instead of via each key's `hashCode()` method. `identityHashCode()` returns the same hash code for `x` as returned by `Object's hashCode()` method, whether or not `x`'s class overrides `hashCode()`. The hash code for the null reference is zero.

These characteristics give IdentityHashMap a performance advantage over other Map implementations. Also, IdentityHashMap supports *mutable keys* (objects used as keys and whose hash codes change when their field values change while in the map).

Note IdentityHashMap's documentation states that "a typical use of this class is topology-preserving object graph transformations, such as serialization or deep copying." (We discuss serialization in Chapter 12.) It also states that "another typical use of this class is to maintain proxy objects." Also, Stack Overflow's "Use Cases for IdentityHashMap" topic (<http://stackoverflow.com/questions/838528/use-cases-for-identity-hashmap>) mentions that it is much faster to use IdentityHashMap than HashMap when the keys are `java.lang.Class` objects.

WeakHashMap

The `WeakHashMap` class provides a `Map` implementation that's based on weakly reachable keys. Each key object is stored indirectly as the referent of a weak reference; and an entry is automatically removed from the map after the garbage collector clears all weak references to the entry's key.

Practically this means: if a key from this kind of map nowhere has a reference pointing to it from the whole application, and the same holds for the value, the map entry is subject to garbage collection and could be automatically removed from the garbage collector. This implies that for ordinary maps the existence of a key inside the map already prevents the garbage collector from removing it.

Note Use `WeakHashMap` with caution. Usually mixing application design with unpredictable garbage collector activities could introduce some unnecessary complexity.

EnumMap

The `EnumMap` class provides a `Map` implementation whose keys are the members of the same enum. Null keys are not permitted; any attempt to store a null key results in a thrown `NullPointerException`. Because an enum map is represented internally as an array, an enum map approaches an array in terms of performance.

Exploring Sorted Maps

`TreeMap` is an example of a *sorted map*, which is a map that maintains its entries in ascending order, sorted according to the keys' natural ordering or according to a comparator that is supplied when the sorted map is created. Sorted maps are described by the `SortedMap` interface.

`SortedMap` (whose generic type is `SortedMap<K, V>`) extends `Map`. With two exceptions, the methods it inherits from `Map` behave identically on sorted maps as on other maps:

- The `Iterator` instance returned by the `iterator()` method on any of the sorted map's `Collection` views traverses the collections in order.
- The arrays returned by the `Collection` views' `toArray()` methods contain the keys, values, or entries in order.

Note Although not guaranteed, the `toString()` methods of the `Collection` views of `SortedMap` implementations in the Collections Framework (such as `TreeMap`) return a string containing all of the view's elements in order.

Exploring Navigable Maps

`TreeMap` is an example of a *navigable map*, which is a sorted map that can be iterated over in descending order as well as ascending order and which can report closest matches for given search targets. Navigable maps are described by the `NavigableMap` interface, whose generic type is `NavigableMap<K,V>`, which extends `SortedMap`.

Exploring the Arrays and Collections Utility APIs

The Collections Framework would be incomplete without its `Arrays` and `Collections` utility classes. Each class supplies various class methods that implement useful algorithms in the contexts of collections and arrays.

The following is a sampling of the `Arrays` class's array-oriented utility methods:

- `static <T> List<T> asList(T... a)` returns a fixed-size list backed by array `a`. (Changes to the returned list "write through" to the array.) For example, `List<String> birds = Arrays.asList("Robin", "Oriole", "Bluejay");` converts the three-element array of `Strings` (recall that a variable sequence of arguments is implemented as an array) to a `List` whose reference is assigned to `birds`.

- `static int binarySearch(int[] a, int key)` searches array `a` for entry `key` using the binary search algorithm (explained following this list). The array must be sorted before calling this method; otherwise, the results are undefined. This method returns the index of the search `key`, if it is contained in the array; otherwise, `(-insertion point) - 1` is returned. The insertion point is the point at which `key` would be inserted into the array (the index of the first element greater than `key`, or `a.length` if all elements in the array are less than `key`) and guarantees that the return value will be greater than or equal to 0 if and only if `key` is found. For example, `Arrays.binarySearch(new String[] {"Robin", "Oriole", "Bluejay"}, "Oriole")` returns 1, "Oriole"'s index.
- `static void fill(char[] a, char ch)` stores `ch` in each element of the specified character array. For example, `Arrays.fill(screen[i], ' ')`; fills the `i`th row of a 2D screen array with spaces.
- `static void sort(long[] a)` sorts the elements in the long integer array `a` into ascending numerical order, for example, `long lArray = new long[] { 20000L, 89L, 66L, 33L}; Arrays.sort(lArray);`.
- `static <T> void sort(T[] a, Comparator<? super T> c)` sorts the elements in array `a` using comparator `c` to order them. For example, when given `Comparator<String> cmp = new Comparator<String>() { @Override public int compare(String e1, String e2) { return e2.compareTo(e1); } };` `String[] innerPlanets = { "Mercury", "Venus", "Earth", "Mars" };` `Arrays.sort(innerPlanets, cmp);` uses `cmp` to help in sorting `innerPlanets` into descending order of its elements: Venus, Mercury, Mars, Earth is the result.

There are two common algorithms for searching an array for a specific element. *Linear search* searches the array element by element from index 0 to the index of the searched-for element or the end of the array. On average, half of the elements must be searched; larger arrays take longer to search. However, the arrays don't need to be sorted.

In contrast, *binary search* searches ordered array `a`'s `n` items for element `e` in a much faster amount of time.

The following is a sampling of the `Collections` class's collection-oriented class methods:

- `static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> c)` returns the minimum element of collection `c` according to the natural ordering of its elements. For example, `System.out.println(Collections.min(Arrays.asList(10, 3, 18, 25)))`; outputs `3`. All of `c`'s elements must implement the `Comparable` interface. Furthermore, all elements must be mutually comparable. This method throws `NoSuchElementException` when `c` is empty.
- `static void reverse(List<?> l)` reverses the order of list `l`'s elements. For example, `List<String> birds = Arrays.asList("Robin", "Oriole", "Bluejay"); Collections.reverse(birds); System.out.println(birds);` results in `[Bluejay, Oriole, Robin]` as the output.
- `static <T> List<T> singletonList(T o)` returns an immutable list containing only object `o`. For example, `list.removeAll(Collections.singletonList(null))`; removes all null elements from list.
- `static <T> Set<T> synchronizedSet(Set<T> s)` returns a synchronized (thread-safe) set backed by the specified set `s`, for example, `Set<String> ss = Collections.synchronizedSet(new HashSet<String>());`. To guarantee serial access, it's critical that all access to the backing set (`s`) is accomplished through the returned set.
- `static <K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m)` returns an unmodifiable view of map `m`, for example, `Map<String, Integer> msi = Collections.unmodifiableMap(new HashMap<String, Integer>());`. Query operations on the returned map "read through" to the specified map; and attempts to modify the returned map, whether direct or via its collection views, result in an `UnsupportedOperationException`.

Note For performance reasons, collection implementations are unsynchronized—unsynchronized collections have better performance than synchronized collections. To use a collection in a multithreaded context, however, you need to obtain a synchronized version of that collection. You obtain that version by calling a synchronized-prefixed method such as `synchronizedSet()`.

You might be wondering about the purpose for the various “empty” class methods in the `Collections` class. For example, `static final <T> List<T> emptyList()` returns an immutable empty list, as in `List<String> ls = Collections.emptyList();`. These methods are present because they offer a useful alternative to returning null (and avoiding potential `NullPointerExceptions`) in certain contexts.

Exploring the Legacy Collection APIs

Java 1.2 introduced the Collections Framework. Before the framework’s inclusion in Java, developers had two choices where collections were concerned: create their own frameworks, or use the `Vector`, `Enumeration`, `Stack`, `Dictionary`, `Hashtable`, `Properties`, and `BitSet` types, which were introduced by Java 1.0.

`Vector` is a concrete class that describes a growable array, much like `ArrayList`. Unlike an `ArrayList` instance, a `Vector` instance is synchronized. `Vector` has been generified and also retrofitted to support the Collections Framework, which makes statements such as `List<String> list = new Vector<String>();` legal. You may need to perform similar assignments when working with legacy code that depends on `Vector`.

The Collections Framework provides `Iterator` for iterating over a collection’s elements. In contrast, `Vector`’s `elements()` method returns an instance of a class that implements the `Enumeration` interface for *enumerating* (iterating over and returning) a `Vector` instance’s elements via `Enumeration`’s `hasMoreElements()` and `nextElement()` methods. Consider the following example:

```
Enumeration e = vector.elements();
while (e.hasMoreElements())
    System.out.println(e.nextElement());
```

`Vector` is subclassed by the concrete `Stack` class, which represents a LIFO data structure. `Stack` provides an `E push(E item)` method for pushing an object onto the stack, an `E pop()` method for popping an item off the top of the stack, and a few other methods, such as `boolean empty()`, for determining whether or not the stack is empty.

`Stack` is a good example of bad API design. By inheriting from `Vector`, it's possible to call `Vector`'s `void add(int index, E element)` method to add an element anywhere you wish and violate a `Stack` instance's integrity. In hindsight, `Stack` should have used composition in its design: use a `Vector` instance to store a `Stack` instance's elements.

`Dictionary` is an abstract superclass for subclasses that map keys to values. The concrete `Hashtable` class is `Dictionary`'s only subclass. As with `Vector`, `HashTable` instances are synchronized, `HashTable` has been generified, and `HashTable` has been retrofitted to support the Collections Framework.

`Hashtable` is subclassed by `Properties`, a concrete class representing a persistent set of *properties* (String-based key/value pairs that identify application settings). `Properties` provides `Object setProperty(String key, String value)` for storing a property and `String getProperty(String key)` for returning a property's value.

Note Applications use properties for various purposes. For example, if your application has a graphical user interface, you could store the screen location and size of its main window in a file via a `Properties` object so that the application can restore the window's location and size when it next runs.

`Properties` is another good example of bad API design. By inheriting from `Hashtable`, you can call `Hashtable`'s `V put(K key, V value)` method to store an entry with a non-String key and/or a non-String value. In hindsight, `Properties` should have leveraged composition: store a `Properties` instance's elements in a `Hashtable` instance.

Finally, `BitSet` is a concrete class that describes a variable-length set of bits. This class's ability to represent bitsets of arbitrary length contrasts with the previously described integer-based, fixed-length bitset that is limited to a maximum number of members: 32 members for an `int`-based bitset, or 64 members for a `long`-based bitset.

The Collections Framework has made Vector, Enumeration, Stack, Dictionary, and Hashtable obsolete. These types continue to be part of the standard class library to support legacy code. Also, the Preferences API has made Properties largely obsolete. Because BitSet is still relevant, this class continues to be included (as recently as Java 12).

EXERCISES

The following exercises are designed to test your understanding of Chapter 9's content:

1. What is a collection?
2. What is the Collections Framework?
3. The Collections Framework largely consists of what components?
4. Define comparable.
5. When would you have a class implement the Comparable interface?
6. What is a comparator and what is its purpose?
7. True or false: A collection uses a comparator to define the natural ordering of its elements.
8. What does the Iterable interface describe?
9. What does the Collection interface represent?
10. Identify a situation where Collection's add() method would throw an instance of the UnsupportedOperationException class.
11. What is the purpose of the enhanced for loop statement?
12. How is the enhanced for loop statement expressed?
13. True or false: The enhanced for loop works with arrays.
14. Define autoboxing.
15. Define unboxing.
16. What is a list?
17. What does a ListIterator instance use to navigate through a list?
18. What is a view?

CHAPTER 9 EXPLORING THE COLLECTIONS FRAMEWORK

19. Why would you use the `subList()` method?
20. What does the `ArrayList` class provide?
21. What does the `LinkedList` class provide?
22. True or false: `ArrayList` provides faster element insertions and deletions than `LinkedList`.
23. What is a set?
24. What does the `TreeSet` class provide?
25. What does the `HashSet` class provide?
26. True or false: To avoid duplicate elements in a hashset, your own classes must correctly override `equals()` and `hashCode()`.
27. What is the difference between `HashSet` and `LinkedHashSet`?
28. What does the `EnumSet` class provide?
29. Define sorted set.
30. What is a navigable set?
31. True or false: `HashSet` is an example of a sorted set.
32. Why would a sorted set's `add()` method throw `ClassCastException` when you attempt to add an element to the sorted set?
33. What is a queue?
34. True or false: Queue's `element()` method throws `NoSuchElementException` when it is called on an empty queue.
35. What does the `PriorityQueue` class provide?
36. What is a map?
37. What does the `TreeMap` class provide?
38. What does the `HashMap` class provide?
39. What does a hashtable use to map keys to integer values?
40. Continuing from the previous question, what are the resulting integer values called and what do they accomplish?

41. What is a hashtable's capacity?
42. What is a hashtable's load factor?
43. What is the difference between `HashMap` and `LinkedHashMap`?
44. What does the `IdentityHashMap` class provide?
45. What does the `EnumMap` class provide?
46. Define sorted map.
47. What is a navigable map?
48. True or false: `TreeMap` is an example of a sorted map.
49. What is the purpose of the `Arrays` class's static `<T> List<T> asList(T... array)` method?
50. True or false: Binary search is slower than linear search.
51. Which Collections method would you use to return a synchronized variation of a `HashSet`?
52. Identify the seven legacy collections-oriented types.
53. As an example of array list usefulness, create a `JavaQuiz` application that presents a multiple choice-based quiz on Java features. The `JavaQuiz` class's `main()` method first populates the array list with the entries in a `QuizEntry` array (such as `new QuizEntry("What was Java's original name?", new String[] { "Oak", "Duke", "J", "None of the above" }, 'A')`). Each entry consists of a question, four possible answers, and the letter (A, B, C, or D) of the correct answer. `main()` then uses the array list's `iterator()` method to return an `Iterator` instance and this instance's `hasNext()` and `next()` methods to iterate over the list. Each of the iterations outputs the question and four possible answers and then prompts the user to enter the correct choice. After the user enters A, B, C, or D (via `System.in.read()`), `main()` outputs a message stating whether or not the user made the correct choice.
54. Why is `(int) (f ^ (f >> 32))` used instead of `(int) (f ^ (f >> 32))` in the hash code generation algorithm?

55. Collections provides the static int frequency(Collection<?> c, Object o) method to return the number of collection c elements that are equal to o. Create a FrequencyDemo application that reads its command-line arguments and stores all arguments except for the last argument in a list and then calls frequency() with the list and last command-line argument as this method's arguments. It then outputs this method's return value (the number of occurrences of the last command-line argument in the previous command-line arguments). For example, java FrequencyDemo should output Number of occurrences of null = 0, and java FrequencyDemo how much wood could a woodchuck chuck if a woodchuck could chuck wood wood should output Number of occurrences of wood = 2.
-

Summary

A collection is a group of objects that are stored in an instance of a class designed for this purpose. To save you from having to create your own collections classes, Java provides the Collections Framework for representing and manipulating collections.

The Collections Framework largely consists of core interfaces, implementation classes, and the Arrays and Collections utility classes. The core interfaces make it possible to manipulate collections independently of their implementations.

Core interfaces include Iterable, Collection, List, Set, SortedSet, NavigableSet, Queue, Deque, Map, SortedMap, and NavigableMap. Collection extends Iterable; List, Set, and Queue each extend Collection; SortedSet extends Set; NavigableSet extends SortedSet; Deque extends Queue; SortedMap extends Map; and NavigableMap extends SortedMap.

Implementation classes include ArrayList, LinkedList, TreeSet, HashSet, LinkedHashSet, EnumSet, PriorityQueue, ArrayDeque, TreeMap, HashMap, LinkedHashMap, IdentityHashMap, WeakHashMap, and EnumMap. The name of each concrete class ends in a core interface name, identifying the core interface on which it is based.

The Collections Framework would not be complete without its Arrays and Collections utility classes. Each class supplies various class methods that implement useful algorithms in the contexts of arrays and collections. For example, Arrays lets you efficiently search and sort arrays, and Collections lets you obtain synchronized and unmodifiable collections.

Before Java 1.2's introduction of the Collections Framework, developers could create their own frameworks or use the `Vector`, `Enumeration`, `Stack`, `Dictionary`, `Hashtable`, `Properties`, and `BitSet` types, which were introduced by Java 1.0.

The Collections Framework has made `Vector`, `Enumeration`, `Stack`, `Dictionary`, and `Hashtable` obsolete. Also, the Preferences API has made `Properties` largely obsolete. Because `BitSet` is still relevant, this class continues to be included.

In Chapter 10 we explore functional programming for Java.

CHAPTER 10

Functional Programming

Functional programming is about promoting functions to first-class citizens in a computer language. This means apart from primitives and objects, reference-typed fields can also directly point to functions, and function arguments can be functions themselves. In addition, a language may also allow for special functional constructs like unnamed lambda calculus functions.

Consider, for example, the following code, which takes a list of strings and adds a “*” to all strings to create a new list:

```
List<String> l = Arrays.asList( "Hello", "World" );
List<String> newList = new ArrayList<>();
for( String s : l )
    newList.add( s + "*" );
```

The functional view of this task reads instead: take one thing (the original list), apply a function (add “*” to each value), and give back the result as a new list. So we could make the code more elegant and self-explanatory if we had something like:

```
List<String> l = ...;
List<String> newList = l.applyToEach( add-*-to-value );
```

And in fact, the new functional features of Java 8 allow for this kind of notation:

```
List<String> l = ...;
List<String> newList = l.stream().map( v -> v + "*" ).collect(Collectors.
toList());
```

Here the `stream()` and `collect()` methods serve as interfaces to the streaming API (also since Java 8). The first creates a stream from a collection, and the second gathers the stream elements to build a new collection. The algorithm thus boils down to the `v -> v + "*"` lambda expression inside the `map()` function.

Functions and Operators

The Java package `java.util.function` contains more than 40 interfaces especially useful for functional programming. The list includes the following:

- Functions with one or two arguments. This implies specialized variants for native data types (`integer`, `long`, `double`, ...), Boolean checks (for filters), and operators (functions with the same type for inputs and output).
- Consumers, which are functions with input but no output.
- Suppliers, which are functions with no input but an output.

The API documentation of this package lists all the interfaces and gives you a detailed description of each of them.

Heavily coupled with the functional interfaces are the interfaces and classes from the `java.util.stream` package, because this is where the data streams are described which enter and exit a pipeline of functional calculations. While it is possible to use functions without streams, in the majority of cases, you use functions in conjunction with streams.

Note In other books you will find a chapter about functional programming and another chapter for the streaming API. We however consider them being a unit, because functions without streams and streams without functions are of very limited use.

Lambda Calculus

Before we continue talking about streams, we investigate a language feature called lambda calculus. Lambdas are function literals at places where functions are expected. So you can use them on the right-hand side of a function assignment as in `Function<String, Integer> f = ...`, or as a function parameter in a method call as in `list.map(...)`.

A lambda expression reads as follows: `(parameter-list) -> body`. The parameter list is a comma-separated list of function parameters, as in `(String s, Integer i)`. Parameter types can be omitted if they can be deduced by the compiler, and the

parentheses can be omitted if there is just one parameter. No parameter always reads `()`. The body is either a single expression, a method call, or a “`{}`”-delimited block with a return statement. Listing 10-1 shows some examples.

Listing 10-1. Lambda Function Examples

```
Function<Integer, String> f1 = (Integer i) -> "_" + i + "_";
Function<Integer, String> f2 = (i) -> "_" + i + "_";
Function<Integer, String> f3 = i -> "_" + i + "_";
Function<Integer, String> f4 = i -> { return "_" + i + "_"; };
BiFunction<String, String, Integer> bf1 = (str1, str2) -> (str1+str2).
length();
Supplier<String> sp1 = () -> "Hello";
Consumer<String> c1 = s -> { System.out.println(s); };
Consumer<String> c2 = s -> System.out.println(s);
Stream<Integer> strm1 = Arrays.asList(1,2,3,4).stream().filter( i -> i < 2 );
```

Entering a Stream

Starting a stream pipeline from a collection is very easy. All collections, that is, sets, lists, and queues, provide a `stream()` method to create a stream from the collection.

Note Streams are conceptionally similar to iterators. However, a stream consumes the incoming data—you cannot rewind a stream and instead have to re-create a stream if you want to reset a looping over it.

Streams can also be generated using static stream generator methods, and you can use a Supplier or an UnaryOperator object for that aim. Listing 10-2 shows some ways to generate a stream.

Listing 10-2. Generating Streams

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Supplier;
import java.util.function.UnaryOperator;
```

CHAPTER 10 FUNCTIONAL PROGRAMMING

```
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class A {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Orange", "Apple", "Banana", "Car");

        Stream<String> s1 = list.stream();
        s1.forEach(System.out::println);

        Stream<String> s2 = Stream.of("Orange", "Apple", "Banana", "Car");
        s2.forEach(System.out::println);

        // Beware, this is not a stream of ints, but a stream with
        // an int array as its only element!
        Stream<int[]> s3 = Stream.of(new int[] { 3, 7, 23, 59, -2 });
        s3.forEach(System.out::println);

        // This is a stream of ints
        IntStream s4 = IntStream.of(new int[] { 3, 7, 23, 59, -2 });
        s4.forEach(System.out::println);

        // The same, as a stream of integer objects
        Stream<Integer> s5 = IntStream.of(new int[] { 3, 7, 23, 59, -2 })
            .boxed();
        s5.forEach(System.out::println);

        Supplier<Double> rndDouble = () -> Math.random();
        Stream<Double> s6 = Stream.generate(rndDouble);
        // This is an infinite stream! For diagnostic output, we have to limit it
        s6.limit(4).forEach(System.out::println);

        UnaryOperator<String> addX = (s) -> s + "x";
        Stream<String> s7 = Stream.iterate("0", addX);
        s7.limit(4).forEach(System.out::println);
    }
}
```

Each stream of that listing receives a `forEach()` as a terminal operation. Here we use it for printing the contents of a stream. The `forEach()` needs a `Consumer` as an argument—the `System.out::println` is a way to address a method as a functional object.

In the listing you can also see that we use an `IntStream`, which is a specialized stream for int values. This stream exists because it is somewhat cheaper from a computational point of view if we can stream over int values and not int objects. Apart from `IntStream`, there are also specialized streams for long and double values: `LongStream` and `DoubleStream`.

Mapping

While dealing with collections, you frequently encounter the requirement to transform the data in one or the other way. Streams are very good at that by providing various `map*`() and `flatMap*`() methods. The `map*`() variants take one element at a time for the transformation, and the `flatMap*`() variants may be used to map (inner) streams and provide a way to perform a stacked mapping. The API documentation of the `Stream` class (<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>) gives you more details, and Listing 10-3 exposes some usage scenarios.

Listing 10-3. Mapping in Streams

```
import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.Set;
import java.util.stream.Stream;

...
class Record {
    int id;
    String name;
    Collection<String> items;
    Record(int id, String name, String... items) {
        this.id = id; this.name = name;
        this.items = Arrays.asList(items);
    }
}
```

```

List<Record> records = Arrays.asList(
    new Record(1, "rec01", "Blue", "Red", "Yellow"),
    new Record(2, "rec02", "Whale", "Cat"),
    new Record(3, "rec03", "John", "Linda", "Angie", "Cat")
);

List<Integer> ids = records.stream().
    map( r -> r.id ).
    collect(Collectors.toList());
List<Integer> nameLens = records.stream().
    map( r -> r.name.length() ).
    collect(Collectors.toList());
List<String> allItems = records.stream().
    flatMap( r -> r.items.stream() ).
    collect(Collectors.toList());
Set<String> distinctItems = records.stream().
    flatMap( r -> r.items.stream() ).
    collect(Collectors.toSet());
...

```

The first mapping returns all IDs, the second mapping retrieves all name lengths, the third mapping collects all record items, and the fourth mapping collects all distinct record items.

Filtering

The `filter()` method of class `Stream` may be used to filter the elements of a stream. The argument of the `filter()` method is a *Predicate*, which is a function returning a Boolean:

```
... .stream().filter( elem -> some boolean ) ...
```

To, for example, extend the fore-last mapping from Listing 10-3 to return only items unequal “Cat”, you’d write

```
List<String> allItems = records.stream().
    flatMap( r -> r.items.stream() ).
    filter( itm -> !(itm.equals("Cat")) ).
    collect(Collectors.toList());
```

This short listing also reveals one of the powers of streams: you can chain mappings, filters, and other stream operations in stream expressions at will.

Terminating a Stream

Streams perform computations lazily. This means that a statement like

```
List<String> allItems = records.stream().
    flatMap( r -> r.items.stream() ).
    filter( itm -> !(itm.equals("Cat")) );
```

doesn't do anything at all apart from providing another stream. For a stream to actually perform any calculation or transformation, you have to add a terminal operation. So far we used `forEach()` and `collect()` as terminal operations, but there are more, like `findFirst()`, `findAny()`, `max()`, `min()`, `count()`, and others.

Performing Actions on Each Element

The `forEach()` method is a terminating operation on a stream. You frequently use it to print out the contents of a stream or perform other operations which cannot be performed in a stream scope (like writing to a database). The argument of the `forEach()` method is a *Consumer* of the stream element type. For example:

```
Stream.of(1, 2, 6, 3, -4).
    forEach( theInt -> System.out.println(theInt) );
```

or

```
Stream.of(1, 2, 6, 3, -4).
    forEach( System.out::println );
```

The latter is possible because `println()` is static and can handle a string.

Limiting, Skipping, Sorting, and Distinct

Streams can be limited, which might be important especially for infinite streams. Just insert

```
... .limit(10). ...
```

to limit the number of stream items to, for example, 10. To skip a number of elements from the beginning of a stream, use the `skip()` function.

In case you need to sort a stream, use `sorted()` to apply a natural element ordering, or `sorted(Comparator)` for a sorting controlled by a `Comparator` parameter. Note that for sorting internally the stream gets terminated, because for a sorting to work we need to know all the elements first.

The `distinct()` stream method makes sure elements in the stream won't repeat. Again, this is internally terminating a stream, because for the distinction-check we also first need to know all the elements.

Ranges

The class `IntStream` provides static methods `range(int startInclusive, int endExclusive)` and `rangeClosed(int startInclusive, int endInclusive)` which you can use to create a stream of consecutive integers. Class `LongStream` has corresponding creator methods for a long values stream.

```
IntStream.range(0, 10).forEach(System.out.println); // -> 0,...,9
```

Reducing

A reduction operation iterates over all stream elements and combines elements using a `BinaryOperator` or a `BiFunction`. To, for example, gather invoices and build a master invoice, you'd write

```
public class Invoice {
    BigDecimal amount = BigDecimal.ZERO;
    // ... more fields
    public Invoice() { }
```

```

public Invoice(BigDecimal amount) {
    this.amount = amount;
}
}

List<Invoice> invl = Arrays.asList(
    new Invoice(new BigDecimal("1.99")),
    new Invoice(new BigDecimal("7.00")),
    new Invoice(new BigDecimal("13.98"))
);

Invoice master = invl.stream().
    reduce(new Invoice(),
        (inv1,inv2) ->
            new Invoice(inv1.amount.add(inv2.amount)) );

```

where instead of `new Invoice()` you can also write `Invoice::new`.

A characteristic of reduction is the result being of the same type as the elements. For more details and other variants of the `reduce()` function, see the API documentation.

Collecting

Collecting is similar to reduction, but the result type of the collection operation is free. For Java 8 (and Android), there are two collection operations. The first one reads

```
R collect(Collector<? super T,A,R> collector)
```

and you can use a Collector to handle the elements supplied by the stream. This operation gets more powerful by various predefined collectors you find in the `Collectors` class. We already used the `Collectors.toList()` collector to gather the stream elements in a list, but there are others to gather the elements resulting in other collection types and maps, and more. One you'd maybe use often is a String joining collector:

```
Arrays.asList("Orange", "Apple", "Banana", "Car").stream().
    collect(Collectors.joining(","));
```

The second `collect()` operation has the signature

```
R collect(Supplier<R> supplier,
BiConsumer<R,? super T> accumulator,
BiConsumer<R,R> combiner)
```

You can use it for your own sophisticated collection needs. To, for example, get a statistics containing the standard deviation from a `DoubleStream`, you can write

```
class MyStats {
    List<Double> l = new ArrayList<>();
    void add(double d) { l.add(d); }
    int count() { return l.size(); }
    double avg() {
        return l.stream().collect(Collectors.summingDouble(d -> d));
    }
    double stddev() {
        double a = avg();
        return Math.sqrt(
            l.stream().collect(Collectors.summingDouble(d -> (d-a) * (d-a)))
            / (l.size()-1) );
    }
}
MyStats stat = DoubleStream.of(4.6, 3.2, 9.6, 27.0).collect(MyStats::new,
    (bas, inj) -> { bas.add(inj); },
    (bas1, bas2) -> { bas2.l.stream().forEach( d -> bas1.add(d) ); });
System.out.println("Count = " + stat.count() +
    ", Average = " + stat.avg() +
    ", StdDev = " + stat.stddev());
```

You can however improve this by constructing your own collector instance:

```
public class MyStats {
    List<Double> l = new ArrayList<>();
    void add(double d) { l.add(d); }
    int count;
    double avg;
    double stddev;
```

```

MyStats finish() {
    count = l.size();
    avg = l.stream().collect(Collectors.summingDouble(d -> d));
    stddev = Math.sqrt(
        l.stream().collect(Collectors.summingDouble(d -> (d-avg) * (d-avg)))
        / (count-1) );
    return this;
}
static MyStats merge(MyStats s1, MyStats s2) {
    s2.l.stream().forEach( d -> s1.add(d) );
    return s1;
}
}

...
Collector<Double,MyStats,MyStats> statsCollector = Collector.of(
    MyStats::new,
    MyStats::add,
    MyStats::merge,
    MyStats::finish
);
MyStats stat = Stream.of(4.6, 3.2, 9.6, 27.0).collect(statsCollector);
System.out.println("Count = " + stat.count + ", Average = " + stat.avg + ", "
StdDev = " + stat.stddev);

```

The first parameter to the `Collector.of()` function supplies a statistics object. The `::new` is a shortcut to directly specify the constructor. The second parameter points to a `Consumer` (a function not returning something) for receiving a stream element. Because the `add()` method exactly does that, we can directly provide it via `::add`. The third argument merges two intermediate collection results—this is necessary if the collection operation gets distributed to several threads. We added this as a static helper method to our statistics class, so we can write `::merge` here. The last parameter finishes the result object. Here we calculate the count, the average, and the standard deviation. Because `finish()` returns the result object, we can directly specify `::finish`.

For more details about `collect()`, `Collector`, and `Collectors`, please see the API documentation.

Methods As Functions

Whenever a function, predicate, operator, supplier, or consumer is requested during some functional computation, you can use appropriately structured class methods as substitutes. This can significantly improve the readability of your application. The rules are as follows:

- A static method with void return and a single parameter (Type1) can serve as a substitute for `Consumer<Type1>`, `LongConsumer`, `IntConsumer`, or `DoubleConsumer` (depending on the type). For example, `DoubleConsumer dc = SomeClass::method` (where `method` is `static void method(double d) { ... }`).
- A static method with void return and two parameters (Type1, Type2) can serve as a substitute for `BiConsumer<Type1, Type2>`. For example, `BiConsumer<Double, Integer> bc = SomeClass::method` (where `method` is `static void method(double d, int i) { ... }`).
- A static method with return type `RetType` and no method parameter can serve as a `Supplier`, `BooleanSupplier`, `IntSupplier`, `LongSupplier`, or `DoubleSupplier` (depending on the type) for type `RetType`. For example, `Supplier<String> su = SomeClass::method` (where `method` is `static String method() { ...; return someString; }`).
- Generally, a static method with zero, one, or two parameters and returning or not returning a value can serve as a functional interface, provided the parameter and return types match. You write `SomeClass::someMethod` to use the static method as a functional interface. For example, `Function<Long, String> f = SomeClass::method`, where the method has signature `static String method(long l)`.
- Any nonstatic method can serve as a functional interface, provided the first type parameter of the functional interface matches the class where the method resides in. You write `SomeClass::someMethod`

to use the method as a functional interface. For example,
`Function<SomeClass, Integer> f = SomeClass::method`, where
method is inside SomeClass and has signature `int method() { ...;`
`return someInt; }.`

- To refer to a constructor of some class A as a functional interface, use `A::new`.

A prominent example is the `System.out::println` function you can use in `forEach()` to print out the contents of a stream:

```
Stream.forEach(System.out::println)
```

Single-Method Interfaces

Any functional interface automatically can serve as an object implementing some interface, provided the interface has just one method.

A good example for this kind of equivalence is the `Runnable` interface. The power and boost in expressiveness here comes from using a lambda expression as a `Runnable`. Remember the `Thread` class, which has a constructor `Thread(Runnable r)`. The usual code for threads

```
Thread thr = new Thread(new Runnable() {
    public void run() {
        ... do some work
    }
});
thr.start();
```

can thus be rewritten to the more expressive version:

```
Thread thr = new Thread( () -> {
    ... do some work
});
thr.start();
```

Streams and Parallelization

Streams were invented with parallelization in mind. Unless some side effects contradict parallelization, you can use method `.parallelStream()` instead of `.stream()` on a collection to switch to multithreaded streams. Or you invoke `.parallel()` on a stream which was generated by some other means to switch to a stream which can be worked at in parallel.

What you need to have in mind though is that parallel execution of streams might introduce some administration overhead, so whether parallelization really speeds up your application needs to be tested with real-world examples.

Protonpack, a Stream Utility Library

Java 8 (and thus Android) streams have a small number of flaws, which can be remedied by the small but nice protonpack library from <https://github.com/poetix/protonpack>.

For example, a Java 8 stream does not have a mean to limit the stream based on some criterion. It is just not possible to write

```
Stream.iterate(0, i -> i+1).limit( i -> someMethod(i) == true )
```

to limit an infinite stream based on some check. With protonpack, you can however write

```
Stream<Integer> s = StreamUtils.takeWhile(Stream.iterate(0, i -> i+1), i -> i <= 10);
```

to limit the originally infinite stream.

Protonpack has a couple of more and useful functions which you can learn about by looking at <https://github.com/poetix/protonpack>.

The Maven coordinates of protonpack are

```
<dependency>
    <groupId>com.codepoetics</groupId>
    <artifactId>protonpack</artifactId>
    <version>1.16</version>
</dependency>
```

EXERCISES

The following exercises are designed to test your understanding of Chapter 10's content:

1. Right or wrong? A class field can be a function.
2. Right or wrong? A stream is a subconcept of functional programming.
3. Right or wrong? Functions are lambda calculus expressions.
4. Define consumer.
5. Define supplier.
6. Create a lambda expression `Function<Double,Double> plusOne = ...` which adds 1.0 to any double.
7. Create a lambda expression `Function<BigDecimal,String> monetary = ...` which creates a monetary string representation like \$37.99 for any `BigDecimal`.
8. Right or wrong? `Stream.of(new int[] { 3, 7 })` and `Stream.of(3, 7)` are the same.
9. Create an infinite stream of long values 100, 101, 102,
10. Create a stream of integer values 0, 1, ..., 99.
11. Create an infinite stream of integer values 1, 4, 9, 16, 25,
12. Make a set 1, 6, 3, 7 and build a stream of integer objects from it.
13. From the set of the previous exercise, build an `IntStream` from it.
14. From the `IntStream` of the last exercise, build a stream of integer objects from it.
15. Build a filter which removes strings of length smaller than 3 from a stream of strings.
16. From each stream of the last exercises, build a list out of it.
17. From the stream 1, 4, 9, 16, ..., limit the output to 10 values and print each stream element to the console.

18. From the stream 1, 6, 7, 3, use a reduction operation to multiply all values ($1 * 6 * 7 * 3$).
 19. Use a collector to create a Deque (ArrayDeque) from a stream 1, 6, 7, 3.
 20. Use a method as function literal (the `Clazz::method` notation) and Stream.`map()` to convert the list ["Hello", "World", "A"] to a list of the corresponding hash values.
 21. Create an infinite stream 1, -1/3, 1/5, -1/7, 1/9, Hint: Class `IntStream` has a method for mapping integers to doubles using some formula.
 22. Limit the stream from the last exercise to 1,000,000 elements. Use a method from `DoubleStream` to sum up all values, and multiply the result by 4. This approximates Pi (Leibniz formula).
 23. Measure the time needed for the Pi calculation from the last exercise (use `System.currentTimeMillis()` for the number of milliseconds since 1970-01-01 00:00:00 UTC).
-

Summary

Functional programming is about promoting functions to first-class citizens in a computer language. This means apart from primitive and object, reference-typed fields can also directly point to functions, and function arguments can be functions themselves. In addition, a language may also allow for special functional constructs like unnamed lambda calculus functions.

The Java package `java.util.function` contains more than 40 interfaces especially useful for functional programming. Heavily coupled with the functional interfaces are the interfaces and classes from the `java.util.stream` package.

Lambdas are function literals at places where functions are expected. So you can use them on the right-hand side of a function assignment as in `Function<String, Integer> f = ...`, or as a function parameter in a method call as in `list.map(...)`. A lambda expression reads as follows: `(parameter-list) -> body`. The parameter list is a comma-separated list of function parameters, as in `(String s, Integer i)`. Parameter types can be omitted if they can be deduced by the compiler, and the parentheses can be omitted if there is just one parameter. No parameter always reads `()`. The body is either a single expression, a method call, or a “`{}`”-delimited block with a return statement.

All collections, that is, sets, lists, and queues, provide a `stream()` method to create a stream from the collection. Apart from `Stream`, there are also specialized streams for long and double values: `IntStream`, `LongStream`, and `DoubleStream`.

Streams have various `map*`() and `flatMap*`() methods for mapping stream elements. The `filter()` method of class `Stream` may be used to filter the elements of a stream. The argument of the `filter()` method is a `Predicate`, which is a function returning a Boolean.

Streams perform computations lazily. You must add a terminating operation like `forEach()` for a stream to actually do some work. Streams can also be limited, elements at the beginning may be skipped, streams can be sorted, and duplicates can be removed.

Both `IntStream` and `LongStream` provide static methods `range(int startInclusive, int endExclusive)` and `rangeClosed(int startInclusive, int endInclusive)` which you can use to create a stream of consecutive integers or long integers.

Streams have `reduce()` and `collect()` methods you can use to aggregate its elements.

Static and member methods of classes can be used as functions, provided the parameters match (for nonstatic methods the first parameter must be the receiver object).

Any functional interface automatically can serve as an object implementing some interface, provided the interface has just one method.

Streams were invented with parallelization in mind. You can use method `.parallelStream()` instead of `.stream()` on a collection to switch to multithreaded streams. Or you invoke `.parallel()` on a stream which was generated by some other means to switch to a stream which can be worked at in parallel.

In the next chapter, we talk about concurrency utilities.

CHAPTER 11

Exploring the Concurrency Utilities

Chapter 7 introduced Java’s Threads API. Significant problems with Threads resulted in the development of the more powerful Concurrency Utilities framework. In this chapter, we take you on a tour of this framework from Android’s perspective.

Introducing the Concurrency Utilities

The low-level Threads API lets you create multithreaded applications that offer better performance and responsiveness over their single-threaded counterparts. However, there are problems:

- Low-level concurrency primitives, such as `synchronized` and `wait()/notify()`, are often hard to use correctly. Incorrect use of these primitives can result in race conditions, thread starvation, deadlock, and other hazards, which can be hard to detect and debug.
- Too much reliance on the `synchronized` primitive can lead to performance issues, which affect an application’s *scalability*. This is a significant problem for highly threaded applications such as web servers.
- Developers often need to use higher-level constructs such as thread pools and semaphores. Because these constructs aren’t included with Java’s low-level threading capabilities, developers have been forced to build their own constructs, which is a time-consuming and error-prone activity.

To address these problems, Java 5 introduced the *Concurrency Utilities*, a powerful and extensible framework of high-performance threading utilities such as thread pools and blocking queues. This framework consists of the following packages:

- `java.util.concurrent` contains utility types that are often used in concurrent programming, for example, executors, thread pools, and concurrent hashmaps.
- `java.util.concurrent.atomic` contains utility classes that support lock-free, thread-safe programming on single variables.
- `java.util.concurrent.locks` contains utility types for locking and waiting on *conditions* (objects that let threads suspend execution [wait] until notified by other threads that some Boolean state may now be true). Locking and waiting via these types provides better performance and is more flexible than doing so via Java's monitor-based synchronization and wait/notification mechanisms.

This framework also introduces a `long nanoTime()` method to the `java.lang.System` class, which lets you access a nanosecond-granularity time source for making relative time measurements.

Note Two terms that are commonly encountered when exploring the Concurrency Utilities framework are parallelism and concurrency. According to Oracle's "Multithreading Guide" (<http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html>), *parallelism* is "a condition that arises when at least two threads are **executing** simultaneously." In contrast, *concurrency* is "a condition that exists when at least two threads are **making progress** [simultaneously or not. It is a] more generalized form of parallelism that can include time-slicing as a form of virtual parallelism."

The concurrency utilities can be classified as executors, synchronizers, concurrent collections, a locking framework, and atomic variables. We explore each category in the following sections.

Exploring Executors

The Threads API lets you execute runnable tasks via expressions such as `new java.lang.Thread(new RunnableTask()).start();`. These expressions tightly couple task submission with the task's execution mechanics (run on the current thread, a new thread, or a thread arbitrarily chosen from a *pool* [group] of threads).

Note A *task* is an object whose class implements the `java.lang.Runnable` interface (a runnable task) or the `java.util.concurrent.Callable` interface (a callable task).

The concurrency-oriented utilities provide executors as a high-level alternative to low-level Threads API expressions for executing runnable tasks. An *executor* is an object whose class directly or indirectly implements the `java.util.concurrent.Executor` interface, which decouples task submission from task-execution mechanics.

Note The executor framework's use of interfaces to decouple task submission from task-execution mechanics is analogous to the Collections Framework's use of core interfaces to decouple lists, sets, queues, and maps from their implementations. Decoupling results in flexible code that's easier to maintain.

`Executor` declares a solitary `void execute(Runnable runnable)` method that executes the runnable task named `runnable` at some point in the future. `execute()` throws `java.lang.NullPointerException` when `runnable` is `null` and `java.util.concurrent.RejectedExecutionException` when it cannot execute `runnable`.

Note `RejectedExecutionException` can be thrown when an executor is shutting down and doesn't want to accept new tasks. Also, this exception can be thrown when the executor doesn't have enough room to store the task (perhaps the executor uses a bounded blocking queue to store tasks and the queue is full; we discuss blocking queues later in this chapter).

The following example presents the Executor equivalent of the aforementioned new Thread(new RunnableTask()).start(); expression:

```
Executor executor = ...; // ... represents some executor creation
executor.execute(new RunnableTask());
```

Although Executor is easy to use, this interface is limited in various ways:

- Executor focuses exclusively on Runnable. Because Runnable's run() method doesn't return a value, there's no convenient way for a runnable task to return a value to its caller.
- Executor doesn't provide a way to track the progress of runnable tasks that are executing, cancel an executing runnable task, or determine when the runnable task finishes execution.
- Executor cannot execute a collection of runnable tasks.
- Executor doesn't provide a way for an application to shut down an executor (much less to shut down an executor properly).

These limitations are addressed by the java.util.concurrent.ExecutorService interface, which extends Executor and whose implementation is typically a thread pool.

The API documentation describes all this interface's methods. There you also learn that the interface handles callable tasks, which are analogous to runnable tasks. Unlike Runnable, whose void run() method cannot throw checked exceptions, Callable<V> declares a V call() method that returns a value and which can throw checked exceptions because call() is declared with a throws Exception clause.

Finally, the interface exhibits the Future interface, which represents the result of an asynchronous computation. The result is known as a *future*, because it typically will not be available until some moment in the future. Future, whose generic type is Future<V>, provides methods for canceling a task, for returning a task's value, and for determining whether or not the task has finished.

Suppose that you intend to write an application whose graphical user interface lets the user enter a word. After the user enters the word, the application presents this word to several online dictionaries and obtains each dictionary's entry. These entries are subsequently displayed to the user.

Because online access can be slow, and because the user interface should remain responsive (perhaps the user might want to end the application), you offload the "obtain

word entries” task to an executor that runs this task on a separate thread. The following example uses ExecutorService, Callable, and Future to accomplish this objective:

```
ExecutorService executor = Executors.newFixedThreadPool(4); // some executor
creation

Future<String[]> taskFuture = executor.submit(
    new Callable<String[]>() {
        @Override
        public String[] call() {
            String[] entries = ...;
            // Access online dictionaries
            // with search word and populate
            // entries with their resulting
            // entries.
            return entries;
        }
    });
// Do stuff.
String entries = taskFuture.get();
```

After obtaining an executor in some manner, the example’s main thread submits a callable task to the executor. The submit() method immediately returns with a reference to a Future object for controlling task execution and accessing results. The main thread ultimately calls this object’s get() method to get these results.

The Executors utility class declares several class methods that return instances of various ExecutorService and ScheduledExecutorService implementations (and other kinds of instances).

For example, static ExecutorService newFixedThreadPool(int nThreads) creates a thread pool that reuses a fixed number of threads operating off of a shared unbounded queue. At most, nThreads threads are actively processing tasks. If additional tasks are submitted when all threads are active, they wait in the queue for an available thread.

Note Thread pools are used to eliminate the overhead from having to create a new thread for each submitted task. Thread creation isn’t cheap, and having to create many threads could severely impact an application’s performance.

You would commonly use executors, runnables, callables, and futures in file and network input/output contexts. Performing a lengthy calculation offers another scenario where you could use these types.

Exploring Synchronizers

The Threads API offers synchronization primitives for synchronizing thread access to critical sections. Because it can be difficult to write synchronized code correctly that's based on these primitives, Concurrency Utilities includes *synchronizers*, classes that facilitate common forms of synchronization.

Four commonly used synchronizers are countdown latches, cyclic barriers, exchangers, and semaphores. We explore each synchronizer in this section.

Countdown Latches

A *countdown latch* causes one or more threads to wait at a “gate” until another thread opens this gate, at which point these other threads can continue. It consists of a count and operations for “causing a thread to wait until the count reaches zero” and “decrementing the count.”

The `java.util.concurrent.CountDownLatch` class implements the countdown latch synchronizer. You initialize a `CountDownLatch` instance to a specific count by invoking this class's `CountDownLatch(int count)` constructor. This constructor throws `IllegalArgumentException` when the value passed to `count` is negative.

Cyclic Barriers

A *cyclic barrier* lets a set of threads wait for each other to reach a common barrier point. The barrier is *cyclic* because it can be reused after the waiting threads are released. This synchronizer is useful in applications involving a fixed-size party of threads that must occasionally wait for each other.

The `java.util.concurrent.CyclicBarrier` class implements the cyclic barrier synchronizer. You initialize a `CyclicBarrier` instance to a specific number of *parties* (threads working toward a common goal) by invoking this class's `CyclicBarrier(int parties)` constructor. This constructor throws `IllegalArgumentException` when the value passed to `parties` is less than 1.

Alternatively, you can invoke the `CyclicBarrier(int parties, Runnable barrierAction)` constructor to initialize a cyclic barrier to a specific number of parties and a `barrierAction` that's executed when the barrier is *tripped*. In other words, when `parties - 1` threads are waiting and one more thread arrives, that thread executes `barrierAction` and then all threads proceed. This runnable is useful for updating shared state before any of the threads continue. This constructor throws `IllegalArgumentException` when the value passed to `parties` is less than 1.

Exchangers

An *exchanger* provides a synchronization point where threads can swap objects. Each thread presents some object on entry to the exchanger's `exchange()` method, matches with a partner thread, and receives its partner's object on return. Exchangers can be useful in applications such as genetic algorithms (see http://en.wikipedia.org/wiki/Genetic_algorithm) and pipeline designs.

The generic `java.util.concurrent.Exchanger<V>` class implements the exchanger synchronizer. You initialize an exchanger by invoking the `Exchanger()` constructor.

Semaphores

A *semaphore* maintains a set of *permits* for restricting the number of threads that can access a limited resource. A thread attempting to acquire a permit when no permits are available blocks until some other thread releases a permit.

Note Semaphores whose current values can be incremented past 1 are known as *counting semaphores*, whereas semaphores whose current values can be only 0 or 1 are known as *binary semaphores* or *mutexes*. In either case, the current value cannot be negative.

The `java.util.concurrent.Semaphore` class implements this synchronizer and conceptualizes a semaphore as an object maintaining a set of *permits*. You initialize a semaphore by invoking the `Semaphore(int permits)` constructor where `permits` specifies the number of available permits. The resulting semaphore's *fairness policy* is set to false (unfair). Alternatively, you can invoke the `Semaphore(int permits, boolean fair)` constructor to also set the semaphore's fairness setting to true (fair).

Exploring the Concurrent Collections

In Chapter 9, we introduced you to the Collections Framework. This framework provides interfaces and classes that are located in the `java.util` package. Interfaces include `List`, `Set`, and `Map`; classes include `ArrayList`, `TreeSet`, and `HashMap`.

`ArrayList`, `TreeSet`, `HashMap`, and other implementation classes are not thread-safe. However, you can make them thread-safe by using the synchronized wrapper methods located in the `java.util.Collections` class. For example, you can pass an `ArrayList` instance to `Collections.synchronizedList()` to obtain a thread-safe variant of `ArrayList`.

Although they're often needed to simplify code in a multithreaded environment, there are a couple of problems with thread-safe collections:

- It's necessary to acquire a lock before iterating over a collection that might be modified by another thread during the iteration. If a lock isn't acquired and the collection is modified, it's highly likely that `java.util.ConcurrentModificationException` will be thrown. This happens because Collections Framework classes return *fail-fast iterators*, which are iterators that throw `ConcurrentModificationException` when collections are modified during iteration. Fail-fast iterators are often inconvenient to concurrent applications.
- Performance suffers when synchronized collections are accessed frequently from multiple threads. This performance problem ultimately impacts an application's *scalability*.

The Concurrency Utilities framework addresses these problems by introducing performant and highly scalable collections-oriented types, which are stored in the `java.util.concurrent` package. Its collections-oriented classes return *weakly consistent iterators*, which are iterators that have the following properties:

- An element that's removed after iteration starts but hasn't yet been returned via the iterator's `next()` method won't be returned.
- An element that's added after iteration starts may or may not be returned.
- No element is returned more than once during the collection's iteration, regardless of changes made to the collection during iteration.

The following list offers a short sample of concurrency-oriented collection types that you'll find in the `java.util.concurrent` package:

- `BlockingQueue` is a subinterface of `java.util.Queue` that also supports blocking operations that wait for the queue to become nonempty before retrieving an element and wait for space to become available in the queue before storing an element. Each of the `ArrayBlockingQueue`, `DelayQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, and `SynchronousQueue` classes implements this interface.
- `ConcurrentMap` is a subinterface of `java.util.Map` that declares additional atomic `putIfAbsent()`, `remove()`, and `replace()` methods. The `ConcurrentHashMap` class (the concurrent equivalent of `java.util.HashMap`), the `ConcurrentNavigableMap` class, and the `ConcurrentSkipListMap` class implement this interface.

Oracle's Javadoc for `BlockingQueue`, `ArrayBlockingQueue`, and other concurrency-oriented collection types identifies these types as part of the Collections Framework.

Exploring the Locking Framework

The `java.util.concurrent.locks` package provides a framework of interfaces and classes for locking and waiting for conditions in a manner that's distinct from built-in synchronization and `java.lang.Object`'s `wait`/`notification` mechanism. The locking framework improves on synchronization and `wait`/`notification` by offering capabilities such as lock polling and timed waits.

The locking framework includes the often-used `Lock`, `ReentrantLock`, `Condition`, `ReadWriteLock`, and `ReentrantReadWriteLock` types, which we discuss in this section.

Lock

The `Lock` interface offers more extensive locking operations than can be obtained via the locks associated with monitors. For example, you can immediately back out of a lock-acquisition attempt when a lock isn't available. This interface declares the following methods:

- `void lock()`: Acquires the lock. When the lock isn't available, the calling thread is forced to wait until it becomes available.
- `void lockInterruptibly()`: Acquires the lock unless the calling thread is interrupted. When the lock isn't available, the calling thread is forced to wait until it becomes available or the thread is interrupted, which results in this method throwing `InterruptedException`.
- `Condition newCondition()`: Returns a new `Condition` instance that's bound to this `Lock` instance. This method throws `java.lang.UnsupportedOperationException` when the `Lock` implementation doesn't support conditions.
- `boolean tryLock()`: Acquires the lock when it's available at the time this method is invoked. The method returns true when the lock is acquired and false when the lock isn't acquired.
- `boolean tryLock(long time, TimeUnit unit)`: Acquires the lock when it's available within the specified waiting time and the calling thread isn't interrupted. When the lock isn't available, the calling thread is forced to wait until it becomes available within the waiting time or the thread is interrupted, which results in this method throwing `InterruptedException`.
- `void unlock()`: Releases the lock.

Acquired locks must be released. In the context of synchronized methods and statements, and the implicit monitor lock associated with every object, all lock acquisition and release occurs in a block-structured manner. When multiple locks are acquired, they're released in the opposite order and all locks are released in the same lexical scope in which they were acquired.

With this increased flexibility comes additional responsibility. The absence of block-structured locking removes the automatic release of locks that occurs with synchronized methods and statements. As a result, you should typically employ the following idiom for lock acquisition and release:

```
Lock l = ...; // ... is a placeholder for code that obtains the lock
l.lock();
try {
```

```
// access the resource protected by this lock
} catch (Exception ex) {
    // ...
} finally {
    l.unlock();
}
```

This idiom ensures that an acquired lock will always be released.

ReentrantLock

`Lock` is, for example, implemented by the `ReentrantLock` class, which describes a reentrant mutual exclusion lock. This lock is associated with a hold count. When a thread holds the lock and reacquires the lock by invoking `lock()`, `lockUninterruptibly()`, or one of the `tryLock()` methods, the hold count is increased by 1. When the thread invokes `unlock()`, the hold count is decremented by 1. The lock is released when this count reaches 0.

For more details, please see the API documentation.

Condition

The `Condition` interface factors out `Object`'s wait and notification methods (`wait()`, `notify()`, and `notifyAll()`). So we have distinct objects which correspond with the use of arbitrary lock implementations.

Note A `Condition` instance is intrinsically bound to a lock. To obtain a `Condition` instance for a particular `Lock` instance, use `Lock`'s `newCondition()` method.

`Condition` declares the following methods:

- `void await():` Forces the calling thread to wait until it's signaled or interrupted
- `boolean await(long time, TimeUnit unit):` Forces the calling thread to wait until it's signaled or interrupted, or until the specified waiting time elapses

- `long awaitNanos(long nanosTimeout)`: Forces the current thread to wait until it's signaled or interrupted, or until the specified waiting time elapses
- `void awaitUninterruptibly()`: Forces the current thread to wait until it's signaled
- `boolean awaitUntil(Date deadline)`: Forces the current thread to wait until it's signaled or interrupted, or until the specified deadline elapses
- `void signal()`: Wakes up one waiting thread
- `void signalAll()`: Wakes up all waiting threads

ReadWriteLock

Situations arise where data structures are read more often than they're modified. For example, you may have created an online dictionary of word definitions that many threads will read concurrently, whereas a single thread may add new definitions or update existing definitions. The locking framework provides a read-write locking mechanism for these situations that yields greater concurrency when reading and the safety of exclusive access when writing. This mechanism is based on the `ReadWriteLock` interface.

`ReadWriteLock` maintains a pair of locks: one lock for read-only operations and one lock for write operations. Multiple reader threads may hold the read lock simultaneously, as long as there are no writers. The write lock is exclusive: only a single thread can modify shared data. (The lock that's associated with the `synchronized` keyword is also exclusive.)

`ReadWriteLock` declares the following methods:

- `Lock readLock()`: Returns the lock that's used for reading
- `Lock writeLock()`: Returns the lock that's used for writing

ReentrantReadWriteLock

`ReadWriteLock` is implemented by the `ReentrantReadWriteLock` class, which describes a reentrant read-write lock with similar semantics to `ReentrantLock`.

You initialize a `ReentrantReadWriteLock` instance by invoking either of the following constructors:

- `ReentrantReadWriteLock():` Creates an instance of `ReentrantReadWriteLock`. This constructor is equivalent to `ReentrantReadWriteLock(false)`.
- `ReentrantReadWriteLock(boolean fair):` Creates an instance of `ReentrantReadWriteLock` with the specified fairness policy. Pass `true` to `fair` when this lock should use a fair ordering policy.

After instantiating this class, you would invoke the following methods to obtain the read and write locks:

- `ReentrantReadWriteLock.ReadLock readLock():` Returns the lock used for reading
- `ReentrantReadWriteLock.WriteLock writeLock():` Returns the lock used for writing

Each of the nested `ReadLock` and `WriteLock` classes implements the `Lock` interface and declares its own methods. Furthermore, `ReentrantReadWriteLock` declares additional methods such as the following pair:

- `int getReadHoldCount():` Returns the number of reentrant read holds on this lock by the calling thread, which is 0 when the read lock isn't held by the calling thread. A reader thread has a hold on a lock for each lock action that's not matched by an unlock action.
- `int getWriteHoldCount():` Returns the number of reentrant write holds on this lock by the calling thread, which is 0 when the write lock isn't held by the calling thread. A writer thread has a hold on a lock for each lock action that's not matched by an unlock action.

Exploring Atomic Variables

The `java.util.concurrent.atomic` package provides a small toolkit of classes that support lock-free, thread-safe operations on single variables. The classes in this package extend the notion of volatile values, fields, and array elements to those that also provide an atomic conditional update so that external synchronization isn't required.

Some of the classes located in this package are described as follows:

- `AtomicBoolean`: A boolean value that may be updated atomically
- `AtomicInteger`: An int value that may be updated atomically
- `AtomicIntegerArray`: An int array whose elements may be updated atomically
- `AtomicLong`: A long value that may be updated atomically
- `AtomicLongArray`: A long array whose elements may be updated atomically
- `AtomicReference`: An object reference that may be updated atomically
- `AtomicReferenceArray`: An object reference array whose elements may be updated atomically

In

```
public class ID {
    private static long nextID = 0;
    public static long getNextID() {
        return nextID++;
    }
}
```

we define a small utility class for returning unique long integer identifiers via the `getNextID()` class method. Because this method isn't synchronized, multiple threads could obtain the same identifier. Listing 11-1 fixes this problem by including reserved word `synchronized` in the method header.

Listing 11-1. Returning Unique Identifiers in a Thread-Safe Manner via `synchronized`

```
public class ID {
    private static long nextID = 0;
    public static synchronized long getNextID() {
        return nextID++;
    }
}
```

Although `synchronized` is appropriate for this example, excessive use of this reserved word in more complex classes can lead to deadlock, starvation, or other problems. Listing 11-2 shows you how to avoid these assaults on a concurrent application's *liveness* (the ability to execute in a timely manner) by replacing `synchronized` with an atomic variable.

Listing 11-2. Returning Unique IDs in a Thread-Safe Manner via `AtomicLong`

```
import java.util.concurrent.atomic.AtomicLong;
```

```
public class ID {  
    private static AtomicLong nextID = new AtomicLong(0);  
    public static long getNextID() {  
        return nextID.getAndIncrement();  
    }  
}
```

In Listing 11-2, we've converted `nextID` from a `long` to an `AtomicLong` instance, initializing this object to 0. We've also refactored the `getNextID()` method to call `AtomicLong`'s `getAndIncrement()` method, which increments the `AtomicLong` instance's internal long integer variable by 1 and returns the previous value in one indivisible step.

EXERCISES

The following exercises are designed to test your understanding of Chapter 11's content:

1. Define Concurrency Utilities.
2. Identify the packages in which Concurrency Utilities types are stored.
3. Define task.
4. Define executor.
5. Identify the Executor interface's limitations.
6. What differences exist between `Runnable`'s `run()` method and `Callable`'s `call()` method?
7. Define future.

8. Describe the Executors class's newFixedThreadPool() method.
9. Define synchronizer.
10. What concurrency-oriented extensions to the Collections Framework are provided by the Concurrency Utilities?
11. Define lock.
12. What is the biggest advantage that Lock objects hold over the implicit locks that are obtained when threads enter critical sections (controlled via the synchronized reserved word)?
13. How do you obtain a Condition instance for use with a particular Lock instance?
14. Define atomic variable.
15. What does the AtomicIntegerArray class describe?
16. Convert the following expressions to their atomic variable equivalents:

```
int total = ++counter;  
int total = counter--;
```

Summary

The low-level Threads API lets you create multithreaded applications that offer better performance and responsiveness over their single-threaded counterparts. However, performance issues that affect an application's scalability and other problems resulted in Java 5's introduction of the Concurrency Utilities.

The Concurrency Utilities organizes its many types into three packages: `java.util.concurrent`, `java.util.concurrent.atomic`, and `java.util.concurrent.locks`. Basic types, such as executors, thread pools, and concurrent hashmaps, are stored in `java.util.concurrent`; classes that support lock-free, thread-safe programming on single variables are stored in `java.util.concurrent.atomic`; and types for locking and waiting on conditions are stored in `java.util.concurrent.locks`.

An executor decouples task submission from task-execution mechanics and is described by the `Executor`, `ExecutorService`, and `ScheduledExecutorService` interfaces. You obtain an executor by calling one of the utility methods in the `Executors` class. Executors are associated with callables and futures.

A synchronizer facilitates common forms of synchronization. Countdown latches, cyclic barriers, exchangers, and semaphores are commonly used synchronizers.

A concurrent collection is an extension to the Collections Framework. The `BlockingQueue` and `ConcurrentMap` interfaces along with the `ArrayBlockingQueue` and `ConcurrentHashMap` classes are examples.

The `java.util.concurrent.locks` package provides a framework of interfaces and classes for locking and waiting for conditions in a manner that's distinct from built-in synchronization and `Object`'s `wait`/`notification` mechanism. Java supports locks via the commonly used `Lock`, `Condition`, and `ReadWriteLock` interfaces and via the `ReentrantLock` and `ReentrantReadWriteLock` classes.

The `java.util.concurrent.atomic` package provides a small toolkit of classes that support lock-free, thread-safe operations on single variables. The classes in this package extend the notion of volatile values, fields, and array elements to those that also provide an atomic conditional update so that external synchronization isn't required. Examples of atomic variable classes include `AtomicBoolean` and `AtomicIntegerArray`.

This chapter ends our tour of Java's Collections Framework and related Concurrency Utilities. In Chapter 12, we'll explore Java's classic I/O APIs: `File`, `RandomAccessFile`, streams, and writers/readers.

CHAPTER 12

Performing Classic I/O

Applications often input data for processing and output processing results. Data is input from a file or some other source and is output to a file or some other destination. Java supports I/O via the classic I/O APIs located in the `java.io` package and the new I/O (NIO) APIs located in `java.nio` and related subpackages (and `java.util.regex`). This chapter introduces you to the classic I/O APIs.

We don't describe each and every class of that API though, and we refrain from method listings which do not supersede copying from the official API documentation. For any details, please see the JSE documentation.

Working with the File API

Applications often interact with a *filesystem*, which is usually expressed as a hierarchy of files and directories starting from a *root directory*.

Android and other platforms on which a virtual machine runs typically support at least one filesystem. For example, a Unix/Linux (and Linux-based Android) platform combines all *mounted* (attached and prepared) disks into one virtual filesystem. In contrast, Windows associates a separate filesystem with each active disk drive.

Java offers access to the underlying platform's available filesystem(s) via its concrete `java.io.File` class. `File` declares the `File[] listRoots()` class method to return the root directories (roots) of available filesystems as an array of `File` objects.

Note The set of available filesystem roots is affected by platform-level operations, such as inserting or ejecting removable media, and disconnecting or unmounting physical or virtual disk drives.

Listing 12-1 presents a `DumpRoots` application that uses `listRoots()` to obtain an array of available filesystem roots and then outputs the array's contents.

Listing 12-1. Dumping Available Filesystem Roots to Standard Output

```
import java.io.File;

public class DumpRoots {
    public static void main(String[] args) {
        File[] roots = File.listRoots();
        for (File root: roots)
            System.out.println(root);
    }
}
```

When you run this application on a Windows platform, you might receive something similar to the following output, which reveals a couple of available roots:

```
C:\  
D:\  
E:\  
F:\
```

If you happened to run `DumpRoots` on a Unix or Linux platform, you would receive one line of output that consists of the virtual filesystem root (`/`).

Constructing File Instances

Apart from using `listRoots()`, you can obtain a `File` instance by calling a `File` constructor such as `File(String pathname)`, which creates a `File` instance that stores the pathname string. The following assignment statements demonstrate this constructor:

```
File file1 = new File("/x/y");
File file2 = new File("C:\\temp\\\\x.dat");
```

The first statement assumes a Unix/Linux platform, starts the pathname with root directory symbol `/`, and continues with directory name `x`, separator character `/`, and file or directory name `y`. (It also works on Windows, which assumes this path begins at the root directory on the current drive.)

Note A *path* is a hierarchy of directories that must be traversed to locate a file or a directory. A *pathname* is a string representation of a path; a platform-dependent *separator character* (such as the Windows backslash [\] character) appears between consecutive names.

The second statement assumes a Windows platform, starts the pathname with drive specifier C:, and continues with root directory symbol \, directory name temp, separator character \ (escaped by a second backslash), and file name x.dat (although x.dat might refer to a directory). (We could also use forward slashes on Windows.)

Each statement's pathname is an *absolute pathname*, which is a pathname that starts with the root directory symbol; no other information is required to locate the file/directory that it denotes. In contrast, a *relative pathname* doesn't start with the root directory symbol; it's interpreted via information taken from some other pathname.

Note The java.io package's classes default to resolving relative pathnames against the current user (also known as working) directory, which is identified by system property user.dir and which is typically the directory in which the virtual machine was launched. (Chapter 7 showed you how to read system properties via java.lang.System's getProperty() method.)

File instances contain abstract representations of file and directory pathnames (these files or directories may or may not exist in their filesystems) by storing *abstract pathnames*, which offer platform-independent views of hierarchical pathnames. In contrast, user interfaces and operating systems use platform-dependent *pathname strings* to name files and directories.

An abstract pathname consists of an optional platform-dependent prefix string, such as a disk drive specifier—which is / for the Unix/Linux root directory or \\ for a Windows Universal Naming Convention (UNC) pathname—and a sequence of zero or more string names. The first name in an abstract pathname may be a directory name or, in the case of Windows UNC pathnames, a hostname. Each subsequent name denotes a directory; the last name may denote a directory or a file. The *empty abstract pathname* has no prefix and an empty name sequence.

The conversion of a pathname string to or from an abstract pathname is inherently platform dependent. When a pathname string is converted into an abstract pathname,

the names within this string may be separated by the default name-separator character or by any other name-separator character that is supported by the underlying platform. When an abstract pathname is converted into a pathname string, each name is separated from the next by a single copy of the default name-separator character.

Note The *default name-separator character* is defined by the system property `file.separator` and is made available in `File`'s public static `separator` and `separatorChar` fields; the first field stores the character in a `java.lang.String` instance and the second field stores it as a `char` value.

`File` offers additional constructors for instantiating this class. See the API documentation for details.

Learning About Stored Abstract Pathnames

After obtaining a `File` object, you can interrogate it to learn about its stored abstract pathname and other properties by calling various methods that are described in the API documentation.

Listing 12-2 instantiates `File` with its pathname command-line argument and calls some of the `File` methods to learn about this pathname.

Listing 12-2. Obtaining Abstract Pathname Information

```
import java.io.File;
import java.io.IOException;

public class PathnameInfo {
    public static void main(final String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("usage: java PathnameInfo pathname");
            return;
        }
        File file = new File(args[0]);
        System.out.println("Absolute path = " + file.getAbsolutePath());
        System.out.println("Canonical path = " + file.getCanonicalPath());
        System.out.println("Name = " + file.getName());
    }
}
```

```

        System.out.println("Parent = " + file.getParent());
        System.out.println("Path = " + file.getPath());
        System.out.println("Is absolute = " + file.isAbsolute());
    }
}

```

For example, when we specify `java PathnameInfo .` (the period represents the current directory on Windows and Linux platforms), we observe something similar to the following output:

```

Absolute path = C:\prj\dev\ljfad3\ch11\code\PathnameInfo\.
Canonical path = C:\prj\dev\ljfad3\ch11\code\PathnameInfo
Name = .
Parent = null
Path = .
Is absolute = false

```

You can try the same code with a dot (.) as pathname and “” for the empty string, or of course any other pathname pointing to a file.

Learning About a Pathname's File or Directory

Other methods of the `File` class allow us to learn about whether a file exists or not, is a directory or not, and more. See Listing 12-3.

Listing 12-3. Obtaining File/Directory Information

```

import java.io.File;
import java.io.IOException;

import java.util.Date;

public class FileInfo {
    public static void main(final String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("usage: java FileInfo pathname");
            return;
    }
}

```

```

File file = new File(args[0]);
System.out.println("About " + file + ":");
System.out.println("Exists = " + file.exists());
System.out.println("Is directory = " + file.isDirectory());
System.out.println("Is file = " + file.isFile());
System.out.println("Is hidden = " + file.isHidden());
System.out.println("Last modified = " + new Date(file.
lastModified()));
System.out.println("Length = " + file.length());
}
}

```

For example, suppose we have a three-byte file named `x.dat`. When we specify `java FileInfo x.dat`, we observe the following output:

```

About x.dat:
Exists = true
Is directory = false
Is file = true
Is hidden = false
Last modified = Mon Oct 14 15:31:04 CDT 2019
Length = 3

```

Obtaining Disk Space Information

A *partition* is a platform-specific portion of storage for a filesystem, for example, `C:\`. Obtaining the amount of partition free space is important to installers and other applications. Until Java 6 arrived, the only portable way to accomplish this task was to guess by creating files of different sizes.

Java 6 added to the `File` class `long getFreeSpace()`, `long getTotalSpace()`, and `long getUsableSpace()` methods that return space information about the partition described by the `File` instance's abstract pathname. Android also supports these methods:

- `long getFreeSpace()` returns the number of unallocated bytes in the partition identified by this `File` object's abstract pathname; it returns zero when the abstract pathname doesn't name a partition.

- `long getTotalSpace()` returns the size (in bytes) of the partition identified by this `File` object's abstract pathname; it returns zero when the abstract pathname doesn't name a partition.
- `long getUsableSpace()` returns the number of bytes available to the current virtual machine on the partition identified by this `File` object's abstract pathname; it returns zero when the abstract pathname doesn't name a partition.

Although `getFreeSpace()` and `getUsableSpace()` appear to be equivalent, they differ in the following respect: unlike `getFreeSpace()`, `getUsableSpace()` checks for write permissions and other platform restrictions, resulting in a more accurate estimate.

Note The `getFreeSpace()` and `getUsableSpace()` methods return a hint (not a guarantee) that a Java application can use all (or most) of the unallocated or available bytes. These values are a hint because a program running outside the virtual machine can allocate partition space, resulting in actual unallocated and available values being lower than the values returned by these methods.

Listing Directories

`File` also declares methods that return the names of files and directories located in the directory identified by a `File` object's abstract pathname. The method names are `list()` and `listFiles()` (with varying parameters). See the API documentation for more details.

Listing 12-4 presents a `Dir(ectomy)` application that uses `list(FilenameFilter)` to obtain only those names that end with a specific extension.

Listing 12-4. Listing Specific Names

```
import java.io.File;
import java.io.FilenameFilter;

public class Dir {
    public static void main(final String[] args) {
```

```

if (args.length != 2) {
    System.err.println("usage: java Dir dirpath ext");
    return;
}
File file = new File(args[0]);
FilenameFilter fnf = new FilenameFilter() {
    @Override
    public boolean accept(File dir, String name){
        return name.endsWith(args[1]);
    }
};
String[] names = file.list(fnf);
for (String name: names)
    System.out.println(name);
}
}

```

When we, for example, specify `java Dir c:\windows exe` on our Windows platform, `Dir` outputs only those `\windows` directory file names that have the `.exe` extension:

```

bfsvc.exe
explorer.exe
fveupdate.exe
HelpPane.exe
hh.exe
notepad.exe
regedit.exe
splwow64.exe
twunk_16.exe
twunk_32.exe
winhlp32.exe
write.exe

```

The overloaded `listFiles()` methods return arrays of `Files`. For the most part, they're symmetrical with their `list()` counterparts.

Creating and Manipulating Files and Directories

`File` also declares several methods for creating new files and directories and manipulating existing files and directories. You can create and initialize files with `createNewFile()`, create temporary files via `createTempFile()`, delete files via `delete()`, make directories via `mkdir()` or `mkdirs()`, rename files via `renameTo()`, and change the modification date and time via `setLastModified()`. The API documentation tells you more.

Suppose you're designing a text-editor application that a user will use to open a text file and make changes to its content. Until the user explicitly saves these changes to the file, you want the text file to remain unchanged.

Because the user doesn't want to lose these changes when the application crashes or the computer loses power, you design the application to save these changes to a temporary file every few minutes. This way, the user has a backup of the changes.

You can use the overloaded `createTempFile()` methods to create the temporary file. If you don't specify a directory in which to store this file, it's created in the directory identified by the `java.io.tmpdir` system property.

You probably want to remove the temporary file after the user tells the application to save or discard the changes. The `deleteOnExit()` method lets you register a temporary file for deletion; it's deleted when the virtual machine ends without a crash/power loss.

Listing 12-5 presents a `TempFileDemo` application for experimenting with the `createTempFile()` and `deleteOnExit()` methods.

Listing 12-5. Experimenting with Temporary Files

```
import java.io.File;
import java.io.IOException;

public class TempFileDemo {
    public static void main(String[] args) throws IOException {
        System.out.println(System.getProperty("java.io.tmpdir"));
        File temp = File.createTempFile("text", ".txt");
        System.out.println(temp);
        temp.deleteOnExit();
    }
}
```

After outputting the location where temporary files are stored, TempFileDemo creates a temporary file whose name begins with text and ends with the .txt extension.

TempFileDemo next outputs the temporary file's name and registers the temporary file for deletion upon the successful termination of the application.

We observed the following output during one run of TempFileDemo (and the file disappeared on exit):

```
C:\Users\Owner\AppData\Local\Temp\  
C:\Users\Owner\AppData\Local\Temp\text3173127870811188221.txt
```

Setting and Getting Permissions

Java and Android support the following methods to check and set file-related permissions:

- `boolean setExecutable(boolean executable, boolean ownerOnly)` enables (pass true to executable) or disables (pass false to executable) this abstract pathname's execute permission for its owner (pass true to ownerOnly) or everyone (pass false to ownerOnly). When the filesystem doesn't differentiate between the owner and everyone, this permission always applies to everyone. It returns true when the operation succeeds. It returns false when the user doesn't have permission to change this abstract pathname's access permissions or when executable is false and the filesystem doesn't implement an execute permission.
- `boolean setExecutable(boolean executable)` is a convenience method that invokes the previous method to set the execute permission for the owner.
- `boolean setReadable(boolean readable, boolean ownerOnly)` enables (pass true to readable) or disables (pass false to readable) this abstract pathname's read permission for its owner (pass true to ownerOnly) or everyone (pass false to ownerOnly). When the filesystem doesn't differentiate between the owner and everyone, this permission always applies to everyone. It returns true when the operation succeeds. It returns false when the user doesn't have permission to change this abstract pathname's access permissions or when readable is false and the filesystem doesn't implement a read permission.

- `boolean setReadable(boolean readable)` is a convenience method that invokes the previous method to set the read permission for the owner.
- `boolean setWritable(boolean writable, boolean ownerOnly)` enables (pass true to `writable`) or disables (pass false to `writable`) this abstract pathname's write permission for its owner (pass true to `ownerOnly`) or everyone (pass false to `ownerOnly`). When the filesystem doesn't differentiate between the owner and everyone, this permission always applies to everyone. It returns true when the operation succeeds. It returns false when the user doesn't have permission to change this abstract pathname's access permissions.
- `boolean setWritable(boolean writable)` is a convenience method that invokes the previous method to set the write permission for the owner.

Along with these methods, Java 6 retrofitted `File`'s `boolean canRead()` and `boolean canWrite()` methods and introduced a `boolean canExecute()` method to return an abstract pathname's access permissions. These methods return true when the file or directory object identified by the abstract pathname exists and when the appropriate permission is in effect. For example, `canWrite()` returns true when the abstract pathname exists and when the application has permission to write to the file.

The `canRead()`, `canWrite()`, and `canExecute()` methods can be used to implement a simple utility that identifies which permissions have been assigned to an arbitrary file or directory. This utility's source code is presented in Listing 12-6.

Listing 12-6. Checking a File's or Directory's Permissions

```
import java.io.File;

public class Permissions {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage: java Permissions filespec");
            return;
    }
}
```

```
File file = new File(args[0]);
System.out.println("Checking permissions for " + args[0]);
System.out.println(" Execute = " + file.canExecute());
System.out.println(" Read = " + file.canRead());
System.out.println(" Write = " + file.canWrite());
}
}
```

Compile Listing 12-6 (`javac Permissions.java`). Assuming a readable and executable (only) file named `x` in the current directory, `java Permissions x` generates the following output:

```
Checking permissions for x
Execute = true
Read = true
Write = false
```

Working with the RandomAccessFile API

Files can be created and/or opened for *random access* in which a mixture of write and read operations can occur until the file is closed. Java supports this random access via its concrete `java.io.RandomAccessFile` class.

Note `RandomAccessFile` has its place in Android app development. For example, you can use this class to read an app's raw resource file. To learn how, check out “`RandomAccessFile` in Android raw resource file” (<http://stackoverflow.com/questions/9335379/randomaccessfile-in-android-raw-resource-file>).

The details about how to work with random access files can be looked up in the API documentation. For Android development it is usually recommended to work with databases instead of random access files.

Working with Streams

Along with `File` and `RandomAccessFile`, Java uses streams to perform I/O operations. A *stream* is an ordered sequence of bytes of arbitrary length. Bytes flow over an *output stream* from an application to a destination and flow over an *input stream* from a source to an application. Figure 12-1 illustrates these flows.

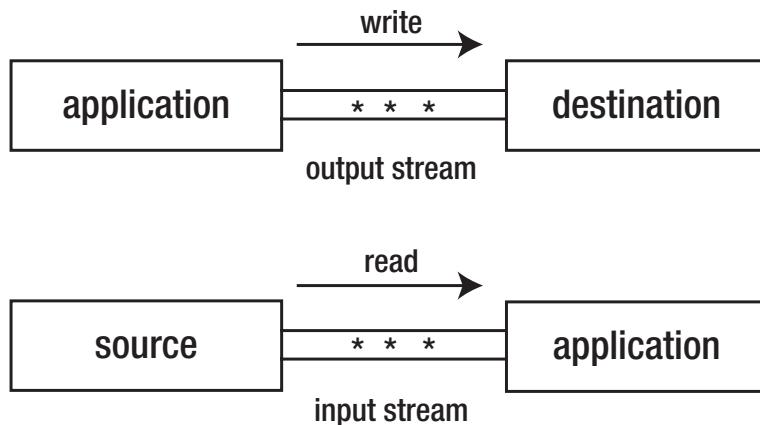


Figure 12-1. Conceptualizing output and input streams as flows of bytes

Note Java's use of the word *stream* is analogous to stream of water, stream of electrons, and so on.

Java recognizes various stream destinations, such as byte arrays, files, screens, *sockets* (network endpoints), and thread pipes. Java also recognizes various stream sources. Examples include byte arrays, files, keyboards, sockets, and thread pipes. (We will discuss sockets in Chapter 13.)

Stream Classes Overview

The `java.io` package provides several output stream and input stream classes that are descendants of the abstract `OutputStream` and `InputStream` classes. Figure 12-2 reveals the hierarchy of the most important output stream classes.

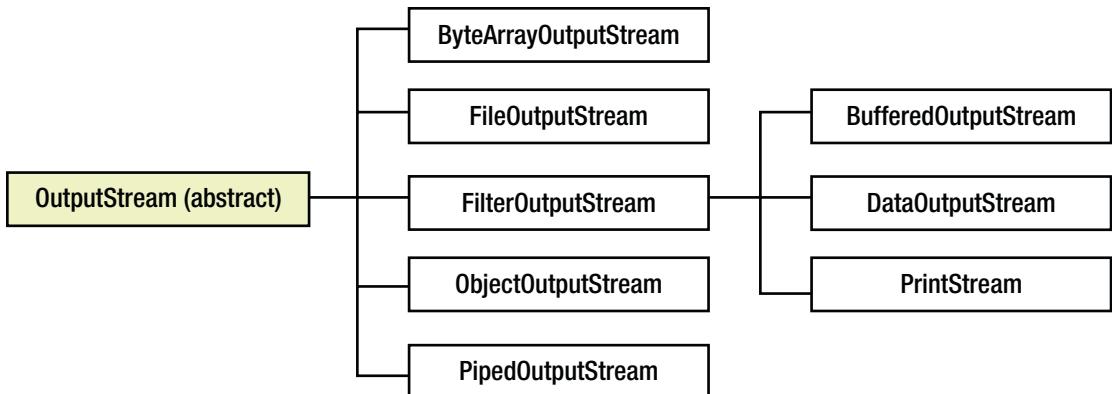


Figure 12-2. All output stream classes except for `PrintStream` are denoted by their `OutputStream` suffixes

Figure 12-3 reveals the hierarchy of the most important input stream classes.

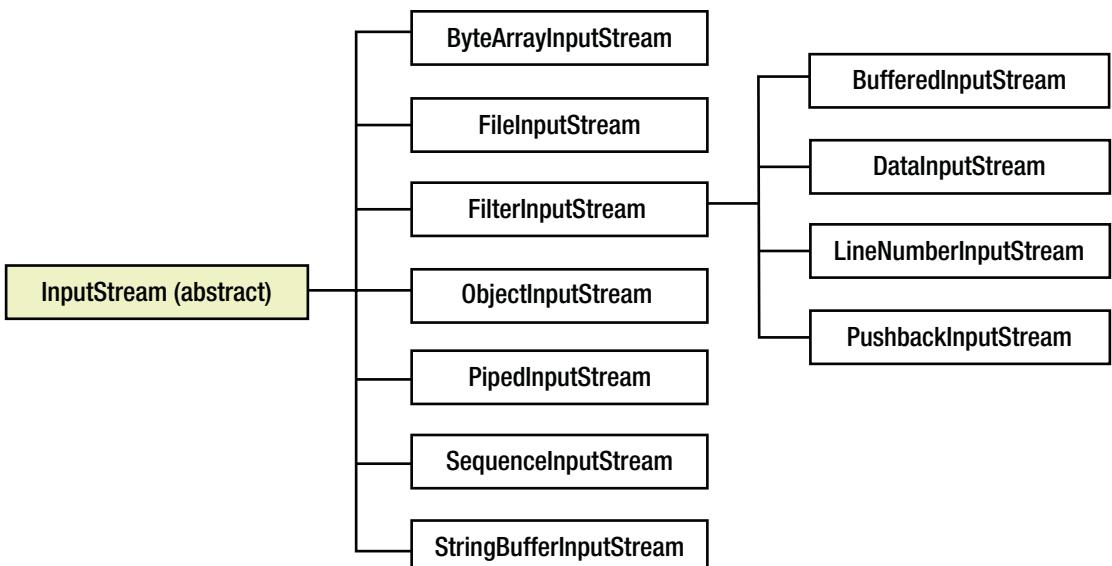


Figure 12-3. Note that `LineNumberInputStream` and `StringBufferInputStream` are deprecated

`LineNumberInputStream` and `StringBufferInputStream` have been deprecated because they don't support different character encodings, a topic we discuss later in this chapter. `LineNumberReader` and `StringReader` are their replacements. (We discuss readers later in this chapter.)

Note `PrintStream` is another class that should be deprecated because it doesn't support different character encodings; `PrintWriter` is its replacement. However, it's doubtful that Oracle (and Google) will deprecate this class because `PrintStream` is the type of the `System` class's `out` and `err` class fields, and too much legacy code depends upon this fact.

In the next several sections, we take you on a tour of most of `java.io`'s output stream and input stream classes.

ByteArrayOutputStream and ByteArrayInputStream

Byte arrays are often useful as stream destinations and sources. The `ByteArrayOutputStream` class lets you write a stream of bytes to a byte array; the `ByteArrayInputStream` class lets you read a stream of bytes from a byte array.

The following example uses `ByteArrayOutputStream()` to create a byte array output stream with an internal byte array set to the default size:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

The following example uses `ByteArrayInputStream(byte[])` to create a byte array input stream whose source is a copy of the previous byte array output stream's byte array:

```
ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
```

`ByteArrayOutputStream` and `ByteArrayInputStream` are useful in a scenario where you need to convert an image to an array of bytes, process these bytes in some manner, and convert the bytes back to the image.

For example, suppose you're writing an Android-based image-processing application. You decode a file containing the image into an Android-specific `android.graphics.Bitmap` instance, compress this instance into a `ByteArrayOutputStream` instance, obtain a copy of the byte array output stream's array, process this array in some manner, convert this array to a `ByteArrayInputStream` instance, and use the byte array input stream to decode these bytes into another `Bitmap` instance, as follows:

```
String pathname = ... ; // Assume a legitimate pathname to an image.  
Bitmap bm = BitmapFactory.decodeFile(pathname);  
ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```
if (bm.compress(Bitmap.CompressFormat.PNG, 100, baos)) {  
    byte[] imageBytes = baos.toByteArray();  
    // Do something with imageBytes.  
    bm = BitmapFactory.decodeStream(new ByteArrayInputStream(imageBytes));  
}
```

This example obtains an image file's pathname and then calls the concrete `android.graphics.BitmapFactory` class's `Bitmap decodeFile(String pathname)` class method. This method decodes the image file identified by pathname into a bitmap and returns a `Bitmap` instance that represents this bitmap.

After creating a `ByteArrayOutputStream` object, the example uses the returned `BitMap` instance to call `BitMap`'s `boolean compress(Bitmap.CompressFormat format, int quality, OutputStream stream)` method to write a compressed version of the bitmap to the byte array output stream:

- `format` identifies the format of the compressed image. We've chosen to use the popular Portable Network Graphics (PNG) format.
- `quality` hints to the compressor as to how much compression is required. This value ranges from 0 through 100, where 0 means maximum compression at the expense of quality and 100 means maximum quality at the expense of compression. Formats such as PNG ignore `quality` because they employ lossless compression.
- `stream` identifies the stream on which to write the compressed image data.

When `compress()` returns true, which means that it successfully compressed the image onto the byte array output stream in the PNG format, the `ByteArrayOutputStream` object's `toByteArray()` method is called to create and return a byte array with the image's bytes.

Continuing, the array is processed, a `ByteArrayInputStream` object is created with the processed bytes serving as the source of this stream, and `BitmapFactory`'s `BitMap decodeStream(InputStream is)` class method is called to convert the byte array input stream's source of bytes to a `BitMap` instance

FileOutputStream and FileInputStream

Files are common stream destinations and sources. The concrete `FileOutputStream` class lets you write a stream of bytes to a file; the concrete `FileInputStream` class lets you read a stream of bytes from a file.

The following example uses `FileOutputStream(String pathname)` to create a file output stream with `employee.dat` as its destination:

```
FileOutputStream fos = new FileOutputStream("employee.dat");
```

Tip `FileOutputStream(String name)` overwrites an existing file. To append data instead of overwriting existing content, call a `FileOutputStream` constructor that includes a boolean `append` parameter and pass `true` to this parameter.

The following example uses `FileInputStream(String name)` to create a file input stream with `employee.dat` as its source:

```
FileInputStream fis = new FileInputStream("employee.dat");
```

`FileOutputStream` and `FileInputStream` are useful in a file-copying context. Listing 12-7 presents the source code to a `Copy` application that provides a demonstration.

Listing 12-7. Copying a Source File to a Destination File

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Copy {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Copy srcfile dstfile");
            return;
    }}
```

```

InputStream fis = null;
OutputStream fos = null;
try {
    fis = new FileInputStream(args[0]);
    fos = new FileOutputStream(args[1]);
    int b; // We chose b instead of byte because byte is a reserved word.
    while ((b = fis.read()) != -1)
        fos.write(b);
} catch (FileNotFoundException fnfe) {
    System.err.println(args[0] + " could not be opened for input, or " +
                       args[1] + " could not be created for output");
} catch (IOException ioe) {
    System.err.println("I/O error: " + ioe.getMessage());
} finally {
    if (fis != null)
        try {
            fis.close();
        } catch (IOException ioe) {
        }

    if (fos != null)
        try {
            fos.close();
        } catch (IOException ioe) {
        }
    }
}
}

```

Listing 12-7's `main()` method first verifies that two command-line arguments, identifying the names of source and destination files, are specified. It then proceeds to instantiate `FileInputStream` and `FileOutputStream` and enter a while loop that repeatedly reads bytes from the file input stream and writes them to the file output stream.

Of course something might go wrong. Perhaps the source file doesn't exist, or perhaps the destination file cannot be created (e.g., a same-named read-only file might exist). In either scenario, `FileNotFoundException` is thrown and must be handled. Another possibility is that an I/O error occurred during the copy operation. Such an error results in `IOException`.

Regardless of an exception being thrown or not, the input and output streams are closed via the `finally` block. In a simple application like this, we could ignore the `close()` method calls and let the application terminate. Although Java automatically closes open files at this point, it's good form to explicitly close files upon exit.

Because `close()` is capable of throwing an instance of the checked `IOException` class, a call to this method is wrapped in a `try` block with an appropriate catch block that catches this exception. Notice the `if` statement that precedes each `try` block. This statement is necessary to avoid a thrown `NullPointerException` instance should either `fis` or `fos` contain the null reference.

PipedOutputStream and PipedInputStream

Threads must often communicate. One approach involves using shared variables. Another approach involves using piped streams via the `PipedOutputStream` and `PipedInputStream` classes. The `PipedOutputStream` class lets a sending thread write a stream of bytes to an instance of the `PipedInputStream` class, which a receiving thread uses to subsequently read those bytes.

Caution Attempting to use a `PipedOutputStream` object and a `PipedInputStream` object from a single thread is not recommended because it might deadlock the thread.

`PipedOutputStream` declares a `void connect(PipedInputStream dest)` method that connects this piped output stream to `dest`. This method throws `IOException` when this piped output stream is already connected to another piped input stream.

`PipedInputStream` declares a `void connect(PipedOutputStream src)` method that connects this piped input stream to `src`. This method throws `IOException` when this piped input stream is already connected to another piped output stream.

The easiest way to create a pair of piped streams is in the same thread and in either order. For example, you can first create the piped output stream.

```
PipedOutputStream pos = new PipedOutputStream();
PipedInputStream pis = new PipedInputStream(pos);
```

Alternatively, you can first create the piped input stream.

```
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream(pis);
```

You can leave both streams unconnected and later connect them to each other using the appropriate piped stream's `connect()` method, as follows:

```
PipedOutputStream pos = new PipedOutputStream();
PipedInputStream pis = new PipedInputStream();
// ...
pos.connect(pis);
```

[Listing 12-8](#) presents a `PipedStreamsDemo` application whose sender thread streams a sequence of randomly generated byte integers to a receiver thread, which outputs this sequence.

Listing 12-8. Piping Randomly Generated Bytes from a Sender Thread to a Receiver Thread

```
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

public class PipedStreamsDemo {
    public static void main(String[] args) throws IOException {
        final PipedOutputStream pos = new PipedOutputStream();
        final PipedInputStream pis = new PipedInputStream(pos);
        Runnable senderTask = new Runnable() {
            final static int LIMIT = 10;
            @Override
```

```
public void run() {
    try {
        for (int i = 0 ; i < LIMIT; i++)
            pos.write((byte) (Math.random() * 256));
    } catch (IOException ioe) {
        ioe.printStackTrace();
    } finally {
        try {
            pos.close();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
};

Runnable receiverTask = new Runnable() {
    @Override
    public void run() {
        try {
            int b;
            while ((b = pis.read()) != -1)
                System.out.println(b);
        } catch (IOException ioe) {
            ioe.printStackTrace();
        } finally {
            try {
                pis.close();
            } catch (IOException ioe) {
                ioe.printStackTrace();
            }
        }
    }
};
```

```

        Thread sender = new Thread(senderTask);
        Thread receiver = new Thread(receiverTask);
        sender.start();
        receiver.start();
    }
}

```

Listing 12-8's main() method creates piped output and piped input streams that will be used by the senderTask thread to communicate a sequence of randomly generated byte integers and by the receiverTask thread to receive this sequence.

The sender task's run() method explicitly closes its pipe stream when it finishes sending the data. If it didn't do this, an IOException instance with a "write end dead" message would be thrown when the receiver thread invoked read() for the final time (which would otherwise return -1 to indicate end of stream). For more information on this message, check out Daniel Ferbers's "Whats this? IOException: Write end dead" blog post (<http://techtavern.wordpress.com/2008/07/16/whats-this-ioexception-write-end-dead/>).

Compile Listing 12-8 (javac PipedStreamsDemo.java) and run this application (java PipedStreamsDemo). You'll discover output similar to the following:

```

93
23
125
50
126
131
210
29
150
91

```

FilterOutputStream and FilterInputStream

Byte array, file, and piped streams pass bytes unchanged to their destinations. Java also supports *filter streams* that buffer, compress/uncompress, encrypt/decrypt, or otherwise manipulate a stream's byte sequence (i.e., input to the filter) before it reaches its destination.

A *filter output stream* takes the data passed to its `write()` methods (the input stream), filters it, and writes the filtered data to an underlying output stream, which might be another filter output stream or a destination output stream such as a file output stream.

Filter output streams are created from subclasses of the concrete `FilterOutputStream` class, an `OutputStream` subclass. `FilterOutputStream` declares a single `FilterOutputStream(OutputStream out)` constructor that creates a filter output stream built on top of `out`, the underlying output stream.

Listing 12-9 reveals that it's easy to subclass `FilterOutputStream`. At minimum, you declare a constructor that passes its `OutputStream` argument to `FilterOutputStream`'s constructor and override `FilterOutputStream`'s `write(int)` method.

Listing 12-9. Scrambling a Stream of Bytes

```
import java.io.FilterOutputStream;
import java.io.IOException;
import java.io.OutputStream;

public class ScrambledOutputStream extends FilterOutputStream {
    private int[] map;

    public ScrambledOutputStream(OutputStream out, int[] map) {
        super(out);
        if (map == null)
            throw new NullPointerException("map is null");
        if (map.length != 256)
            throw new IllegalArgumentException("map.length != 256");
        this.map = map;
    }

    @Override
    public void write(int b) throws IOException {
        out.write(map[b]);
    }
}
```

Listing 12-9 presents a `ScrambledOutputStream` class that performs trivial encryption on its input stream by scrambling the input stream's bytes via a remapping operation. This constructor takes a pair of arguments:

- `out` identifies the output stream on which to write the scrambled bytes.
- `map` identifies an array of 256 byte integer values to which input stream bytes map.

The constructor first passes its `out` argument to the `FilterOutputStream` parent via a `super(out)` call. It then verifies its `map` argument's integrity (`map` must be nonnull and have a length of 256: a byte stream offers exactly 256 bytes to map) before saving `map`.

The `write(int)` method is trivial: it calls the underlying output stream's `write(int)` method with the byte to which argument `b` maps. `FilterOutputStream` declares `out` to be protected (for performance), which is why we can directly access this field.

Note It's only essential to override `write(int)` because `FilterOutputStream`'s other two `write()` methods are implemented via this method.

A *filter input stream* takes the data obtained from its underlying input stream—which might be another filter input stream or a source input stream such as a file input stream—filters it, and makes this data available via its `read()` methods (the output stream).

Filter input streams are created from subclasses of the concrete `FilterInputStream` class, an `InputStream` subclass. `FilterInputStream` declares a single `FilterInputStream(InputStream in)` constructor that creates a filter input stream built on top of `in`, the underlying input stream.

It is easy to subclass `FilterInputStream`. At minimum, declare a constructor that passes its `InputStream` argument to `FilterInputStream`'s constructor and override `FilterInputStream`'s `read()` and `read(byte[], int, int)` methods.

Note When a stream instance is passed to another stream class's constructor, the two streams are *chained together*. For example, the scrambled input stream is chained to the file input stream.

For an example of a filter output stream and its complementary filter input stream, check out the “Extending Java Streams to Support Bit Streams” article (www.drdobbs.com/184410423) on the Dr. Dobb’s website. This article introduces `BitStreamOutputStream` and `BitStreamInputStream` classes that are useful for outputting and inputting bit streams. The article then demonstrates these classes in a Java implementation of the Lempel-Ziv-Welch (LZW) data compression and decompression algorithm.

BufferedOutputStream and BufferedInputStream

`FileOutputStream` and `FileInputStream` have a performance problem. Each file output stream `write()` method call and file input stream `read()` method call results in a call to one of the underlying platform’s native methods, and these native calls slow down I/O.

Note A *native method* is an underlying platform API function that Java connects to an application via the *Java Native Interface (JNI)*. Java supplies reserved word `native` to identify a native method. For example, the `RandomAccessFile` class declares a `private native void open(String name, int mode)` method. When a `RandomAccessFile` constructor calls this method, Java asks the underlying platform (via the `JNI`) to open the specified file in the specified mode on Java’s behalf.

The concrete `BufferedOutputStream` and `BufferedInputStream` filter stream classes improve performance by minimizing underlying output stream `write()` and underlying input stream `read()` method calls. Instead, calls to `BufferedOutputStream`’s `write()` and `BufferedInputStream`’s `read()` methods take Java buffers into account.

- When a write buffer is full, `write()` calls the underlying output stream `write()` method to empty the buffer. Subsequent calls to `BufferedOutputStream`’s `write()` methods store bytes in this buffer until it’s once again full.
- When the read buffer is empty, `read()` calls the underlying input stream `read()` method to fill the buffer. Subsequent calls to `BufferedInputStream`’s `read()` methods return bytes from this buffer until it’s once again empty.

The following example chains a `BufferedOutputStream` instance to a `FileOutputStream` instance. Subsequent `write()` method calls on the `BufferedOutputStream` instance buffer bytes and occasionally result in internal `write()` method calls on the encapsulated `FileOutputStream` instance.

```
FileOutputStream fos = new FileOutputStream("employee.dat");
BufferedOutputStream bos = new BufferedOutputStream(fos); // Chain bos to fos.
bos.write(0); // Write to employee.dat through the buffer.
// Additional write() method calls.
bos.close(); // This method call internally calls fos's close() method.
```

The following example chains a `BufferedInputStream` instance to a `FileInputStream` instance. Subsequent `read()` method calls on the `BufferedInputStream` instance unbuffer bytes and occasionally result in internal `read()` method calls on the encapsulated `FileInputStream` instance.

```
FileInputStream fis = new FileInputStream("employee.dat");
BufferedInputStream bis = new BufferedInputStream(fis); // Chain bis to fis.
int ch = bis.read(); // Read employee.dat through the buffer.
// Additional read() method calls.
bis.close(); // This method call internally calls fis's close() method.
```

DataOutputStream and DataInputStream

`FileOutputStream` and `FileInputStream` are useful for writing and reading bytes and arrays of bytes. However, they provide no support for writing and reading primitive-type values (such as integers) and strings.

For this reason, Java provides the concrete `DataOutputStream` and `DataInputStream` filter stream classes. Each class overcomes this limitation by providing methods to write or read primitive-type values and strings in a platform-independent way.

- Integer values are written and read in *big-endian format* (the most significant byte comes first). Check out Wikipedia’s “Endianness” entry (<http://en.wikipedia.org/wiki/Endianness>) to learn about the concept of *endianness*.

- Floating-point and double-precision floating-point values are written and read according to the IEEE 754 standard, which specifies 4 bytes per floating-point value and 8 bytes per double-precision floating-point value.
- Strings are written and read according to a modified version of *UTF-8*, a variable-length encoding standard for efficiently storing 2-byte Unicode characters. Check out Wikipedia’s “*UTF-8*” entry (<http://en.wikipedia.org/wiki/Utf-8>) to learn more about *UTF-8*.

`DataOutputStream` declares a single `DataOutputStream(OutputStream out)` constructor. Because this class implements the `DataOutput` interface, `DataOutputStream` also provides access to the same-named write methods as provided by `RandomAccessFile`.

`DataInputStream` declares a single `DataInputStream(InputStream in)` constructor. Because this class implements the `DataInput` interface, `DataInputStream` also provides access to the same-named read methods as provided by `RandomAccessFile`.

Listing 12-10 presents the source code to a `DataStreamsDemo` application that uses a `DataOutputStream` instance to write multibyte values to a `FileOutputStream` instance and uses a `DataInputStream` instance to read multibyte values from a `FileInputStream` instance.

Listing 12-10. Outputting and Then Inputting a Stream of Multibyte Values

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class DataStreamsDemo {
    private final static String FILENAME = "values.dat";

    public static void main(String[] args) {
        DataOutputStream dos = null;
        DataInputStream dis = null;
```

```
try {
    FileOutputStream fos = new FileOutputStream(FILENAME);
    dos = new DataOutputStream(fos);
    dos.writeInt(1995);
    dos.writeUTF("Saving this String in modified UTF-8 format!");
    dos.writeFloat(1.0F);
    dos.close(); // Close underlying file output stream.
    // The following null assignment prevents another close attempt on
    // dos (which is now closed) should IOException be thrown from
    // subsequent method calls.
    dos = null;
    FileInputStream fis = new FileInputStream(FILENAME);
    dis = new DataInputStream(fis);
    System.out.println(dis.readInt());
    System.out.println(dis.readUTF());
    System.out.println(dis.readFloat());
} catch (IOException ioe) {
    System.err.println("I/O error: " + ioe.getMessage());
} finally {
    if (dos != null)
        try {
            dos.close();
        } catch (IOException ioe2) {
        }
    if (dis != null)
        try {
            dis.close();
        } catch (IOException ioe2) {
        }
}
}
```

`DataStreamsDemo` creates a file named `values.dat`; calls `DataOutputStream` methods to write an integer, a string, and a floating-point value to this file; and calls `DataInputStream` methods to read back these values. Unsurprisingly, it generates the following output:

```
1995  
Saving this String in modified UTF-8 format!  
1.0
```

Object Serialization and Deserialization

Java provides the `DataOutputStream` and `DataInputStream` classes to stream primitive-type values and `String` objects. However, you cannot use these classes to stream non-`String` objects. Instead, you must use object serialization and deserialization to stream objects of arbitrary types.

Object serialization is a virtual machine mechanism for *serializing* object state into a stream of bytes. Its *deserialization* counterpart is a virtual machine mechanism for *deserializing* this state from a byte stream.

Note An object's state consists of instance fields that store primitive-type values and/or references to other objects. When an object is serialized, the objects that are part of this state are also serialized (unless you prevent them from being serialized). Furthermore, the objects that are part of those objects' states are serialized (unless you prevent this), and so on.

Java supports default serialization and deserialization, custom serialization and deserialization, and externalization.

Default Serialization and Deserialization

Default serialization and deserialization is the easiest form to use but offers little control over how objects are serialized and deserialized. Although Java handles most of the work on your behalf, there are a couple of tasks that you must perform.

Your first task is to have the class of the object that's to be serialized implement the `java.io.Serializable` interface, either directly or indirectly via the class's superclass. The rationale for implementing `Serializable` is to avoid unlimited serialization.

Note `Serializable` is an empty marker interface (there are no methods to implement) that a class implements to tell the virtual machine that it's okay to serialize the class's objects. When the serialization mechanism encounters an object whose class doesn't implement `Serializable`, it throws an instance of the `java.io.NotSerializableException` class (an indirect subclass of `IOException`).

Unlimited serialization is the process of serializing an entire object graph. Java doesn't support unlimited serialization for the following reasons:

- *Security*: If Java automatically serialized an object containing sensitive information (such as a password or a credit card number), it would be easy for a hacker to discover this information and wreak havoc. It's better to give the developer a choice to prevent this from happening.
- *Performance*: Serialization leverages the Reflection API (discussed in Chapter 8), which tends to slow down application performance. Unlimited serialization could really hurt an application's performance.
- *Objects not amenable to serialization*: Some objects exist only in the context of a running application and it's meaningless to serialize them. For example, a file stream object that's deserialized no longer represents a connection to a file.

Listing 12-11 declares an `Employee` class that implements the `Serializable` interface to tell the virtual machine that it's okay to serialize `Employee` objects.

Listing 12-11. Implementing `Serializable`

```
import java.io.Serializable;

public class Employee implements Serializable {
    private String name;
    private int age;
```

```

public Employee(String name, int age) {
    this.name = name;
    this.age = age;
}

public String getName() { return name; }

public int getAge() { return age; }
}

```

Because `Employee` implements `Serializable`, the serialization mechanism will not throw a `NotSerializableException` instance when serializing an `Employee` object. Not only does `Employee` implement `Serializable`, the `String` class also implements this interface.

Your second task is to work with the `ObjectOutputStream` class and its `writeObject()` method to serialize an object and the `ObjectInputStream` class and its `readObject()` method to deserialize the object.

Note Although `ObjectOutputStream` extends `OutputStream` instead of `FilterOutputStream`, and although `ObjectInputStream` extends `InputStream` instead of `FilterInputStream`, these classes behave as filter streams.

Java provides the concrete `ObjectOutputStream` class to initiate the serialization of an object's state to an object output stream. This class declares an `ObjectOutputStream(ObjectStream out)` constructor that chains the object output stream to the output stream specified by `out`.

When you pass an output stream reference to `out`, this constructor attempts to write a serialization header to that output stream. It throws `NullPointerException` when `out` is `null` and `IOException` when an I/O error prevents it from writing this header.

`ObjectOutputStream` serializes an object via its `void writeObject(Object obj)` method. This method attempts to write information about `obj`'s class followed by the values of `obj`'s instance fields to the underlying output stream.

`writeObject()` doesn't serialize the contents of `static` fields. In contrast, it serializes the contents of all instance fields that are not explicitly prefixed with the `transient` reserved word. For example, consider the following field declaration:

```
public transient char[] password;
```

This declaration specifies `transient` to avoid serializing a password for some hacker to encounter. The virtual machine’s serialization mechanism ignores any instance field that’s marked `transient`.

Note Check out the “Transience” blog post (www.javaworld.com/community/node/13451) to learn more about `transient`.

`writeObject()` throws `IOException` or an instance of an `IOException` subclass when something goes wrong. For example, this method throws `NotSerializableException` when it encounters an object whose class doesn’t implement `Serializable`.

Note Because `ObjectOutputStream` implements `DataOutput`, it also declares methods for writing primitive-type values and strings to an object output stream.

Java provides the concrete `ObjectInputStream` class to initiate the deserialization of an object’s state from an object input stream. This class declares an `ObjectInputStream(InputStream in)` constructor that chains the object input stream to the input stream specified by `in`.

When you pass an input stream reference to `in`, this constructor attempts to read a serialization header from that input stream. It throws `NullPointerException` when `in` is `null`, `IOException` when an I/O error prevents it from reading this header, and `java.io.StreamCorruptedException` (an indirect subclass of `IOException`) when the stream header is incorrect.

`ObjectInputStream` deserializes an object via its `Object readObject()` method. This method attempts to read information about `obj`’s class followed by the values of `obj`’s instance fields from the underlying input stream.

`readObject()` throws `java.lang.ClassNotFoundException`, `IOException`, or an instance of an `IOException` subclass when something goes wrong. For example, this method throws `java.io.OptionalDataException` when it encounters primitive-type values instead of objects.

Note Because `ObjectInputStream` implements `DataInput`, it also declares methods for reading primitive-type values and strings from an object input stream.

Listing 12-12 presents an application that uses these classes to serialize and deserialize an instance of Listing 12-11's Employee class to and from an employee.dat file.

Listing 12-12. Serializing and Deserializing an Employee Object

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializationDemo {
    final static String FILENAME = "employee.dat";

    public static void main(String[] args) {
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try {
            FileOutputStream fos = new FileOutputStream(FILENAME);
            oos = new ObjectOutputStream(fos);
            Employee emp = new Employee("John Doe", 36);
            oos.writeObject(emp);
            oos.close();
            oos = null;
            FileInputStream fis = new FileInputStream(FILENAME);
            ois = new ObjectInputStream(fis);
            emp = (Employee) ois.readObject(); // (Employee) cast is
            necessary.
            ois.close();
            System.out.println(emp.getName());
            System.out.println(emp.getAge());
        } catch (ClassNotFoundException cnfe) {
            System.err.println(cnfe.getMessage());
        } catch (IOException ioe) {
            System.err.println(ioe.getMessage());
        } finally {
```

```
if (oos != null)
    try {
        oos.close();
    } catch (IOException ioe) {
    }
if (ois != null)
    try {
        ois.close();
    } catch (IOException ioe) {
    }
}
}
```

Listing 12-12's `main()` method first instantiates `Employee` and serializes this instance via `writeObject()` to `employee.dat`. It then deserializes this instance from this file via `readObject()` and invokes the instance's `getName()` and `getAge()` methods. Along with `employee.dat`, you'll discover the following output when you run this application:

```
John Doe
36
```

There's no guarantee that the same class will exist when a serialized object is deserialized (perhaps an instance field has been deleted). During deserialization, this mechanism causes `readObject()` to throw `java.io.InvalidClassException`—an indirect subclass of the `IOException` class—when it detects a difference between the deserialized object and its class.

Every serialized object has an identifier. The deserialization mechanism compares the identifier of the object being deserialized with the serialized identifier of its class (all serializable classes are automatically given unique identifiers unless they explicitly specify their own identifiers) and causes `InvalidClassException` to be thrown when it detects a mismatch.

Perhaps you've added an instance field to a class, and you want the deserialization mechanism to set the instance field to a default value rather than have `readObject()` throw an `InvalidClassException` instance. (The next time you serialize the object, the new field's value will be written out.)

You can avoid the thrown `InvalidClassException` instance by adding a static `final long serialVersionUID = long integer value;` declaration to the class. The *long integer value* must be unique and is known as a *stream unique identifier (SUID)*.

During deserialization, the virtual machine will compare the serialized object's SUID to its class's SUID. If they match, `readObject()` will not throw `InvalidClassException` when it encounters a *compatible class change* (such as adding an instance field). However, it will still throw this exception when it encounters an *incompatible class change* (such as changing an instance field's name or type).

Note Whenever you change a class in some fashion, you must calculate a new SUID and assign it to `serialVersionUID`.

Externalization

Along with default serialization/deserialization and custom serialization/deserialization, Java supports externalization. Unlike default/custom serialization/deserialization, *externalization* offers complete control over the serialization and deserialization tasks.

Note Externalization helps you improve the performance of the reflection-based serialization and deserialization mechanisms by giving you complete control over what fields are serialized and deserialized.

For details, please see the API documentation of the `Externalizable` interface.

PrintStream

Of all the stream classes, `PrintStream` is an oddball: it should have been named `PrintOutputStream` for consistency with the naming convention. This filter output stream class writes string representations of input data items to the underlying output stream.

Note `PrintStream` uses the default character encoding to convert a string's characters to bytes. (We'll discuss character encodings when we introduce you to writers and readers in the next section.) Because `PrintStream` doesn't support different character encodings, you should use the equivalent `PrintWriter` class instead of `PrintStream`. However, you need to know about `PrintStream` because of standard I/O (see Chapter 1 for an introduction to this topic).

`PrintStream` instances are print streams whose various `print()` and `println()` methods print string representations of integers, floating-point values, and other data items to the underlying output stream. Unlike the `print()` methods, `println()` methods append a line terminator (as defined by the operating system) to their output.

The `println()` methods call their corresponding `print()` methods followed by the equivalent of the `void println()` method, which eventually results in `line.separator`'s value being output. For example, `void println(int x)` outputs `x`'s string representation and calls this method to output the line separator.

`PrintStream` offers three other features that you'll find useful:

- Unlike other output streams, a print stream never rethrows an `IOException` instance thrown from the underlying output stream. Instead, exceptional situations set an internal flag that can be tested by calling `PrintStream`'s boolean `checkError()` method, which returns true to indicate a problem.
- `PrintStream` objects can be created to automatically flush their output to the underlying output stream. In other words, the `flush()` method is automatically called after a byte array is written, one of the `println()` methods is called, or a newline is written.
- `PrintStream` declares a `PrintStream format(String format, Object... args)` method for achieving formatted output. Behind the scene, this method works with the `Formatter` class that we introduce in Chapter 14. `PrintStream` also declares a `printf(String format, Object... args)` convenience method that delegates to the `format()` method. For example, invoking `printf()` via `out.printf(format, args)` is identical to invoking `out.format(format, args)`.

Standard I/O Revisited

In previous chapters we already frequently used standard I/O functionalities to write data to the console or some pipe. Namely, `System.err` and `System.out` can readily be used to write data to the standard error and output stream. And also, although we didn't stress it, we can use `System.in.read()` to input data from the standard input stream.

`System.in`, `System.out`, and `System.err` are formally described by the following class fields in the `System` class:

- `public static final InputStream in`
- `public static final PrintStream out`
- `public static final PrintStream err`

These fields contain references to `InputStream` and `PrintStream` objects that represent the standard input, standard output, and standard error streams.

When you invoke `System.in.read()`, the input is originating from the source identified by the `InputStream` instance assigned to `in`. Similarly, when you invoke `System.out.print()` or `System.err.println()`, the output is being sent to the destination identified by the `PrintStream` instance assigned to `out` or `err`, respectively.

Tip On an Android device, you can view content sent to standard output and standard error by looking at the Android Studio's logcat output.

Java initializes `in` to refer to the keyboard or a file when the standard input stream is redirected to the file. Similarly, Java initializes `out/err` to refer to the screen or a file when the standard output/error stream is redirected to the file. You can programmatically specify the input source, output destination, and error destination by calling the following `System` class methods:

- `void setIn(InputStream in)`
- `void setOut(PrintStream out)`
- `void setErr(PrintStream err)`

Listing 12-13 presents an application that shows you how to use these methods to programmatically redirect the standard input, standard output, and standard error destinations.

Listing 12-13. Programmatically Specifying the Standard Input Source and Standard Output/Error Destinations

```

import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintStream;

public class RedirectIO {
    public static void main(String[] args) throws IOException {
        if (args.length != 3) {
            System.err.println("usage: java RedirectIO stdinfile stdoutfile
stderofile");
            return;
        }

        System.setIn(new FileInputStream(args[0]));
        System.setOut(new PrintStream(args[1]));
        System.setErr(new PrintStream(args[2]));

        int ch;
        while ((ch = System.in.read()) != -1)
            System.out.print((char) ch);

        System.err.println("Redirected error output");
    }
}

```

Listing 12-13 presents a `RedirectIO` application that lets you specify (via command-line arguments) the name of a file from which `System.in.read()` obtains its content as well as the names of files to which `System.out.print()` and `System.err.println()` send their content. It then proceeds to copy standard input to standard output and then demonstrates outputting content to standard error.

Next, `new FileInputStream(args[0])` provides access to the input sequence of bytes that is stored in the file identified by `args[0]`. Similarly, `new PrintStream(args[1])` provides access to the file identified by `args[1]`, which will store the output sequence of bytes, and `new PrintStream(args[2])` provides access to the file identified by `args[2]`, which will store the error sequence of bytes.

Compile Listing 12-13 (`javac RedirectIO.java`). Then execute the following command line:

```
java RedirectIO RedirectIO.java out.txt err.txt
```

This command line produces no visual output on the screen. Instead, it copies the contents of `RedirectIO.java` to `out.txt`. It also stores `Redirected error` output in `err.txt`.

Working with Writers and Readers

Java's stream classes are good for streaming sequences of bytes, but they're not good for streaming sequences of characters because bytes and characters are two different things: a byte represents an 8-bit data item and a character represents a 16-bit data item. Also, Java's `char` and `String` types naturally handle characters instead of bytes.

More importantly, byte streams have no knowledge of *character sets* (sets of mappings between integer values, known as *code points*, and symbols, such as Unicode) and their *character encodings* (mappings between the members of a character set and sequences of bytes that encode these characters for efficiency, such as UTF-8).

If you need to stream characters, you should take advantage of Java's writer and reader classes, which were designed to support character I/O (they work with `char` instead of `byte`). Furthermore, the writer and reader classes take character encodings into account.

A BRIEF HISTORY OF CHARACTER SETS AND CHARACTER ENCODINGS

Early computers and programming languages were created mainly by English-speaking programmers in countries where English was the native language. They developed a standard mapping between code points 0 through 127, and the 128 commonly used characters in the English language (such as A–Z). The resulting character set/encoding was named *American Standard Code for Information Interchange (ASCII)*.

The problem with ASCII is that it's inadequate for most non-English languages. For example, ASCII doesn't support diacritical marks such as the cedilla used in French. Because a byte can represent a maximum of 256 different characters, developers around the world started creating different character sets/encodings that encoded the 128 ASCII characters, but also

encoded extra characters to meet the needs of languages such as French, Greek, or Russian. Over the years, many legacy (and still important) data files have been created whose bytes represent characters defined by specific character sets/encodings.

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) have worked to standardize these 8-bit character sets/encodings under a joint umbrella standard called ISO/IEC 8859. The result is a series of substandards named ISO/IEC 8859-1, ISO/IEC 8859-2, and so on. For example, ISO/IEC 8859-1 (also known as Latin-1) defines a character set/encoding that consists of ASCII plus the characters covering most Western European countries. Also, ISO/IEC 8859-2 (also known as Latin-2) defines a similar character set/encoding covering Central and Eastern European countries.

Despite ISO's/IEC's best efforts, a plethora of character sets/encodings is still inadequate. For example, most character sets/encodings only allow you to create documents in a combination of English and one other language (or a small number of other languages). You cannot, for example, use an ISO/IEC character set/encoding to create a document using a combination of English, French, Turkish, Russian, and Greek characters.

This and other problems are being addressed by an international effort that has created and is continuing to develop *Unicode*, a single universal character set. Because Unicode characters are bigger than ISO/IEC characters, Unicode uses one of several variable-length encoding schemes known as *Unicode Transformation Format (UTF)* to encode Unicode characters for efficiency. For example, UTF-8 encodes every character in the Unicode character set in 1 to 4 bytes (and is backward compatible with ASCII).

The terms *character set* and *character encoding* are often used interchangeably. They mean the same thing in the context of ISO/IEC character sets in which a code point is the encoding. However, these terms are different in the context of Unicode in which Unicode is the character set and UTF-8 is one of several possible character encodings for Unicode characters.

Writer and Reader Classes Overview

The `java.io` package provides several writer and reader classes that are descendants of the abstract `Writer` and `Reader` classes. Figure 12-4 reveals the hierarchy of writer classes.

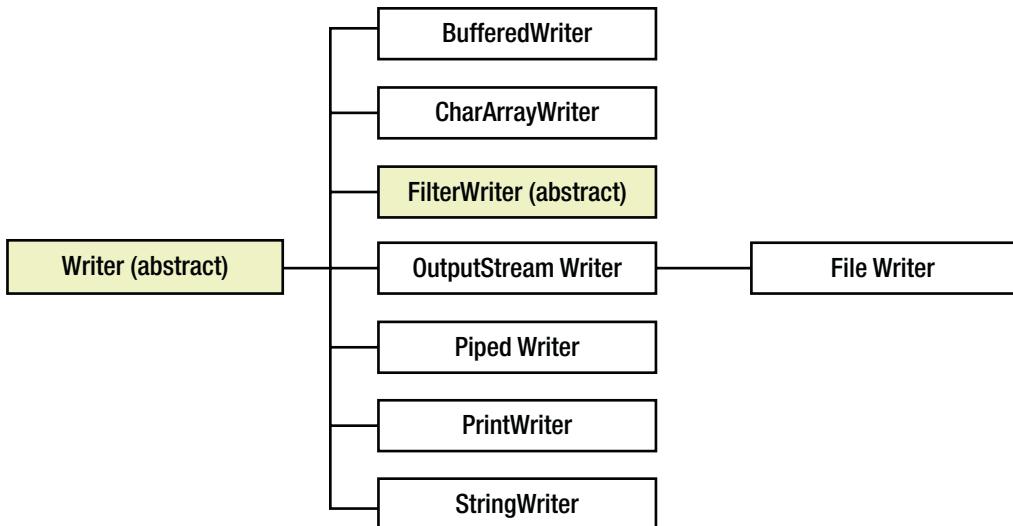


Figure 12-4. Unlike `FilterOutputStream`, `FilterWriter` is abstract

Figure 12-5 reveals the hierarchy of reader classes.

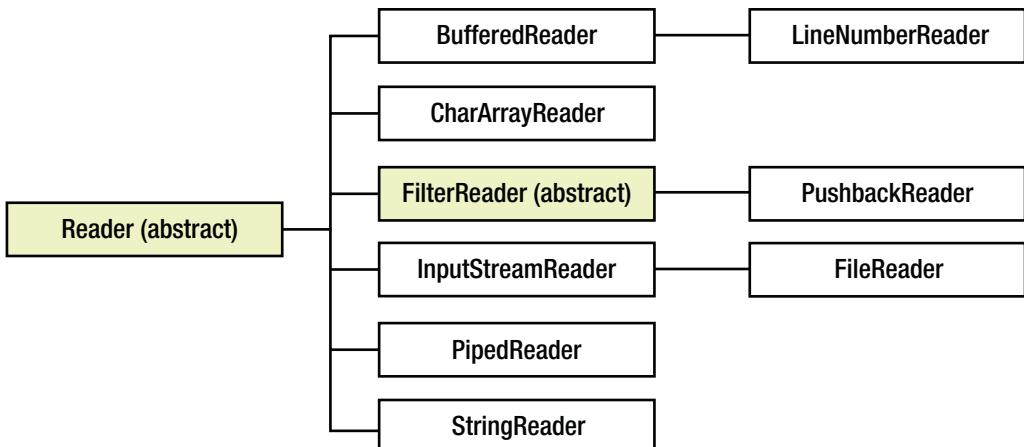


Figure 12-5. Unlike `FilterInputStream`, `FilterReader` is abstract

For brevity, we focus only on the `Writer`, `Reader`, `OutputStreamWriter`, `OutputStreamReader`, `FileWriter`, and `FileReader` classes in this chapter

Writer and Reader

Java provides the `Writer` and `Reader` classes for performing character I/O. `Writer` is the superclass of all writer subclasses. The following list identifies differences between `Writer` and `OutputStream`:

- `Writer` declares several `append()` methods for appending characters to this writer. These methods exist because `Writer` implements the `java.lang.Appendable` interface, which is used in partnership with the `Formatter` class (discussed in Chapter 14) to output formatted strings.
- `Writer` declares additional `write()` methods, including a convenient `void write(String str)` method for writing a `String` object's characters to this writer.

`Reader` is the superclass of all reader subclasses. The following list identifies differences between `Reader` and `InputStream`:

- `Reader` declares `read(char[])` and `read(char[], int, int)` methods instead of `read(byte[])` and `read(byte[], int, int)` methods.
- `Reader` doesn't declare an `available()` method.
- `Reader` declares a boolean `ready()` method that returns true when the next `read()` call is guaranteed not to block for input.
- `Reader` declares an `int read(CharBuffer target)` method for reading characters from a character buffer. (I discuss `CharBuffer` in Chapter 14.)

OutputStreamWriter and InputStreamReader

The concrete `OutputStreamWriter` class (a `Writer` subclass) is a bridge between an incoming sequence of characters and an outgoing stream of bytes. Characters written to this writer are encoded into bytes according to the default or specified character encoding.

Note The default character encoding is accessible via the `file.encoding` system property.

Each call to one of `OutputStreamWriter`'s `write()` methods causes an encoder to be called on the given character(s). The resulting bytes are accumulated in a buffer before being written to the underlying output stream. The characters passed to the `write()` methods are not buffered.

The following example uses a charset specification in the constructor to create a bridge to an underlying file output stream so that Polish text can be written to an ISO/IEC 8859-2-encoded file:

```
FileOutputStream fos = new FileOutputStream("polish.txt");
OutputStreamWriter osw = new OutputStreamWriter(fos, "8859_2");
char ch = '\u0323'; // Accented N.
osw.write(ch);
```

The concrete `InputStreamReader` class (a `Reader` subclass) is a bridge between an incoming stream of bytes and an outgoing sequence of characters. Characters read from this reader are decoded from bytes according to the default or specified character encoding.

Each call to one of `InputStreamReader`'s `read()` methods may cause one or more bytes to be read from the underlying input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation.

The following example also uses a character-encoding specification in the constructor to create a bridge to an underlying file input stream so that Polish text can be read from an ISO/IEC 8859-2-encoded file:

```
FileInputStream fis = new FileInputStream("polish.txt");
InputStreamReader isr = new InputStreamReader(fis, "8859_2");
char ch = isr.read(ch);
```

Note `OutputStreamWriter` and `InputStreamReader` declare a `String getEncoding()` method that returns the name of the character encoding in use. If the encoding has a historical name, that name is returned; otherwise, the encoding's canonical name is returned.

FileWriter and FileReader

`FileWriter` is a convenience class for writing characters to files. It subclasses `OutputStreamWriter`, and its constructors call `OutputStreamWriter(OutputStream)`. An instance of this class is equivalent to the following code fragment:

```
FileOutputStream fos = new FileOutputStream(pathname);
OutputStreamWriter osw;
osw = new OutputStreamWriter(fos, System.getProperty("file.encoding"));
```

In Chapter 5, we presented a logging library with a `File` class that didn't incorporate file-writing code. Listing 12-14 addresses this situation by presenting a revised `File` class that uses `FileWriter` to log messages to a file.

Listing 12-14. Logging Messages to an Actual File

```
package logging;

import java.io.FileWriter;
import java.io.IOException;

public class File implements Logger {
    private final static String LINE_SEPARATOR = System.getProperty("line.separator");

    private String dstName;
    private FileWriter fw;

    public File(String dstName) {
        this.dstName = dstName;
    }
```

```
public boolean connect() {  
    if (dstName == null)  
        return false;  
    try {  
        fw = new FileWriter(dstName);  
    } catch (IOException ioe) {  
        return false;  
    }  
    return true;  
}  
  
public boolean disconnect() {  
    if (fw == null)  
        return false;  
    try {  
        fw.close();  
    } catch (IOException ioe) {  
        return false;  
    }  
    return true;  
}  
  
public boolean log(String msg) {  
    if (fw == null)  
        return false;  
    try {  
        fw.write(msg + LINE_SEPARATOR);  
    } catch (IOException ioe) {  
        return false;  
    }  
    return true;  
}  
}
```

Listing 12-14 refactors Chapter 5's `File` class to support `FileWriter` by making changes to each of the `connect()`, `disconnect()`, and `log()` methods:

- `connect()` attempts to instantiate `FileWriter`, whose instance is saved in `fw` on success; otherwise, `fw` continues to store its default null reference.
- `disconnect()` attempts to close the file by calling `FileWriter`'s `close()` method, but only when `fw` doesn't contain its default null reference.
- `log()` attempts to write its `String` argument to the file by calling `FileWriter`'s `void write(String str)` method, but only when `fw` doesn't contain its default null reference.

`connect()`'s catch block specifies `IOException` instead of `FileNotFoundException` because `FileWriter`'s constructors throw `IOException` when they cannot connect to existing normal files; `FileOutputStream`'s constructors throw `FileNotFoundException`.

`log()`'s `write(String)` method appends the `line.separator` value (which we assigned to a constant for convenience) to the string being output instead of appending `\n`, which would violate portability.

`FileReader` is a convenience class for reading characters from files. It subclasses `InputStreamReader`, and its constructors call `InputStreamReader(InputStream)`. An instance of this class is equivalent to the following code fragment:

```
FileInputStream fis = new FileInputStream(pathname);
InputStreamReader isr;
isr = new InputStreamReader(fis, System.getProperty("file.encoding"));
```

EXERCISES

The following exercises are designed to test your understanding of Chapter 12's content:

1. What is the purpose of the `File` class?
2. What do instances of the `File` class contain?
3. What does `File`'s `listRoots()` method accomplish?
4. What is a path and what is a pathname?

5. What is the difference between an absolute pathname and a relative pathname?
6. How do you obtain the current user (also known as working) directory?
7. Define parent pathname.
8. File's constructors normalize their pathname arguments. What does normalize mean?
9. How do you obtain the default name-separator character?
10. What is a canonical pathname?
11. What is the difference between File's getParent() and getName() methods?
12. True or false: File's exists() method only determines whether or not a file exists.
13. What is a normal file?
14. What does File's lastModified() method return?
15. True or false: File's list() method returns an array of Strings where each entry is a file name rather than a complete path.
16. What is the difference between the FilenameFilter and FileFilter interfaces?
17. True or false: File's createNewFile() method doesn't check for file existence and create the file when it doesn't exist in a single operation that's atomic with respect to all other filesystem activities that might affect the file.
18. File's createTempFile(String, String) method creates a temporary file in the default temporary directory. How can you locate this directory?
19. Temporary files should be removed when no longer needed after an application exits (to avoid cluttering the filesystem). How do you ensure that a temporary file is removed when the virtual machine ends normally (it doesn't crash and the power isn't lost)?
20. How would you accurately compare two File objects?
21. What is the purpose of the RandomAccessFile class?

22. What is the purpose of the "rwd" and "rws" mode arguments?
23. What is a file pointer?
24. True or false: When you call `RandomAccessFile`'s `seek(long)` method to set the file pointer's value, and when this value is greater than the length of the file, the file's length changes.
25. What is a stream?
26. What is the purpose of `OutputStream`'s `flush()` method?
27. True or false: `OutputStream`'s `close()` method automatically flushes the output stream.
28. What is the purpose of `InputStream`'s `mark(int)` and `reset()` methods?
29. How would you access a copy of a `ByteArrayOutputStream` instance's internal byte array?
30. True or false: `FileOutputStream` and `FileInputStream` provide internal buffers to improve the performance of write and read operations.
31. Why would you use `PipedOutputStream` and `PipedInputStream`?
32. Define filter stream.
33. What does it mean for two streams to be chained together?
34. How do you improve the performance of a file output stream or a file input stream?
35. How do `DataOutputStream` and `DataInputStream` support `FileOutputStream` and `FileInputStream`?
36. What is object serialization and deserialization?
37. What is the purpose of the `Serializable` interface?
38. What does the serialization mechanism do when it encounters an object whose class doesn't implement `Serializable`?
39. Identify the three stated reasons for Java not supporting unlimited serialization.
40. How do you initiate serialization? How do you initiate deserialization?

41. True or false: Class fields are automatically serialized.
42. What is the purpose of the transient reserved word?
43. What does the deserialization mechanism do when it attempts to deserialize an object whose class has changed?
44. How does the deserialization mechanism detect that a serialized object's class has changed?
45. How can you add an instance field to a class and avoid trouble when deserializing an object that was serialized before the instance field was added?
46. How do you customize the default serialization and deserialization mechanisms without using externalization?
47. How do you tell the serialization and deserialization mechanisms to serialize or deserialize the object's normal state before serializing or deserializing additional data items?
48. What is the difference between PrintStream's print() and println() methods?
49. What does PrintStream's noargument void println() method accomplish?
50. Why are Java's stream classes not good at streaming characters?
51. What does Java provide as the preferred alternative to stream classes when it comes to character I/O?
52. What is the purpose of the OutputStreamWriter class? What is the purpose of the InputStreamReader class?
53. How do you identify the default character encoding?
54. What is the purpose of the FileWriter class? What is the purpose of the FileReader class?
55. Create a Java application named Touch for setting a file's or directory's timestamp to the current time. This application has the following usage syntax:
`java Touch pathname.`

56. Improve Listing 12-7's Copy application (performance wise) by using `BufferedInputStream` and `BufferedOutputStream`. Copy should read the bytes to be copied from the buffered input stream and write these bytes to the buffered output stream.
 57. Create a Java application named `Split` for splitting a large file into a number of smaller `partx` files (where `x` starts at 0 and increments, for example, `part0`, `part1`, `part2`, and so on). Each `partx` file (except possibly the last `partx` file, which holds the remaining bytes) will have the same size. This application has the following usage syntax: `java Split pathname`. Furthermore, your implementation must use the `BufferedInputStream`, `BufferedOutputStream`, `File`, `FileInputStream`, and `FileOutputStream` classes.
-

Summary

Applications often input data for processing and output processing results. Data is input from a file or some other source and is output to a file or some other destination. Java supports I/O via the classic I/O APIs located in the `java.io` package.

File I/O activities often interact with a filesystem. Java offers access to the underlying platform's available filesystem(s) via its concrete `File` class. `File` instances contain the pathnames of files and directories that may or may not exist in their filesystems.

Files can be opened for random access in which a mixture of write and read operations can occur until the file is closed. Java supports this random access by providing the concrete `RandomAccessFile` class.

Java uses streams to perform I/O operations. A stream is an ordered sequence of bytes of arbitrary length. Bytes flow over an output stream from an application to a destination and flow over an input stream from a source to an application.

The `java.io` package provides several output stream and input stream classes that are descendants of the abstract `OutputStream` and `InputStream` classes. `BufferedOutputStream` and `FileInputStream` are examples.

Java's stream classes are good for streaming sequences of bytes but are not good for streaming sequences of characters because bytes and characters are two different things, and because byte streams have no knowledge of character sets and encodings.

If you need to stream characters, you should take advantage of Java's writer and reader classes, which were designed to support character I/O (they work with `char` instead of `byte`). Furthermore, the writer and reader classes take character encodings into account.

The `java.io` package provides several writer and reader classes that are descendants of the abstract `Writer` and `Reader` classes. `FileWriter` and `FileReader` are examples. These convenience classes are based on file output/input streams and `OutputStreamWriter`/`InputStreamReader`.

This chapter focused on I/O in the context of a filesystem. However, you can also perform I/O in the context of a network. Chapter 13 introduces you to several of Java's network-oriented APIs.

CHAPTER 13

Accessing Networks

Applications often need to access networks to acquire resources (such as images) or to communicate with remote executable entities (such as web services). A *network* is a group of interconnected *nodes* (computing devices such as tablets, and peripherals such as scanners or laser printers) that can be shared among the network's users.

Note Intranets (networks within an organization) and internets often use *TCP/IP* (http://en.wikipedia.org/wiki/TCP/IP_model) to communicate between nodes. TCP/IP includes *Transmission Control Protocol (TCP)*, which is a connection-oriented protocol; *User Datagram Protocol (UDP)*, which is a connectionless protocol; and *Internet Protocol (IP)*, which is the basic protocol over which TCP and UDP perform their tasks.

The `java.net` package provides types that support TCP/IP between *processes* (executing applications) running on the same or different *hosts* (computer-based TCP/IP nodes). In this chapter, we first present the types for performing socket-based and URL-based communication. We then present the low-level network interface and interface address types and cookie-oriented types.

Accessing Networks via Sockets

Two processes communicate by way of *sockets*, which are endpoints in a communications link between these processes. Each endpoint is identified by an *IP address* that identifies the host and by a *port number* that identifies the process running on that host.

Network nodes are identified via IP addresses, and a port number specifying a communication channel identifier.

One process writes a *message* (a sequence of bytes) to its socket. The network management software portion of the underlying platform breaks the message into a sequence of *packets* (addressable message chunks that are often referred to as *IP datagrams*) and forwards them to the other process's socket where they are recombined into the original message for processing.

Figure 13-1 shows how two sockets communicate in a TCP/IP context.

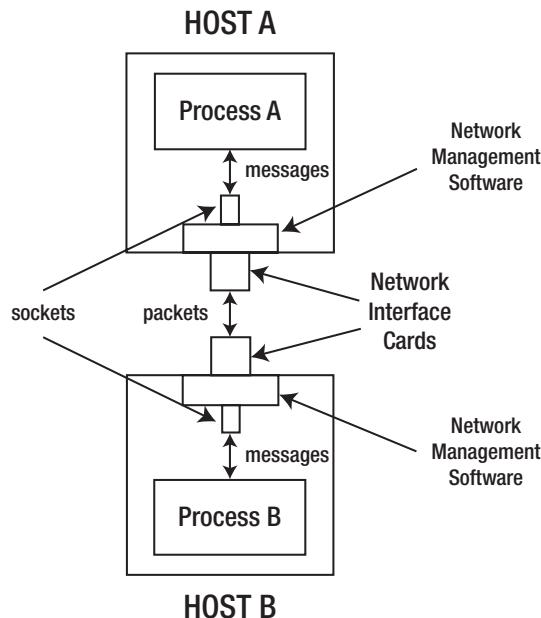


Figure 13-1. Two processes communicate via a pair of sockets

In the context of Figure 13-1, suppose that Process A wants to send a message to Process B. Process A sends that message to its socket with the destination socket address of Process B. Host A's network management software (often referred to as a *protocol stack*) obtains this message and reduces it to a sequence of packets, with each packet including the destination host's IP address and port number. The network management software then sends these packets through Host A's network interface card (NIC) to Host B.

Note The NIC's various *network interfaces* are connections between a computer and a network.

Host B's protocol stack receives packets through the NIC and reassembles them into the original message (packets may be received out of order), which it then makes available to Process B via its socket. This scenario reverses when Process B communicates with Process A.

The network management software uses TCP to create an ongoing conversation between two hosts in which messages are sent back and forth. Before this conversation occurs, a connection is established between these hosts. After the connection has been established, TCP enters a pattern where it sends message packets and waits for a reply that they arrived correctly (or for a timeout to expire when the reply doesn't arrive because of some network problem). This pattern repeats and guarantees a reliable connection. For detailed information on this pattern, check out http://en.wikipedia.org/wiki/Tcp_receive_window#Flow_control.

Because it can take time to establish a connection, and it also takes time to send packets (as it is necessary to receive reply acknowledgments and also because of timeouts), TCP is slow. On the other hand, UDP, which doesn't require connections and packet acknowledgment, is much faster. The downside is that UDP isn't as reliable (there's no guarantee of packet delivery, ordering, or protection against duplicate packets, although UDP uses checksums to verify that data is correct) because there's no acknowledgment. Furthermore, UDP is limited to single-packet conversations.

The `java.net` package provides `Socket`, `ServerSocket`, and other `Socket`-suffixed classes for performing TCP-based or UDP-based communications. Before investigating these classes, you need to understand socket addresses and socket options.

Socket Addresses

An instance of a `Socket`-suffixed class is associated with a *socket address* comprised of an IP address and a port number. These classes often rely on the `InetAddress` class to represent the IPv4 or IPv6 address portion of the socket address, and represent the port number separately.

Note `InetAddress` relies on its `Inet4Address` subclass to represent an IPv4 address and on its `Inet6Address` subclass to represent an IPv6 address.

`InetAddress` declares several class methods for obtaining an `InetAddress` instance. For details please consult the API documentation for class `InetAddress`.

The abstract class `SocketAddress` represents a socket address “with no protocol attachment.” (This class’s creator might have anticipated that Java would eventually support low-level communication protocols other than the widely popular Internet Protocol.)

`SocketAddress` is subclassed by the concrete `InetSocketAddress` class, which represents a socket address as an IP address and a port number. It can also represent a hostname and a port number and will make an attempt to resolve the hostname.

`InetSocketAddress` instances are created by invoking `InetSocketAddress(InetAddress addr, int port)` and other constructors. After an instance has been created, you can call methods such as `InetAddress getAddress()` and `int getPort()` to return socket address components.

Socket Options

An instance of a `Socket`-suffixed class shares the concept of *socket options*, which are parameters for configuring socket behavior. Socket options are described by constants that are declared in the `SocketOptions` interface.

- `IP_MULTICAST_IF` specifies the outgoing network interface for multicast packets (on *multihomed* [multiple NIC] hosts). This option isn’t implemented by Android.
- `IP_MULTICAST_IF2` specifies the outgoing network interface for multicast packets using an interface index.
- `IP_MULTICAST_LOOP` enables or disables local loopback of multicast datagrams.
- `IP_TOS` sets the type-of-service (IPv4) or traffic class (IPv6) field in the IP header for a TCP or UDP socket.
- `SO_BINDADDR` fetches the socket’s local address binding. This option isn’t implemented by Android.
- `SO_BROADCAST` enables a socket to send broadcast messages.
- `SO_KEEPALIVE` turns on socket keepalive.
- `SO_LINGER` specifies the number of seconds to wait when closing a socket when there is still some buffered data to be sent.

- `SO_OOBINLINE` enables inline reception of TCP urgent data.
- `SO_RCVBUF` sets or gets the maximum socket receive buffer size (in bytes).
- `SO_REUSEADDR` enables a socket's reuse address.
- `SO_SNDBUF` sets or gets the maximum socket send buffer size (in bytes).
- `SO_TIMEOUT` specifies a timeout (in milliseconds) on blocking accept or read/receive (but not write/send) socket operations. (Don't block forever!)
- `TCP_NODELAY` disables Nagle's algorithm (http://en.wikipedia.org/wiki/Nagle's_algorithm). Written data to the network is not buffered pending acknowledgment of previously written data.

`SocketOptions` also declares the following methods for setting and getting these options:

- `void setOption(int optID, Object value)`
- `Object getOption(int optID)`

`optID` is one of the aforementioned constants and `value` is an object of a suitable class (such as `java.lang.Boolean`).

`SocketOptions` is implemented by the abstract `SocketImpl` and `DatagramSocketImpl` classes. Concrete instances of these classes are wrapped by the various `Socket`-suffixed classes. As a result, you cannot invoke these methods. Instead, you work with the typesafe setter and getter methods provided by the `Socket`-suffixed classes for setting and getting these options.

For example, `Socket` declares `void setKeepAlive(boolean keepAlive)` for setting the `SO_KEEPALIVE` option, and `ServerSocket` declares `void setSoTimeout(int timeout)` for setting the `SO_TIMEOUT` option. Check the documentation on the `Socket`-suffixed classes to learn about these and other socket option methods.

Note `Socket` option methods that apply to `DatagramSocket` also apply to its `MulticastSocket` subclass.

Socket and ServerSocket

The `Socket` and `ServerSocket` classes support TCP-based communications between client processes (such as an application running on a tablet) and server processes (such as an application running on one of your Internet service provider's computers that provides access to the World Wide Web). Because `Socket` is associated with the `java.io.InputStream` and `java.io.OutputStream` classes, sockets based on the `Socket` class are commonly referred to as *stream sockets*.

`Socket` supports the creation of client-side sockets. It declares several constructors for this purpose; see the API documentation.

After a `Socket` instance is created, it's bound to an arbitrary local host socket address before a connection is made to the remote host socket address. Binding makes a client socket address available to a server socket so that a server process can communicate with the client process via the server socket.

After creating a `Socket` instance, and possibly invoking `bind()` and `connect()` on that instance, an application invokes `Socket`'s `InputStream getInputStream()` and `OutputStream getOutputStream()` methods to acquire an input stream for reading bytes from the socket and an output stream for writing bytes to the socket. Also, the application often calls `Socket`'s `void close()` method to close the socket when no longer needed for I/O.

The following example demonstrates how to create a socket that's bound to port number 9999 on the local host and then access its input and output streams—exceptions are ignored for brevity:

```
Socket socket = new Socket("localhost", 9999);
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
// Do some work with the socket.
socket.close();
```

`ServerSocket` supports the creation of server-side sockets. After a server socket is created, a server application enters a loop that first invokes `ServerSocket`'s `Socket accept()` method to listen for a connection request and return a `Socket` instance that lets it communicate with the associated client socket. It then communicates with the client socket to perform some kind of processing. When processing finishes, the server socket calls the client socket's `close()` method to terminate its connection with the client.

Note `ServerSocket` declares a `void close()` method for closing a server socket before terminating the server application. An unclosed socket is automatically closed when an application terminates.

The following example demonstrates how to create a server socket that's bound to port 9999 on the current host, listen for incoming connection requests, return their sockets, perform work on those sockets, and close the sockets—exceptions are ignored for brevity:

```
ServerSocket ss = new ServerSocket(9999);
while (true) {
    Socket socket = ss.accept();
    // obtain socket input/output streams and communicate with socket
    socket.close();
}
```

The `accept()` method call blocks until a connection request is available and then returns a `Socket` object so that the server application can communicate with its associated client. The socket is closed after this communication takes place. The server socket is automatically closed when the application exits.

This example assumes that socket communication takes place on the server application's main thread, which is a problem when processing takes time to perform because server response time to incoming connection requests decreases. To speed up response time, it's often necessary to communicate with the socket on a worker thread, as demonstrated in the following example:

```
ServerSocket ss = new ServerSocket(9999);
while (true) {
    final Socket s = ss.accept();
    new Thread(new Runnable() {
        @Override
        public void run() {
            // obtain socket input/output streams and communicate with
            // socket
            try { s.close(); } catch (IOException ioe) {}
        }
    })
}
```

```
}).start();
}
```

Each time a connection request arrives, `accept()` returns a `Socket` instance, and then a `java.lang.Thread` object is created whose `Runnable` accesses that socket for communicating with the socket on a worker thread.

Tip Although this example uses the `Thread` class, you could use an executor (see Chapter 11) instead.

We've created `EchoClient` and `EchoServer` applications that demonstrate `Socket` and `ServerSocket`. Listing 13-1 presents `EchoClient`'s source code.

Listing 13-1. Echoing Data to and Receiving It Back from a Server

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

import java.net.Socket;
import java.net.UnknownHostException;

public class EchoClient {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage : java EchoClient message");
            System.err.println("example: java EchoClient \"This is a test.\"");
            return;
        }
        try {
            Socket socket = new Socket("localhost", 9999);
            OutputStream os = socket.getOutputStream();
```

```

OutputStreamWriter osw = new OutputStreamWriter(os);
PrintWriter pw = new PrintWriter(osw);
pw.println(args[0]);
pw.flush();
InputStream is = socket.getInputStream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
System.out.println(br.readLine());
} catch (UnknownHostException uhe) {
    System.err.println("unknown host: " + uhe.getMessage());
} catch (IOException ioe) {
    System.err.println("I/O error: " + ioe.getMessage());
}
}
}

```

EchoClient first verifies that it has received a single command-line argument and then creates a socket that will connect to a process running on port 9999 of the local host.

After creating the socket, EchoClient obtains an output stream for writing a string to the socket. Because the output stream can only handle a sequence of bytes, the `java.io.OutputStreamWriter` and `java.io.PrintWriter` classes (see Chapter 12) are used to connect the writer that outputs characters to the byte-oriented output stream.

After instantiating `PrintWriter`, EchoClient invokes its `void println(String str)` method to write the string followed by a newline character. The `void flush()` method is subsequently called to ensure that all pending data is written to the server.

EchoClient now obtains an input stream for reading the string as a sequence of bytes. It then connects the reader (that inputs characters) to the byte-oriented input stream by instantiating `java.io.InputStreamReader` and `java.io.BufferedReader` (see Chapter 12).

Finally, EchoClient invokes `BufferedReader's String readLine()` method to read the characters followed by a newline from the socket. (`readLine()` doesn't include the newline character in the returned string.) These characters followed by a newline are then written to standard output.

Note In a long-running application, you would explicitly close the socket instance by invoking its `void close()` method when the socket is no longer needed. For brevity, we've chosen not to do so in this and most of the remaining Socket-suffixed class examples.

Listing 13-2 presents EchoServer's source code.

Listing 13-2. Receiving Data from and Echoing It Back to a Client

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer {
    public static void main(String[] args) throws IOException {
        System.out.println("Starting echo server...");
        ServerSocket ss = new ServerSocket(9999);
        while (true) {
            Socket s = ss.accept();
            try {
                InputStream is = s.getInputStream();
                InputStreamReader isr = new InputStreamReader(is);
                BufferedReader br = new BufferedReader(isr);
                String msg = br.readLine();
                System.out.println(msg);
                OutputStream os = s.getOutputStream();
                OutputStreamWriter osw = new OutputStreamWriter(os);
                PrintWriter pw = new PrintWriter(osw);
                pw.println(msg);
            }
        }
    }
}
```

```
    pw.flush();
} catch (IOException ioe) {
    System.err.println("I/O error: " + ioe.getMessage());
} finally {
    try {
        s.close();
    } catch (IOException ioe) {
    }
}
}
}
```

EchoServer first outputs an introductory message to standard output and then creates a server socket that listens for connections on port 9999. It then enters an infinite loop, where each iteration invokes ServerSocket's `Socket accept()` method to block until a connection is received and then returns a `Socket` object representing this connection.

After obtaining the socket, EchoServer obtains an input stream for reading from the socket. Because the input stream can only handle a sequence of bytes, the `InputStreamReader` and `BufferedReader` classes are used to connect the reader that inputs characters to the byte-oriented input stream.

EchoServer now obtains an output stream for writing the string as a sequence of bytes. It then connects the writer that outputs characters to the byte-oriented output stream by instantiating `OutputStreamWriter` and `PrintWriter`.

After outputting the message to standard output, EchoServer calls `flush()` to flush the output to the client. The client socket is then closed.

To experiment with these applications, copy `EchoClient.java` and `EchoServer.java` to the same directory and open two console windows with this directory being current. Compile each source file and execute `java EchoServer` in one window—you should observe an introductory message, although you might first need to enable port 9999 in case you have a *firewall* running ([http://en.wikipedia.org/wiki/Firewall_\(computing\)](http://en.wikipedia.org/wiki/Firewall_(computing))). Having started the server, echo the following command to echo text to both windows:

```
java EchoClient "This is a test."
```

You should observe “This is a test.” in both windows.

DatagramSocket and MulticastSocket

The DatagramSocket and MulticastSocket classes let you perform UDP-based communications between a pair of hosts (DatagramSocket) or between many hosts (MulticastSocket). With either class, you communicate one-way messages via *datagram packets*, which are arrays of bytes associated with instances of the DatagramPacket class.

Note Although you might think that Socket and ServerSocket are all that you need, DatagramSocket and its MulticastSocket subclass have their uses. For example, consider a scenario in which a group of machines need to occasionally tell a server that they're alive. It shouldn't matter when the occasional message is lost or even when the message doesn't arrive on time. Another example is a low-priority stock ticker that periodically broadcasts stock prices. When a packet doesn't arrive, odds are that the next packet will arrive and you'll then receive notification of the latest prices. Timely rather than reliable or orderly delivery is more important in real-time applications.

DatagramPacket declares several constructors with DatagramPacket(byte[] buf, int length) being the simplest. This constructor requires you to pass byte array and integer arguments to buf and length, where buf is a data buffer that stores data to be sent or received and length (which must be less than or equal to buf.length) specifies the number of bytes (starting at buf[0]) to send/receive.

The following example demonstrates this constructor:

```
byte[] buffer = new byte[100];
DatagramPacket dgp = new DatagramPacket(buffer, buffer.length);
```

Note Additional constructors let you specify an offset into buf that identifies the storage location of the first outgoing or incoming byte, and/or let you specify a destination socket address.

`DatagramSocket` describes a socket for the client or server side of the UDP-communication link. Although this class declares several constructors, we find it convenient in this chapter to use the `DatagramSocket()` constructor for the client side and the `DatagramSocket(int port)` constructor for the server side. Either constructor throws `SocketException` when it cannot create the datagram socket or bind the datagram socket to a local port.

After an application instantiates `DatagramSocket`, it calls `void send(DatagramPacket dgp)` and `void receive(DatagramPacket dgp)` to send and receive datagram packets.

Listing 13-3 demonstrates `DatagramPacket` and `DatagramSocket` in a server context.

Listing 13-3. Receiving Datagram Packets from and Echoing Them Back to Clients

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;

public class DGServer{
    final static int PORT = 10000;

    public static void main(String[] args) throws SocketException{
        System.out.println("Server is starting");
        DatagramSocket dgs = new DatagramSocket(PORT);
        try {
            System.out.println("Send buffer size = " + dgs.
                getSendBufferSize());
            System.out.println("Receive buffer size = " +
                dgs.getReceiveBufferSize());
            byte[] data = new byte[100];
            DatagramPacket dgp = new DatagramPacket(data, data.length);
            while (true) {
                dgs.receive(dgp);
                System.out.println(new String(data));
                dgs.send(dgp);
            }
        }
    }
}
```

```
        } catch (IOException ioe) {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}
```

Listing 13-3's main() method first creates a DatagramSocket object and binds the socket to port 10000 on the local host. It then invokes DatagramSocket's int getSendBufferSize() and int getReceiveBufferSize() methods to get the values of the SO_SNDBUF and SO_RCVBUF socket options, which are then output.

Note Sockets are associated with underlying platform send and receive buffers, and their sizes are accessed by calling getSendBufferSize() and getReceiveBufferSize(). Similarly, their sizes can be set by calling DatagramSocket's void setReceiveBufferSize(int size) and void setSendBufferSize(int size) methods. Although you can adjust these buffer sizes to improve performance, there's a practical limit with regard to UDP. The maximum size of a UDP packet that can be sent or received is 65,507 bytes under IPv4—it's derived from subtracting the 8-byte UDP header and 20-byte IP header values from 65,535. Although you can specify a send/receive buffer with a greater value, doing so is wasteful because the largest packet is restricted to 65,507 bytes. Also, attempting to send or receive a packet with a buffer length that exceeds 65,507 bytes results in IOException.

main() next instantiates DatagramPacket in preparation for receiving a datagram packet from a client and then echoing the packet back to the client. It assumes that packets will be 100 bytes or less in size.

Finally, main() enters an infinite loop that receives a packet, outputs packet content, and sends the packet back to the client—the client's addressing information is stored in DatagramPacket.

Compile Listing 13-3 (javac DGServer.java) and run the application (java DGServer). You should observe output that's the same as or similar to that shown here:

```
Server is starting
Send buffer size = 8192
Receive buffer size = 8192
```

Listing 13-4 demonstrates DatagramPacket and DatagramSocket in a client context.

Listing 13-4. Sending a Datagram Packet to and Receiving It Back from a Server

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

public class DGClient {
    final static int PORT = 10000;
    final static String ADDR = "localhost";

    public static void main(String[] args) throws SocketException {
        System.out.println("client is starting");
        DatagramSocket dgs = new DatagramSocket();
        try {
            byte[] buffer;
            buffer = "Send me a datagram".getBytes();
            InetAddress ia = InetAddress.getByName(ADDR);
            DatagramPacket dgp = new DatagramPacket(buffer, buffer.length, ia, PORT);
            dgs.send(dgp);
            byte[] buffer2 = new byte[100];
            dgp = new DatagramPacket(buffer2, buffer.length, ia, PORT);
            dgs.receive(dgp);
            System.out.println(new String(dgp.getData()));
        } catch (IOException ioe) {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}
```

Listing 13-4 is similar to Listing 13-3, but there's one big difference. We use the `DatagramPacket(byte[] buf, int length, InetAddress address, int port)` constructor to specify the server's destination, which happens to be port 10000 on the

local host, in the datagram packet. The `send()` method call routes the packet to this destination.

Compile Listing 13-4 (`javac DGClient.java`) and run the application (`java DGClient`). Assuming that `DGServer` is also running, you should observe the following output in `DGClient`'s command window (and the last line of this output in `DGServer`'s command window):

```
client is starting  
Send me a datagram
```

`MulticastSocket` describes a socket for the client or server side of a UDP-based multicasting session. Two commonly used constructors are `MulticastSocket()` (it creates a multicast socket not bound to a port) and `MulticastSocket(int port)` (it creates a multicast socket bound to the specified port). Either constructor throws `IOException` when an I/O error occurs.

WHAT IS MULTICASTING?

Previous examples have demonstrated *unicasting*, which occurs when a server sends a message to a single client. However, it's also possible to broadcast the same message to multiple clients (such as transmit a “school closed due to bad weather” announcement to all members of a group of parents who have registered with an online program to receive this announcement); this activity is known as *multicasting*.

A server multicasts by sending a sequence of datagram packets to a special IP address, which is known as a *multicast group address*, and a specific port (as specified by a port number). Clients wanting to receive those datagram packets create a multicast socket that uses that port number. They request to join the group through a *join group operation* that specifies the special IP address. At this point, the client can receive datagram packets sent to the group and can even send datagram packets to other group members. After the client has read all datagram packets that it wants to read, it removes itself from the group by applying a *leave group operation* that specifies the special IP address.

IPv4 addresses 224.0.0.1 to 239.255.255.255 (inclusive) are reserved for use as multicast group addresses.

Accessing Networks via URLs

A *Uniform Resource Locator (URL)* is a character string that specifies where a resource (such as a web page) is located on a TCP-/IP-based network (such as the Internet). Also, it provides the means to retrieve that resource. For example, `http://some.domain.tld` is a URL that locates some website's main page. The `http://` prefix specifies that *HyperText Transfer Protocol (HTTP)*, which is a high-level protocol on top of TCP/IP for locating HTTP resources (such as web pages), must be used to retrieve the web page located at `some.domain.tld`.

URN, URL, AND URI

A *Uniform Resource Name (URN)* is a character string that names a resource and doesn't provide a way to access that resource (the resource might not be available). For example, `urn:isbn:9781430264545` identifies an Apress book named *Learn Java for Android Development* and that's all.

URNs and URLs are examples of *Uniform Resource Identifiers (URIs)*, which are character strings for identifying names (URNs) and resources (URLs). Every URN and URL is also a URI.

The `java.net` package provides `URL` and `URLConnection` classes for accessing URL-based resources. It also provides `URLEncoder` and `URLDecoder` classes for encoding and decoding URLs as well as the `URI` class for performing URI-based operations (such as relativization) and returning `URL` instances containing the results.

URL and HttpURLConnection

The `URL` class represents URLs and provides access to the resources to which they refer. Each `URL` instance unambiguously identifies an Internet resource.

`URL` declares several constructors, with `URL(String s)` being the simplest. This constructor creates a `URL` instance from the `String` argument passed to `s` and is demonstrated as follows:

```
URL url = null;
try {
    url = new URL("http://example.com");
```

```

} catch (MalformedURLException murle) {
    // handle the exception
}

```

This example creates a URL object that uses HTTP to access the web page at `http://example.com`. If we specified an illegal URL (such as `foo`), the constructor would throw `MalformedURLException` (an `IOException` subclass).

Although you'll commonly specify `http://` as the protocol prefix, this isn't your only choice. For example, you can also specify `file:///` when the resource is located on the local host. Furthermore, you can prepend `jar:` to either `http://` or `file:///` when the resource is stored in a JAR file, as demonstrated here:

```
jar:file:///C:/path/to/some.jar!/some/package/SomeClass.class
```

The `jar:` prefix indicates that you want to access a JAR file resource (such as a stored class file). The `file:///` prefix identifies the local host's resource location.

The path to the JAR file is followed by an exclamation mark (!) to separate the JAR file path from the JAR resource path, which happens to be the `/some/package/SomeClass` class file entry in this JAR file (the leading `/` character is required).

Note The URL class in Oracle's Java reference implementation supports additional protocols, including `ftp`.

After creating a URL object, you can invoke various URL methods to access portions of the URL. For example, `String getProtocol()` returns the protocol portion of the URL (such as `http`). You can also retrieve the resource by calling the `InputStream openStream()` method.

`openStream()` creates a connection to the resource and returns an `InputStream` instance for reading resource data from that connection, as demonstrated here:

```

InputStream is = url.openStream();
int ch;
while ((ch = is.read()) != -1)
    System.out.print((char) ch);

```

Note For an HTTP connection, an internal socket is created that connects to HTTP port 80 on the server identified via the URL's domain name/IP address, unless you append a different port number to the domain name/IP address (such as `http://example.com:8080`).

We've created a `ListResource` application that demonstrates URL by using this class to fetch a resource and list its contents. Listing 13-5 presents `ListResource`'s source code.

Listing 13-5. Listing the Contents of the Resource Identified via a URL Command-Line Argument

```
import java.io.InputStream;
import java.io.IOException;

import java.net.MalformedURLException;
import java.net.URL;

public class ListResource {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage: java ListResource url");
            return;
        } try {
            URL url = new URL(args[0]);
            InputStream is = url.openStream();
            try {
                int ch;
                while ((ch = is.read()) != -1)
                    System.out.print((char) ch);
            } finally {
                is.close();
            }
        } catch (MalformedURLException murle) {
            System.err.println("invalid URL");
```

```

} catch (IOException ioe) {
    System.err.println("I/O error: " + ioe.getMessage());
}
}
}
}

```

`ListResource` first verifies that it has received a single command-line argument and then attempts to instantiate `URL` with this argument. Assuming that the URL is valid, which means that `MalformedURLException` isn't thrown, `ListResource` calls `openStream()` on the `URL` instance and proceeds to list the resource contents to standard output.

Compile this source code (`javac ListResource.java`) and execute the following command to list the contents of the page at `http://example.com`:

```
java ListResource http://example.com
```

`openStream()` is a convenience method for invoking `openConnection()`, `getInputStream()`. Each of `URL`'s `URLConnection` `openConnection()` and `URLConnection openConnection(Proxy proxy)` methods returns an instance of the `URLConnection` class, which represents a communications link between the application and a URL.

`URLConnection` gives you additional control over client/server communication. For example, you can use this class to output content to various resources that accept content. In contrast, `URL` only lets you input content via `openStream()`.

The following example shows you how to obtain a `URLConnection` object from the `URL` object referenced by the precreated `url` variable, enable its `doOutput` property, and obtain an output stream for writing to the resource:

```

URLConnection urlc = url.openConnection();
urlc.setDoOutput(true);
OutputStream os = urlc.getOutputStream();

```

`URLConnection` is subclassed by `HttpURLConnection` and `JarURLConnection`. These classes declare constants and/or methods that are specific to working with the HTTP protocol or interacting with JAR-based resources.

For example, `HttpURLConnection` declares `void setRequestMethod(String method)` for specifying the HTTP request command to be sent to a remote HTTP server. GET and POST are commonly specified commands.

URLEncoder and URLDecoder

HyperText Markup Language (HTML) lets you introduce forms into web pages that solicit information from page visitors. After filling out a form's fields, the visitor clicks the form's Submit button (which may specify something other than Submit), and the form content (field names and values) is sent to a server program. Before sending the form content, a web browser encodes this data by replacing spaces and other URL-illegal characters and sets the content's Internet media type (also known as Multipurpose Internet Mail Extensions [MIME] type) to `application/x-www-form-urlencoded`.

Note The data is encoded for HTTP POST and HTTP GET operations. Unlike POST, GET requires a *query string* (a ?-prefixed string containing the encoded content) to be appended to the server program's URL.

The `java.net` package provides `URLEncoder` and `URLDecoder` classes to assist you with the tasks of encoding and decoding form content.

`URLEncoder` applies the following encoding rules:

- Alphanumeric characters (a-z, A-Z, and 0-9) remain the same.
- Special characters “.”, “-”, “*”, and “_” remain the same.
- The space character “ ” is converted into a plus sign “+”.
- All other characters are unsafe and are first converted into 1 or more bytes using some encoding scheme. Each byte is then represented by the three-character string `%xy`, where `xy` is the two-digit hexadecimal representation of that byte. The recommended encoding scheme to use is UTF-8. However, for compatibility reasons, the platform's default encoding is used when an encoding isn't specified.

For example, using UTF-8 as the encoding scheme, the string "string ü@foo-bar" is converted to "string+%C3%BC%40foo-bar". In UTF-8, character ü is encoded as 2 bytes C3 (hex) and BC (hex); and character @ is encoded as 1 byte 40 (hex).

`URLEncoder` declares the following class method for encoding a string:

```
String encode(String s, String enc)
```

This method translates the `String` argument passed to `s` into `application/x-www-form-urlencoded` format using the encoding scheme specified by `enc`. It uses the supplied encoding scheme to obtain the bytes for unsafe characters and throws `java.io.UnsupportedEncodingException` when `enc`'s value isn't supported.

`URLDecoder` applies the following decoding rules:

- Alphanumeric characters (a-z, A-Z, and 0-9) remain the same.
- Special characters “.”, “-”, “*”, and “_” remain the same.
- The plus sign “+” is converted into a space character “ ”.
- A sequence of the form `%xy` will be treated as representing a byte, where `xy` is the two-digit hexadecimal representation of the 8 bits.

Then, all substrings containing one or more of these byte sequences consecutively will be replaced by the character(s) whose encoding would result in those consecutive bytes. The encoding scheme used to decode these characters may be specified; when unspecified, the platform's default encoding is used.

`URLDecoder` declares the following class method for decoding an encoded string:

```
String decode(String s, String enc)
```

This method decodes an `application/x-www-form-urlencoded` string using the encoding scheme specified by `enc`. The supplied encoding is used to determine what characters are represented by any consecutive sequences of the form `%xy`. `UnsupportedEncodingException` is thrown when `enc`'s value isn't supported.

There are two possible ways in which the decoder could deal with illegally encoded strings. It could either leave illegal characters alone or it could throw `IllegalArgumentException`. The approach the decoder takes is left to the implementation.

Note The World Wide Web Consortium recommends that UTF-8 should be used as the encoding scheme for `encode()` and `decode()`; see www.w3.org/TR/html40/appendix/notes.html#non-ascii-chars. Not doing so may introduce incompatibilities.

We've created an ED (encode/decode) application that demonstrates URLEncoder and URLDecoder in the context of the previous "string ü@foo-bar" and "string+%C3%BC%40foo-bar" example. Listing 13-6 presents the application's source code.

Listing 13-6. Encoding and Decoding an Encoded String

```
import java.io.UnsupportedEncodingException;
import java.net.URLDecoder;
import java.net.URLEncoder;

public class ED{
    public static void main(String[] args) throws
UnsupportedEncodingException {
        String encodedData = URLEncoder.encode("string ü@foo-bar", "UTF-8");
        System.out.println(encodedData);
        System.out.println(URLDecoder.decode(encodedData, "UTF-8"));
    }
}
```

When you run this application, it generates the following output:

```
string+%C3%BC%40foo-bar
string ü@foo-bar
```

Note Check out Wikipedia's “Percent-encoding” topic (<http://en.wikipedia.org/wiki/Percent-encoding>) to learn more about URL encoding (and the more accurate percent-encoding term).

URI

The URI class represents URIs (such as URNs and URLs). It doesn't provide access to a resource when the URI is a URL.

A URI instance stores a character string that conforms to the following syntax at the highest level:

[*scheme*:]*scheme-specific-part*[#*fragment*]

This syntax reveals that every URI optionally begins with a *scheme* followed by a colon character, where a scheme can be thought of as an application-level protocol for obtaining an Internet resource. However, this definition is too narrow because it implies that the URI is always a URL. A scheme can have nothing to do with resource location. For example, `urn` is the scheme for identifying URNs.

A scheme is followed by a *scheme-specific-part* that provides an instance of the scheme. For example, given the <http://tutortutor.ca> URI, `tutortutor.ca` is an instance of the `http` scheme. Scheme-specific-parts conform to the allowable syntax of their schemes and to the overall syntax structure of a URI (including what characters can be specified literally and what characters must be encoded).

A scheme concludes with an optional #-prefixed *fragment* level, which is a short string of characters that refers to a resource subordinate to another primary resource. The primary resource is identified by a URI; the fragment points to the subordinate resource. For example, `http://example.com/document.txt#line=5,10` could identify lines 5 through 10 of a text document named `document.txt` on some website.

For more details, please consult the API documentation of class `URI`.

Accessing Network Interfaces and Interface Addresses

The `NetworkInterface` class represents a network interface in terms of a name (such as `le0`) and a list of IP addresses assigned to this interface. Although a network interface is often implemented on a physical NIC, it also can be implemented in software, for example, the *loopback interface* (which is useful for testing a client).

You can use the class's methods to gather useful information about your platform's network interfaces. For example, Listing 13-7 presents an application that iterates over all network interfaces, invoking the methods that obtain the network interface's name and display name, determine if the network interface is a loopback interface, determine if the network interface is up and running, obtain the MTU, determine if the network interface supports multicasting, and enumerate all of the network interface's virtual subinterfaces.

Listing 13-7. Enumerating All Network Interfaces

```
import java.net.NetworkInterface;
import java.net.SocketException;

import java.util.Collections;
import java.util.Enumeration;

public class NetInfo {
    public static void main(String[] args) throws SocketException {
        Enumeration<NetworkInterface> eni;
        eni = NetworkInterface.getNetworkInterfaces();
        for (NetworkInterface ni: Collections.list(eni)) {
            System.out.println("Name = " + ni.getName());
            System.out.println("Display Name = " + ni.getDisplayName());
            System.out.println("Loopback = " + ni.isLoopback());
            System.out.println("Up and running = " + ni.isUp());
            System.out.println("MTU = " + ni.getMTU());
            System.out.println("Supports multicast = " +
                ni.supportsMulticast());
            System.out.println("Sub-interfaces");
            Enumeration<NetworkInterface> eni2;
            eni2 = ni.getSubInterfaces();
            for (NetworkInterface ni2: Collections.list(eni2))
                System.out.println("    " + ni2);
            System.out.println();
        }
    }
}
```

Tip The `java.util.Collections` class's `ArrayList<T> list(Enumeration<T> enumeration)` method is useful for converting a legacy enumeration to a modern array list.

Compile Listing 13-7 (`javac NetInfo.java`) and execute this application (`java NetInfo`). When we run `NetInfo` on our Windows platform, we observe information that begins with the following output:

```
Name = lo
Display Name = Software Loopback Interface 1
Loopback = true
Up and running = true
MTU = -1
Supports multicast = true
Sub-interfaces

Name = net0
Display Name = WAN Miniport (SSTP)
Loopback = false
Up and running = false
MTU = -1
Supports multicast = true
Sub-interfaces
```

The complete output reveals a different MTU size for a few network interfaces. Each size represents the maximum length of a message that can fit into an *IP datagram* without needing to fragment the message into multiple IP datagrams. This fragmentation has performance implications, especially in the context of networked games. For this reason alone, the `getMTU()` method is a valuable member of `NetworkInterface`.

The `getInterfaceAddresses()` method returns a list of `InterfaceAddress` objects, with each object containing a network interface's IP address along with broadcast address and subnet mask (IPv4) or network prefix length (IPv6).

Listing 13-8, which extends Listing 13-7 (with a few lines removed), enumerates all network interfaces, outputting their display names, and enumerates each network interface's interface addresses, outputting interface address information.

Listing 13-8. Enumerating All Network Interfaces and Interface Addresses

```
import java.net.InterfaceAddress;
import java.net.NetworkInterface;
import java.net.SocketException;
```

```
import java.util.Collections;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.List;

public class NetInfo {
    public static void main(String[] args) throws SocketException {
        Enumeration<NetworkInterface> eni;
        eni = NetworkInterface.getNetworkInterfaces();
        for (NetworkInterface ni: Collections.list(eni)) {
            System.out.println("Name = " + ni.getName());
            List<InterfaceAddress> ias = ni.getInterfaceAddresses();
            Iterator<InterfaceAddress> iter = ias.iterator();
            while (iter.hasNext())
                System.out.println(iter.next());
            System.out.println();
        }
    }
}
```

Compile Listing 13-8 (`javac NetInfo.java`) and execute this application (`java NetInfo`). When we run `NetInfo` on our Windows platform, we observe the following information:

```
Name = lo
/127.0.0.1/8 [/127.255.255.255]
/0:0:0:0:0:0:1/128 [null]
```

```
Name = net0
```

```
Name = net1
```

```
Name = net2
```

```
Name = ppp0
```

```
Name = eth0
```

```
Name = eth1
```

```
Name = eth2
Name = ppp1
Name = net3
Name = eth3
/192.xxx.xxx.xxx/xx [/192.xxx.xxx.xxx]
/fe80:0:0:0:xxxx:xxxx:xxxx:xxxx%xx/xx [null]

Name = net4
/fe80:0:0:0:0:xxxx:xxxx:xxxx%xx/xxx [null]

Name = net5
/2001:0:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx/x [null]
/fe80:0:0:0:xxxx:xxxx:xxxx:xxxx%xx/xx [null]

Name = eth4
Name = eth5
Name = eth6
Name = eth7
Name = eth8
```

Managing Cookies

Server applications commonly use *HTTP cookies* (state objects)—*cookies* for short—to persist small amounts of information on clients. For example, the identifiers of currently selected items in a shopping cart can be stored as cookies. It's preferable to store cookies on the client rather than on the server because of the potential for millions of cookies (depending on a website's popularity). In that case, not only would a server require a massive amount of storage just for cookies, but also searching for and maintaining cookies would be time-consuming.

Note Check out Wikipedia's "HTTP cookie" entry (http://en.wikipedia.org/wiki/HTTP_cookie) for a quick refresher on cookies.

A server application sends a cookie to a client as part of an HTTP response. A client (such as a web browser) sends a cookie to the server as part of an HTTP request. Before Java 5, applications worked with the `URLConnection` class (and its `HttpURLConnection` subclass) to get an HTTP response's cookies and to set an HTTP request's cookies. The `String getHeaderFieldKey(int n)` and `String getHeaderField(int n)` methods were used to access a response's Set-Cookie headers, and the `void setRequestProperty(String key, String value)` method was used to create a request's Cookie header.

Note “RFC 2109: HTTP State Management Mechanism” (www.ietf.org/rfc/rfc2109.txt) describes the Set-Cookie and Cookie headers.

Java 5 introduced the abstract `CookieHandler` class as a callback mechanism that connects HTTP state management to an HTTP protocol handler (think concrete `HttpURLConnection` subclass). An application installs a concrete `CookieHandler` subclass as the system-wide cookie handler via the `CookieHandler` class's `void setDefault(CookieHandler cHandler)` class method. A companion `CookieHandler` `getDefault()` class method returns this cookie handler, which is null when a system-wide cookie handler hasn't been installed.

An HTTP protocol handler accesses response and request headers. This handler invokes the system-wide cookie handler's `void put(URI uri, Map<String, List<String>> responseHeaders)` method to store response cookies in a cookie cache and invokes the `Map<String, List<String>> get(URI uri, Map<String, List<String>> requestHeaders)` method to fetch request cookies from this cache. Unlike Java 5, Java 6 introduced a concrete implementation of `CookieHandler` so that HTTP protocol handlers and applications can work with cookies.

The concrete `CookieManager` class extends `CookieHandler` to manage cookies. This class does not interact with the web browser's cookies (stored on the client computer). Instead, it represents a separate and distinct cookie manager.

A `CookieManager` object is initialized as follows:

- With a *cookie store* for storing cookies. The cookie store is based on the `CookieStore` interface.
- With a *cookie policy* for determining which cookies to accept for storage. The cookie policy is based on the `CookiePolicy` interface.

Create a cookie manager by calling either the `CookieManager()` constructor or the `CookieManager(CookieStore store, CookiePolicy policy)` constructor. The `CookieManager()` constructor invokes the latter constructor with `null` arguments, using the default in-memory cookie store and the default accept-cookies-from-the-original-server-only cookie policy. Unless you plan to create your own `CookieStore` and `CookiePolicy` implementations, you'll most likely work with the default constructor. The following example creates and establishes a new `CookieManager` object as the system-wide cookie handler:

```
CookieHandler.setDefault(new CookieManager());
```

In contrast to the `get()` and `put()` methods of that class, which are called by HTTP protocol handlers, an application works with the `getCookieStore()` and `setCookiePolicy()` methods. Consider Listing 13-9.

Listing 13-9. Listing All Cookies for a Specific Domain

```
import java.io.IOException;
import java.net.CookieHandler;
import java.net.CookieManager;
import java.net.CookiePolicy;
import java.net.HttpCookie;
import java.net.URL;
import java.util.List;

public class ListAllCookies {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("usage: java ListAllCookies url");
            return;
        }
        CookieManager cm = new CookieManager();
        cm.setCookiePolicy(CookiePolicy.ACCEPT_ALL);
        CookieHandler.setDefault(cm);
        new URL(args[0]).openConnection().getContent();
        List<HttpCookie> cookies = cm.getCookieStore().getCookies();
        for (HttpCookie cookie: cookies) {
```

```

        System.out.println("Name = " + cookie.getName());
        System.out.println("Value = " + cookie.getValue());
        System.out.println("Lifetime (seconds) = " + cookie.
getMaxAge());
        System.out.println("Path = " + cookie.getPath());
        System.out.println();
    }
}
}

```

Listing 13-9 describes a command-line application that obtains and lists all cookies from its single domain name argument. After creating a cookie manager and invoking `setCookiePolicy()` to set the cookie manager's policy to accept all cookies, `ListAllCookies` installs the cookie manager as the system-wide cookie handler. It next connects to the domain identified by the command-line argument and reads the content (via URL's `Object getContent()` method).

The cookie store is obtained via `getCookieStore()` and used to retrieve all nonexpired cookies via its `List<HttpCookie> getCookies()` method. For each of these `HttpCookies`, `String getName()`, `String getValue()`, and other `HttpCookie` methods are invoked to return cookie-specific information.

Compile Listing 13-9 (`javac ListAllCookies.java`). The following output resulted from invoking `java ListAllCookies http://java.dzone.com`:

```

Name = SESS374e8db54ec3033c25a586b1d093b1d1
Value = irhqtiemls4cp0vf5pe1p0oeo7
Lifetime (seconds) = 2000000
Path = /

```

Note For more information about cookie management, including examples that show you how to create your own `CookiePolicy` and `CookieStore` implementations, check out *The Java Tutorials*' “Working With Cookies” lesson (<http://docs.oracle.com/javase/tutorial/networking/cookies/index.html>).

EXERCISES

The following exercises are designed to test your understanding of Chapter 13's content:

1. Define network.
2. What is an intranet and what is an internet?
3. What do intranets and internets often use to communicate between nodes?
4. Define host.
5. What is a socket?
6. How is a socket identified?
7. Define IP address.
8. What is a packet?
9. A socket address is comprised of what elements?
10. Identify the InetAddress subclasses that are used to represent IPv4 and IPv6 addresses.
11. What is the loopback interface?
12. How is the local host represented?
13. Why are sockets based on the Socket class commonly referred to as stream sockets?
14. What does binding accomplish in the context of a Socket instance?
15. Define proxy. How does Java represent proxy settings?
16. True or false: The ServerSocket() constructor creates a bound sever socket.
17. What is the difference between the DatagramSocket and MulticastSocket classes?
18. What is the difference between unicasting and multicasting?
19. What is a URL?
20. What is a URN?
21. True or false: URLs and URNs are also URIs.

22. What is the equivalent of `openStream()`?
 23. True or false: You need to invoke `URLConnection`'s `void setDoInput(boolean doInput)` method with `true` as the argument before you can input content from a web resource.
 24. What does `URLEncoder` do when it encounters a space character?
 25. What is the purpose of the `URI` class?
 26. What does the `NetworkInterface` class accomplish?
 27. What is a MAC address?
 28. What does MTU stand for and what is its purpose?
 29. True or false: `NetworkInterface`'s `getName()` method returns a human-readable name.
 30. What does `InterfaceAddress`'s `getNetworkPrefixLength()` method return under IPv4?
 31. Define HTTP cookie.
 32. Why is it preferable to store cookies on the client rather than on the server?
 33. Identify the four `java.net` types that are used to work with cookies.
 34. Modify Listing 13-1's `EchoClient` source code to explicitly close the socket.
 35. Modify Listing 13-2's `EchoServer` source code to exit the while loop and explicitly close the server socket when a file named `kill` appears in the directory from which the server was started. After this file appears, the server will probably not die immediately because it's most likely waiting (via the `accept()` call) for an incoming client connection. However, it should die after servicing the next incoming connection.
-

Summary

A network is a group of interconnected nodes that can be shared among the network's users. An *intranet* is a network located within an organization and an *internet* is a network connecting organizations to each other. The *Internet* is the global network of networks.

The `java.net` package provides types that support TCP/IP between processes running on the same or different hosts. Two processes communicate by way of sockets, which are endpoints in a communications link between these processes. Each endpoint is identified by an IP address that identifies a host and by a port number that identifies the process running on that host.

One process writes a message to its socket, the network management software portion of the underlying operating system breaks the message into a sequence of packets that it forwards to the other process's socket, and the other process recombines received packets into the original message for its own processing.

The network management software uses TCP to create an ongoing conversation between two hosts in which messages are sent back and forth. Before this conversation occurs, a connection is established between these hosts. After the connection has been established, TCP enters a pattern where it sends message packets and waits for a reply that they arrived correctly (or for a timeout to expire when the reply doesn't arrive because of some network problem). This pattern repeats and guarantees a reliable connection.

Because it can take time to establish a connection, and it also takes time to send packets (as it is necessary to receive reply acknowledgments, and also because of timeouts), TCP is slow. On the other hand, UDP, which doesn't require connections and packet acknowledgment, is much faster. The downside is that UDP isn't as reliable (there's no guarantee of packet delivery, ordering, or protection against duplicate packets, although UDP uses checksums to verify that data is correct) because there's no acknowledgment. Furthermore, UDP is limited to single-packet conversations.

An instance of a `Socket`-suffixed class is associated with a socket address comprised of an IP address and a port number. These classes often rely on the `InetAddress` class to represent the IPv4 or IPv6 address portion of the socket address, and represent the port number separately.

An instance of a `Socket`-suffixed class shares the concept of socket options, which are parameters for configuring socket behavior. Socket options are described by constants that are declared in the `SocketOptions` interface.

The `Socket` and `ServerSocket` classes support TCP-based communications between client processes and server processes. `Socket` supports the creation of client-side sockets, whereas `ServerSocket` supports the creation of server-side sockets.

The `DatagramSocket` and `MulticastSocket` classes let you perform UDP-based communications between a pair of hosts (`DatagramSocket`) or between as many hosts as

necessary (`MulticastSocket`). With either class, you communicate one-way messages via datagram packets.

Two processes communicating via sockets demonstrate low-level network access. Java also supports high-level access via URLs that identify resources and specify where they are located on TCP-/IP-based networks.

URLs are represented by the `URL` class, which provides access to the resources to which they refer. `URLConnection` gives you additional control over client/server communication. For example, you can use this class to output content to various resources that accept content. In contrast, `URL` only lets you input content via `openStream()`.

HTML lets you introduce forms into web pages that solicit information from page visitors. The `java.net` package provides `URLEncoder` and `URLDecoder` classes to assist you with the tasks of encoding and decoding form content.

URLs are a form of URI, which is a character string that identifies a resource without providing access to the resource or identifies a name. URIs are represented by the `URI` class, which provides methods for extracting parts of a URI (such as the scheme) and for performing normalization, resolution, and relativization operations.

The `NetworkInterface` class represents a network interface in terms of a name (such as `1e0`) and a list of IP addresses assigned to this interface. `NetworkInterface`'s `getInterfaceAddresses()` method returns a list of `InterfaceAddress` objects, with each object containing a network interface's IP address along with broadcast address and subnet mask (IPv4) or network prefix length (IPv6).

Server applications commonly use HTTP cookies (state objects)—cookies, for short—to persist small amounts of information on clients. Java provides the `CookieHandler` and `CookieManager` classes and the `CookiePolicy` and `CookieStore` interfaces for working with cookies.

This chapter focused on I/O in a network context. New I/O lets you perform file-based and network-based I/O in a more performant manner. Chapter 14 introduces you to Java's new I/O APIs.

CHAPTER 14

Migrating to New I/O

Chapters 12 and 13 introduced you to Java's classic I/O APIs. Chapter 12 presented classic I/O in terms of `java.io`'s `File`, `RandomAccessFile`, `stream`, and `writer/reader` types. Chapter 13 presented classic I/O in terms of `java.net`'s `socket` and `URL` types.

Modern operating systems offer powerful I/O features that are not supported by Java's classic I/O APIs. Features include *memory-mapped file I/O* (the ability to map part of a *process* (executing application)'s *virtual memory* (see http://en.wikipedia.org/wiki/Virtual_memory) to some portion of a file so that writes to or reads from that portion of the process's memory space actually write/read the associated portion of the file), *readiness selection* (a step above nonblocking I/O that offloads to the operating system the work involved in checking for I/O stream readiness to perform write and read operations), and *file locking* (the ability for one process to prevent other processes from accessing a file or to limit the access in some way).

Java 1.4 introduced a more powerful I/O architecture that supports memory-mapped file I/O, readiness selection, file locking, and more. This architecture largely consists of buffers, channels, selectors, regular expressions, and charsets, and it is commonly known as *new I/O (NIO)*.

Note Regular expressions are included as part of NIO (see JSR 51 at <http://jcp.org/en/jsr/detail?id=51>) because NIO is all about performance, and regular expressions are useful for scanning text (read from an I/O source) in a highly performant manner.

In this chapter, we introduce you to NIO in terms of buffers, channels, selectors, regular expressions, and charsets. We also discuss the simple `printf`-style formatting facility proposed in JSR 51 but not implemented until Java 5.

Working with Buffers

NIO is based on buffers. A *buffer* is an object that stores a fixed amount of data to be sent to or received from an *I/O service* (a means for performing input/output). It sits between an application and a *channel* that writes the buffered data to the service or reads the data from the service and deposits it into the buffer.

Buffers possess four properties:

- *Capacity*: The total number of data items that can be stored in the buffer. The capacity is specified when the buffer is created and cannot be changed later.
- *Limit*: The number of “live” data items in the buffer. No items starting from the zero-based limit should be written or read.
- *Position*: The zero-based index of the next data item that can be read or the location where the data item can be written.
- *Mark*: A zero-based position that can be recalled. The mark is initially undefined.

These four properties are related as follows:

$$0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$$

Figure 14-1 reveals a newly created and byte-oriented buffer with a capacity of 7.

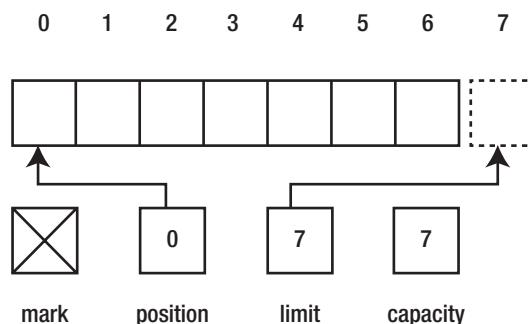


Figure 14-1. The logical layout of a byte-oriented buffer includes an undefined mark, a current position, a limit, and a capacity

Figure 14-1’s buffer can store a maximum of seven elements. The mark is initially undefined, the position is initially set to 0, and the limit is initially set to the capacity (7), which specifies the maximum number of bytes that can be stored in the buffer. You can only access positions 0 through 6. Position 7 lies beyond the buffer.

Buffer and Its Children

Buffers are implemented by classes that are derived from the abstract `java.nio.Buffer` class. The API documentation describes Buffer's methods.

Many of Buffer's methods return Buffer references so that you can chain instance method calls together. (See Chapter 3 for a discussion on instance method call chaining.) For example, instead of specifying the following three lines,

```
buf.mark();
buf.position(2);
buf.reset();
```

you can more conveniently specify the following line:

```
buf.mark().position(2).reset();
```

The documentation also shows that all buffers can be read but not all buffers can be written—for example, a buffer backed by a memory-mapped file that's read-only. You must not write to a read-only buffer; otherwise, `ReadOnlyBufferException` is thrown. Call `isReadOnly()` when you're unsure that a buffer is writable before attempting to write to that buffer.

Caution Buffers are not thread-safe. You must employ synchronization when you want to access a buffer from multiple threads.

The `java.nio` package includes several abstract classes that extend `Buffer`, one for each primitive type except for Boolean: `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, and `ShortBuffer`. Furthermore, this package includes `MappedByteBuffer` as an abstract `ByteBuffer` subclass.

Note Operating systems perform byte-oriented I/O, and you use `ByteBuffer` to create byte-oriented buffers that store the bytes to write to a destination or that are read from a source. The other primitive-type buffer classes let you create multibyte view buffers (discussed later) so that you can conceptually perform I/O in terms of characters, double-precision floating-point values, 32-bit integers, and so on. However, the I/O operation is really being carried out as a flow of bytes.

Listing 14-1 demonstrates the Buffer class in terms of ByteBuffer, capacity, limit, position, and remaining elements.

Listing 14-1. Demonstrating a Byte-Oriented Buffer

```
import java.nio.Buffer;
import java.nio.ByteBuffer;

public class BufferDemo {
    public static void main(String[] args) {
        Buffer buffer = ByteBuffer.allocate(7);
        System.out.println("Capacity: " + buffer.capacity());
        System.out.println("Limit: " + buffer.limit());
        System.out.println("Position: " + buffer.position());
        System.out.println("Remaining: " + buffer.remaining());
        System.out.println("Changing buffer limit to 5");
        buffer.limit(5);
        System.out.println("Limit: " + buffer.limit());
        System.out.println("Position: " + buffer.position());
        System.out.println("Remaining: " + buffer.remaining());
        System.out.println("Changing buffer position to 3");
        buffer.position(3);
        System.out.println("Position: " + buffer.position());
        System.out.println("Remaining: " + buffer.remaining());
        System.out.println(buffer);
    }
}
```

Listing 14-1's main() method first needs to obtain a buffer. It cannot instantiate the Buffer class because that class is abstract. Instead, it uses the ByteBuffer class and its allocate() class method to allocate the 7-byte buffer shown in Figure 14-1. main() then calls assorted Buffer methods to demonstrate capacity, limit, position, and remaining elements.

Compile Listing 14-1 as follows:

```
javac BufferDemo.java
```

Run this application as follows:

```
java BufferDemo
```

You should observe the following output:

```
Capacity: 7
Limit: 7
Position: 0
Remaining: 7
Changing buffer limit to 5
Limit: 5
Position: 0
Remaining: 5
Changing buffer position to 3
Position: 3
Remaining: 2
java.nio.HeapByteBuffer[pos=3 lim=5 cap=7]
```

The final output line reveals that the `ByteBuffer` instance assigned to `buffer` is actually an instance of the package-private `java.nio.HeapByteBuffer` class.

The documentation tells you more about the buffers' capabilities.

Working with Channels

Channels partner with buffers to achieve high-performance I/O. A *channel* is an object that represents an open connection to a hardware device, a file, a network socket, an application component, or another entity that's capable of performing write, read, and other I/O operations. Channels efficiently transfer data between byte buffers and I/O service sources or destinations.

Note Channels are the gateways through which native I/O services are accessed. Channels use byte buffers as the endpoints for sending and receiving data.

There often exists a one-to-one correspondence between an operating system file handle or file descriptor and a channel. When you work with channels in a file context, the channel will often be connected to an open file descriptor. Despite channels being more abstract than file descriptors, they are still capable of modeling an operating system's native I/O facilities.

Channel and Its Children

Java supports channels via its `java.nio.channels` and `java.nio.channels.spi` packages. Applications interact with the types located in the former package; developers who are defining new selector providers work with the latter package. (We will discuss selectors later in this chapter.)

All channels are instances of classes that ultimately implement the `java.nio.channels.Channel` interface. `Channel` declares the following methods:

- `void close()`: Closes this channel. When this channel is already closed, invoking `close()` has no effect. When another thread has already invoked `close()`, a new `close()` invocation blocks until the first invocation finishes, after which `close()` returns without effect. This method throws `IOException` when an I/O error occurs. After the channel is closed, any further attempts to invoke I/O operations upon it result in `java.nio.channels.ClosedChannelException` being thrown.
- `boolean isOpen()`: Returns this channel's open status. This method returns true when the channel is open; otherwise, it returns false.

These methods indicate that only two operations are common to all channels: close the channel and determine whether the channel is open or closed. To support I/O, `Channel` is extended by the `java.nio.channels.WritableByteChannel` and `java.nio.channels.ReadableByteChannel` interfaces:

- `WritableByteChannel` declares an abstract `int write(ByteBuffer buffer)` method that writes a sequence of bytes from `buffer` to the current channel. This method returns the number of bytes actually written. It throws `java.nio.channels.NonWritableChannelException` when the channel was not opened for writing, `java.nio.channels.`

`ClosedChannelException` when the channel is closed, `java.nio.channels.AsynchronousCloseException` when another thread closes the channel during the write, `java.nio.channels.ClosedByInterruptException` when another thread interrupts the current thread while the write operation is in progress (thereby closing the channel and setting the current thread's interrupt status), and `java.io.IOException` when some other I/O error occurs.

- `ReadableByteChannel` declares an abstract `int read(ByteBuffer buffer)` method that reads bytes from the current channel into `buffer`. This method returns the number of bytes actually read (or `-1` when there are no more bytes to read). It throws `java.nio.channels.NonReadableChannelException` when the channel was not opened for reading; `ClosedChannelException` when the channel is closed; `AsynchronousCloseException` when another thread closes the channel during the read; `ClosedByInterruptException` when another thread interrupts the current thread while the write operation is in progress, thereby closing the channel and setting the current thread's interrupt status; and `IOException` when some other I/O error occurs.

Note A channel whose class implements only `WritableByteChannel` or `ReadableByteChannel` is *unidirectional*. Attempting to read from a writable byte channel or write to a readable byte channel results in a thrown exception.

You can use the `instanceof` operator to determine if a channel instance implements either interface. Because it's somewhat awkward to test for both interfaces, Java supplies the `java.nio.channels.ByteChannel` interface, which is an empty marker interface that subtypes `WritableByteChannel` and `ReadableByteChannel`. When you need to learn whether or not a channel is *bidirectional*, it's more convenient to specify an expression such as `channel instanceof ByteChannel`.

`Channel` is also extended by the `java.nio.channels.InterruptibleChannel` interface. `InterruptibleChannel` describes a channel that can be asynchronously closed and interrupted. This interface overrides its `Channel` superinterface's `close()` method

header, presenting the following additional stipulation to Channel's contract for this method: any thread currently blocked in an I/O operation upon this channel will receive `AsynchronousCloseException` (an `IOException` descendant).

A channel that implements this interface is asynchronously *closeable*: when a thread is blocked in an I/O operation on an interruptible channel, another thread may invoke the channel's `close()` method. This causes the blocked thread to receive a thrown `AsynchronousCloseException` instance.

A channel that implements this interface is also *interruptible*: when a thread is blocked in an I/O operation on an interruptible channel, another thread may invoke the blocked thread's `interrupt()` method. Doing this causes the channel to be closed, the blocked thread to receive a thrown `ClosedByInterruptException` instance, and the blocked thread to have its interrupt status set. (When a thread's interrupt status is already set and it invokes a blocking I/O operation on a channel, the channel is closed and the thread will immediately receive a thrown `ClosedByInterruptException` instance; its interrupt status will remain set.)

NIO's designers chose to shut down a channel when a blocked thread is interrupted because they couldn't find a way to handle interrupted I/O operations reliably in the same manner across platforms. The only way to guarantee deterministic behavior was to shut down the channel.

Tip You can determine whether or not a channel supports asynchronous closing and interruption by using the `instanceof` operator in an expression such as `channel instanceof InterruptibleChannel`.

You previously learned that you must call a class method on a `Buffer` subclass to obtain a buffer. Regarding channels, there are two ways to obtain a channel:

- The `java.nio.channels` package provides a `Channels` utility class that offers two methods for obtaining channels from streams—for each of the following methods, the underlying stream is closed when the channel is closed, and the channel isn't buffered:
 - `WritableByteChannel newChannel(OutputStream outputStream)` returns a writable byte channel for the given `outputStream`.
 - `ReadableByteChannel newChannel(InputStream inputStream)` returns a readable byte channel for the given `inputStream`.

- Various classic I/O classes have been retrofitted to support channel creation. For example, `RandomAccessFile` declares a `FileChannel getChannel()` method for returning a file channel instance, and `java.net.Socket` declares a `SocketChannel getChannel()` method for returning a socket channel.

Listing 14-2 uses the `Channels` class to obtain channels for the standard input and output streams and then uses these channels to copy bytes from the input channel to the output channel.

Listing 14-2. Copying Bytes from an Input Channel to an Output Channel

```
import java.io.IOException;  
  
import java.nio.ByteBuffer;  
  
import java.nio.channels.Channels;  
import java.nio.channels.ReadableByteChannel;  
import java.nio.channels.WritableByteChannel;  
  
public class ChannelDemo {  
    public static void main(String[] args) {  
        ReadableByteChannel src = Channels.newChannel(System.in);  
        WritableByteChannel dest = Channels.newChannel(System.out);  
  
        try {  
            copy(src, dest); // or copyAlt(src, dest);  
        } catch (IOException ioe) {  
            System.err.println("I/O error: " + ioe.getMessage());  
        } finally {  
            try {  
                src.close();  
                dest.close();  
            } catch (IOException ioe) {  
            }  
        }  
    }  
}
```

```

private static void copy(ReadableByteChannel src, WritableByteChannel dest)
    throws IOException
{
    ByteBuffer buffer = ByteBuffer.allocateDirect(2048);
    while (src.read(buffer) != -1) {
        buffer.flip();
        dest.write(buffer);
        buffer.compact();
    }
    buffer.flip();
    while (buffer.hasRemaining())
        dest.write(buffer);
}

private static void copyAlt(ReadableByteChannel src, WritableByteChannel dest)
    throws IOException
{
    ByteBuffer buffer = ByteBuffer.allocateDirect(2048);
    while (src.read(buffer) != -1) {
        buffer.flip();
        while (buffer.hasRemaining())
            dest.write(buffer);
        buffer.clear();
    }
}
}

```

Listing 14-2 presents two approaches to copying bytes from the standard input stream to the standard output stream. In the first approach, which is exemplified by the `copy()` method, the goal is to minimize native I/O calls (via the `write()` method calls), although more data may end up being copied as a result of the `compact()` method calls. In the second approach, as demonstrated by `copyAlt()`, the goal is to eliminate data copying, although more native I/O calls might occur.

Each of `copy()` and `copyAlt()` first allocates a direct byte buffer (recall that a direct byte buffer is the most efficient means for performing I/O on the virtual machine) and enters a while loop that continually reads bytes from the source channel until end of input (`read()` returns -1). Following the read, the buffer is flipped so that it can be drained. Here is where the methods diverge:

- The `copy()` method while loop makes a single call to `write()`. Because `write()` might not completely drain the buffer, `compact()` is called to compact the buffer before the next read. Compaction ensures that unwritten buffer content isn't overwritten during the next read operation. Following the while loop, `copy()` flips the buffer in preparation for draining any remaining content and then works with `hasRemaining()` and `write()` to drain the buffer completely.
- The `copyAlt()` method while loop contains a nested while loop that works with `hasRemaining()` and `write()` to continue draining the buffer until the buffer is empty. This is followed by a `clear()` method call that empties the buffer so that it can be filled on the next `read()` call.

Note It's important to realize that a single `write()` method call may not output the entire content of a buffer. Similarly, a single `read()` call may not completely fill a buffer.

Compile Listing 14-2 via the following command line:

```
javac ChannelDemo.java
```

Run `ChannelDemo` via the following command lines:

```
java ChannelDemo
java ChannelDemo <ChannelDemo.java >ChannelDemo.bak
```

The first command line copies standard input to standard output. The second command line copies the contents of `ChannelDemo.java` to `ChannelDemo.bak`. After testing the `copy()` method, replace `copy(src, dest);` with `copyAlt(src, dest);` and repeat.

Working with Selectors

I/O is either block-oriented (such as file I/O) or stream-oriented (such as network I/O). Streams are often slower than block devices (such as fixed disks) and read/write operations often cause the calling thread to block until input is available or output has been fully written. To compensate, modern operating systems let streams operate in *nonblocking mode*, which makes it possible for a thread to read or write data without blocking. The operation fully succeeds, or it indicates that the read/write isn't possible at that time. Either way, the thread is able to perform other useful work instead of waiting.

Nonblocking mode doesn't let an application determine if it can perform an operation without actually performing the operation. For example, when a nonblocking read operation succeeds, the application learns that the read operation is possible but also has read some data that must be managed. This duality prevents you from separating code that checks for stream readiness from the data-processing code without making your code significantly complicated.

Nonblocking mode serves as a foundation for performing *readiness selection*, which offloads to the operating system the work involved in checking for I/O stream readiness to perform write, read, and other operations. The operating system is instructed to observe a group of streams and return some indication of which streams are ready to perform a specific operation (such as read) or operations (such as accept and read). This capability lets a thread *multiplex* a potentially huge number of active streams by using the readiness information provided by the operating system. In this way, network servers can handle large numbers of network connections; they are vastly scalable.

Note Modern operating systems make readiness selection available to applications by providing system calls such as the POSIX `select()` call.

Selectors let you achieve readiness selection in a Java context. In this section, we first introduce you to selector fundamentals and then provide a demonstration.

Selector Fundamentals

A *selector* is an object created from a subclass of the abstract `java.nio.channels.Selector` class. It maintains a set of channels, which it examines to determine which of them are ready for reading, writing, completing a connection sequence, accepting

another connection, or some combination of these tasks. The actual work is delegated to the operating system via a POSIX `select()` or similar system call.

Note The ability to check a channel without having to wait when something isn't ready (such as bytes are not available for reading) and without also having to perform the operation while checking is the key to scalability. A single thread can manage a huge number of channels, which reduces code complexity and potential threading issues.

Selectors are used with *selectable channels*, which are objects whose classes ultimately inherit from the abstract `SelectableChannel` class, which describes a channel that can be multiplexed by a selector. Socket channels, server socket channels, datagram channels, and pipe source/sink channels are selectable channels because `SocketChannel`, `ServerChannel`, `DatagramChannel`, `Pipe.SinkChannel`, and `Pipe.SourceChannel` are derived from `SelectableChannel`. In contrast, file channels are not selectable channels because `FileChannel` doesn't include `SelectableChannel` in its ancestry.

One or more previously created selectable channels are registered with a selector. Each registration returns a *key* (described by a concrete instance of the abstract `java.nio.channels.SelectionKey` class) that's a token signifying the relationship between one channel and the selector. This key keeps track of two sets of operations: *interest set* and *ready set*. The *interest set* identifies the operation categories that will be tested for readiness the next time one of the selector's selection methods is invoked. The *ready set* identifies the operation categories for which the key's channel has been found to be ready by the key's selector. When a selection method is invoked, the selector's associated keys are updated by checking all channels registered with that selector. The application can then obtain a set of keys whose channels were found ready, and iterate over these keys to service each channel that has become ready since a previous select method call.

Note A selectable channel can be registered with more than one selector. It has no knowledge of the selectors to which it's currently registered.

To work with selectors, you first need to create one. You can accomplish this task by invoking Selector's `Selector open()` class method. This method returns a Selector instance on success or throws `IOException` on failure. The following code fragment demonstrates this task:

```
Selector selector = Selector.open();
```

You can create your selectable channels before or after creating the selector. However, you must ensure that each channel is in nonblocking mode before registering the channel with the selector. You register a selectable channel with a selector by invoking either of the following `SelectableChannel` registration methods:

- `SelectionKey register(Selector sel, int ops)`
- `SelectionKey register(Selector sel, int ops, Object att)`

Each method requires that you pass a previously created selector to `sel` and a bitwise ORed combination of the following `SelectionKey` int-based constants to `ops`, which signifies the interest set:

- `OP_ACCEPT`: Operation-set bit for socket-accept operations
- `OP_CONNECT`: Operation-set bit for socket-connect operations
- `OP_READ`: Operation-set bit for read operations
- `OP_WRITE`: Operation-set bit for write operations

The second method also lets you pass an arbitrary `java.lang.Object`/subclass instance (or null) to `att`. The nonnull object is known as an *attachment*, and it is a convenient way of recognizing a given channel or attaching additional information to the channel. It's stored in the `SelectionKey` instance returned from this method.

Upon success, each method returns a `SelectionKey` instance that relates the selectable channel with the selector. Upon failure, an exception is thrown. For example, `ClosedChannelException` is thrown when the channel is closed and `IllegalBlockingModeException` is thrown when the channel hasn't been set to nonblocking mode.

The following code fragment extends the previous code fragment by configuring a previously created channel to nonblocking mode and registering the channel with the selector, whose selection methods are to test the channel for accept, read, and write readiness:

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_ACCEPT |
                                         SelectionKey.OP_READ |
                                         SelectionKey.OP_WRITE);
```

At this point, the application typically enters an infinite loop where it accomplishes the following tasks:

1. Performs a selection operation
2. Obtains the selected keys followed by an iterator over the selected keys
3. Iterates over these keys and perform channel operations

A selection operation is performed by invoking one of Selector's selection methods. For example, `int select()` performs a blocking selection operation. It doesn't return until at least one channel is selected, this selector's `wakeup()` method is invoked, or the current thread is interrupted, whichever comes first.

Note Selector also declares an `int select(long timeout)` method that doesn't return until at least one channel is selected, this selector's `wakeup()` method is invoked, the current thread is interrupted, or the `timeout` value expires, whichever comes first. Additionally, Selector declares `int selectNow()`, which is a nonblocking version of `select()`.

The `select()` method returns the number of channels that have become ready since the last time it was called. For example, if you call `select()` and it returns 1 because one channel has become ready, and if you call `select()` again and a second channel has become ready, `select()` will once again return 1. If you've not yet serviced the first ready channel, you now have two ready channels to service. However, only one channel became ready between these `select()` calls.

A set of the selected keys (the ready set) is now obtained by invoking Selector's `Set<SelectionKey> selectedKeys()` method. Invoke the set's `iterator()` method to obtain an iterator over these keys.

Finally, the application iterates over the keys. For each of the iterations, a `SelectionKey` instance is returned. Some combination of `SelectionKey`'s `boolean isAcceptable()`, `boolean isConnectable()`, `boolean isReadable()`, and `boolean`

`isWritable()` methods are called to determine if the key indicates that a channel is ready to accept a connection, finished connecting, readable, or writable.

Note The aforementioned methods offer a convenient alternative to specifying expressions such as `key.readyOps() & OP_READ != 0`. `SelectionKey`'s `int readyOps()` method returns the key's ready set. The returned set will only contain operation bits that are valid for this key's channel. For example, it never returns an operation bit that indicates that a read-only channel is ready for writing. Note that every selectable channel also declares an `int validOps()` method, which returns a bitwise ORed set of operations that are valid for the channel.

Once the application determines that a channel is ready to perform a specific operation, it can call `SelectionKey`'s `SelectableChannel channel()` method to obtain the channel and then perform work on that channel.

Note `SelectionKey` also declares a `Selector selector()` method that returns the selector for which the key was created.

When you're finished processing a channel, you must remove the key from the set of keys; the selector doesn't perform this task. The next time the channel becomes ready, the Selector will add the key to the selected key set.

The following code fragment continues from the previous code fragment and demonstrates the aforementioned tasks:

```
while (true) {  
    int numReadyChannels = selector.select();  
    if (numReadyChannels == 0)  
        continue; // There are no ready channels to process.  
  
    Set<SelectionKey> selectedKeys = selector.selectedKeys();  
    Iterator<SelectionKey> keyIterator = selectedKeys.iterator();  
  
    while (keyIterator.hasNext()) {  
        SelectionKey key = keyIterator.next();
```

```

if (key.isAcceptable()) {
    // A connection was accepted by a ServerSocketChannel.
    ServerSocketChannel server = (ServerSocketChannel) key.channel();
    SocketChannel client = server.accept();
    if (client == null) // in case accept() returns null
        continue;
    client.configureBlocking(false); // must be nonblocking
    // Register socket channel with selector for read operations.
    client.register(selector, SelectionKey.OP_READ);
} else if (key.isReadable()) {
    // A socket channel is ready for reading.
    SocketChannel client = (SocketChannel) key.channel();
    // Perform work on the socket channel.
} else if (key.isWritable()) {
    // A socket channel is ready for writing.
    SocketChannel client = (SocketChannel) key.channel();
    // Perform work on the socket channel.
}
keyIterator.remove();
}
}

```

In addition to registering the server socket channel with the selector, each incoming client socket channel is also registered with the server socket channel. When a client socket channel becomes ready for read or write operations, `key.isReadable()` or `key.isWritable()` for the associated socket channel returns true and the socket channel can be read or written.

A key represents a relationship between a selectable channel and a selector. This relationship can be terminated by invoking `SelectionKey`'s `void cancel()` method. Upon return, the key will be invalid and will have been added to its selector's canceled-key set. The key will be removed from all of the selector's key sets during the next selection operation.

When you're finished with a selector, call `Selector`'s `void close()` method. If a thread is currently blocked in one of this selector's selection methods, it's interrupted as if by invoking the selector's `wakeup()` method. Any uncanceled keys still associated with

this selector are invalidated, their channels are deregistered, and any other resources associated with this selector are released. If this selector is already closed, invoking `close()` has no effect.

Selector Demonstration

Selectors are commonly used in server applications. Listing 14-3 presents the source code to a server application that sends its local time to clients.

Listing 14-3. Serving Time to Clients

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;

public class SelectorServer {
    private final static int DEFAULT_PORT = 9999;

    private static ByteBuffer bb = ByteBuffer.allocateDirect(8);

    public static void main(String[] args) throws IOException {
        int port = DEFAULT_PORT;
        if (args.length > 0)
            port = Integer.parseInt(args[0]);
        System.out.println("Server starting ... listening on port " + port);

        ServerSocketChannel ssc = ServerSocketChannel.open();
        ServerSocket ss = ssc.socket();
        ss.bind(new InetSocketAddress(port));
        ssc.configureBlocking(false);
```

```

Selector s = Selector.open();
ssc.register(s, SelectionKey.OP_ACCEPT);

while (true) {
    int n = s.select();
    if (n == 0)
        continue;
    Iterator it = s.selectedKeys().iterator();
    while (it.hasNext()) {
        SelectionKey key = (SelectionKey) it.next();
        if (key.isAcceptable()) {
            SocketChannel sc = ((ServerSocketChannel) key.channel()).accept();
            if (sc == null)
                continue;
            System.out.println("Receiving connection");
            bb.clear();
            bb.putLong(System.currentTimeMillis());
            bb.flip();
            System.out.println("Writing current time");
            while (bb.hasRemaining())
                sc.write(bb);
            sc.close();
        }
        it.remove();
    }
}
}

```

Listing 14-3's server application consists of a `SelectorServer` class. This class allocates a direct byte buffer after this class is loaded.

When the `main()` method is executed, it first checks for a command-line argument, which is assumed to represent a port number. If no argument is specified, a default port number is used; otherwise, `main()` tries to convert it to an integer representing the port

by passing the argument to `Integer.parseInt()`. (Remember that this method throws `java.lang.NumberFormatException` when a noninteger argument is passed.)

After outputting a startup message that identifies the listening port, `main()` obtains a server socket channel followed by the underlying socket, which is bound to the specified port. The server socket channel is then configured for nonblocking mode in preparation for registering this channel with a selector.

A selector is now obtained and the server socket channel registers itself with the selector so that it can learn when the channel is ready to perform an accept operation. The returned key isn't saved because it's never canceled (and the selector is never closed).

`main()` now enters an infinite loop, first invoking the selector's `select()` method. If the server socket channel isn't ready (`select()` returns 0), the rest of the loop is skipped.

The selected keys (just one key) along with an iterator for iterating over them are now obtained and `main()` enters an inner loop to loop over these keys. Each key's `isAcceptable()` method is invoked to find out if the server socket channel is ready to perform an accept operation. If this is the case, the channel is obtained and cast to `ServerSocketChannel`, and `ServerSocketChannel`'s `accept()` method is called to accept the new connection.

To guard against the unlikely possibility of the returned `SocketChannel` instance being null (`accept()` returns null when the server socket channel is in nonblocking mode and no connection is available to be accepted), `main()` tests for this scenario and continues the loop when null is detected.

A message about receiving a connection is output, and the byte buffer is cleared in preparation for storing the local time. After this long integer has been stored in the buffer, the buffer is flipped in preparation for draining. A message about writing the current time is output and the buffer is drained. The socket channel is then closed and the key is removed from the set of keys.

Compile Listing 14-3 as follows:

```
javac SelectorServer.java
```

Run this application as follows:

```
java SelectorServer
```

You should observe the following output and the server should continue to run:

```
Server starting ... listening on port 9999
```

We need a client to exercise this server. Listing 14-4 presents the source code to a sample client application.

Listing 14-4. Receiving Time from the Server

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.util.Date;

public class SelectorClient {
    private final static int DEFAULT_PORT = 9999;

    private static ByteBuffer bb = ByteBuffer.allocateDirect(8);

    public static void main(String[] args) {
        int port = DEFAULT_PORT;
        if (args.length > 0)
            port = Integer.parseInt(args[0]);

        try {
            SocketChannel sc = SocketChannel.open();
            InetSocketAddress addr = new InetSocketAddress("localhost", port);
            sc.connect(addr);

            long time = 0;
            while (sc.read(bb) != -1) {
                bb.flip();
                while (bb.hasRemaining()){
                    time <= 8;
                    time |= bb.get() & 255;
                }
                bb.clear();
            }
            System.out.println(new Date(time));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
        sc.close();
    } catch (IOException ioe) {
        System.err.println("I/O error: " + ioe.getMessage());
    }
}
```

Listing 14-4 is much simpler than Listing 14-3 because selectors aren't used. There's no need for a selector in this simple application. You would typically use selectors in a client context when the client interacts with several servers.

There are a couple of interesting items in the source code:

- `bb.get()` returns a 32-bit integer representation of an 8-bit byte. Sign extension is used for byte values greater than 127, which are regarded as negative numbers. Because leading one bits affect the result after bitwise ORing them with `time`, they are removed by bitwise ANDing the integer with 255.
- This value in `time` is passed to the `java.util.Date(long time)` constructor when a new `Date` object is constructed. In turn, the `Date` object is passed to `System.out.println()`, which invokes `Date`'s `toString()` method to obtain a human-readable date/time string.

Compile Listing 14-4 as follows:

```
javac SelectorClient.java
```

In a second command window, run this application as follows:

```
java SelectorClient
```

You should observe output similar to the following:

```
Mon Jan 13 18:48:10 CST 2014
```

In the server command window, you should observe the following messages:

```
Receiving connection
Writing current time
```

Working with Regular Expressions

Text-processing applications often need to match text against *patterns* (character strings that concisely describe sets of strings that are considered to be matches). For example, an application might need to locate all occurrences of a specific word pattern in a text file so that it can replace those occurrences with another word. NIO includes regular expressions to help text-processing applications perform pattern matching with high performance.

Note Despite regular expressions showing up in the NIO chapter, they are extremely powerful and you can use them everywhere in your application and outside any input or output processing context.

Pattern, PatternSyntaxException, and Matcher

A *regular expression* (also known as a *regex* or *regexp*) is a string-based pattern that represents the set of strings that match this pattern. The pattern consists of literal characters and *metacharacters*, which are characters with special meanings instead of literal meanings.

The Regular Expressions API provides the `java.util.regex.Pattern` class to represent patterns via compiled regexes. Regexes are compiled for performance reasons; pattern matching via compiled regexes is much faster than if the regexes were not compiled. The API documentation tells you all about `Pattern`'s methods.

A method of particular importance is the static `compile(String)` method. You can use it to prepare a pattern string for regular expression operations. In case you have several such operations, for example, inside a loop, preparing the pattern once and then using it often is much faster compared to building the same pattern over and over again.

Finally, the `Matcher matcher(CharSequence input)` method reveals that the Regular Expressions API also provides the `Matcher` class, whose *matchers* attempt to match compiled regexes against `input` text. The method you probably use most often to obtain matchers is `Pattern.matches()`. Or you use method `find()` repeatedly to iterate over all matches.

Note A matcher finds matches in a subset of its input called the *region*. By default, the region contains all of the matcher's input. The region can be modified by calling Matcher's Matcher region(int start, int end) method (set the limits of this matcher's region) and queried by calling Matcher's int regionStart() and int regionEnd() methods.

We've created a simple application that demonstrates Pattern, PatternSyntaxException, and Matcher. Listing 14-5 presents this application's source code.

Listing 14-5. Playing with Regular Expressions

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class RegExDemo {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java RegExDemo regex input");
            return;
        }
        try {
            System.out.println("regex = " + args[0]);
            System.out.println("input = " + args[1]);
            Pattern p = Pattern.compile(args[0]);
            Matcher m = p.matcher(args[1]);
            while (m.find())
                System.out.println("Located [" + m.group() + "] starting at "
                    + m.start() + " and ending at " + (m.end() - 1));
        } catch (PatternSyntaxException pse) {
            System.err.println("Bad regex: " + pse.getMessage());
            System.err.println("Description: " + pse.getDescription());
            System.err.println("Index: " + pse.getIndex());
        }
    }
}
```

```
        System.err.println("Incorrect pattern: " + pse.getPattern());
    }
}
}
```

Compile Listing 14-5 as follows:

```
javac RegExDemo.java
```

Run this application as follows:

```
java RegExDemo ox ox
```

You'll discover the following output:

```
regex = ox
input = ox
Located [ox] starting at 0 and ending at 1
```

`find()` searches for a match by comparing regex characters with the input characters in left-to-right order and returns true because o equals o and x equals x.

Continue by executing the following command:

```
java RegExDemo box ox
```

This time, you'll discover the following output:

```
regex = box
input = ox
```

`find()` first compares regex character b with input character o. Because these characters are not equal and because there are not enough characters in the input to continue the search, `find()` doesn't output a "Located" message to indicate a match.

However, if you execute `java RegExDemo ox box`, you'll discover a match:

```
regex = ox
input = box
Located [ox] starting at 1 and ending at 2
```

The `ox` regex consists of literal characters. More sophisticated regexes combine literal characters with *metacharacters* (such as the period `[.]`) and other regex constructs.

Tip To specify a metacharacter as a literal character, precede the metacharacter with a backslash character (as in `\.`) or place the metacharacter between `\Q` and `\E` (as in `\Q.\E`). In either case, make sure to double the backslash character when the escaped metacharacter appears in a string literal, such as `"\\."` or `"\\Q.\\E"`.

The period metacharacter matches all characters except for the line terminator. For example, each of `java RegExDemo .ox box` and `java RegExDemo .ox fox` reports a match because the period matches the `b` in `box` and the `f` in `fox`.

Note Pattern recognizes the following line terminators: carriage return (`\r`), newline (line feed) (`\n`), carriage return immediately followed by newline (`\r\n`), next line (`\u0085`), line separator (`\u2028`), and paragraph separator (`\u2029`). The period metacharacter can also be made to match these line terminators by specifying the `Pattern.DOTALL` flag when calling `Pattern.compile(String, int)`.

Character Classes

A *character class* is a set of characters appearing between [and]. There are six kinds of character classes:

- A *simple character class* consists of literal characters placed side by side and matches only these characters. For example, `[abc]` consists of characters `a`, `b`, and `c`. Also, `java RegExDemo t[aiou]ck` `tack` reports a match because `a` is a member of `[aiou]`. It also reports a match when the input is `tick`, `tock`, or `tuck` because `i`, `o`, and `u` are members.
- A *negation character class* consists of a circumflex metacharacter (^), followed by literal characters placed side by side, and it matches all characters except for those in the class. For example, `[^abc]` consists of all characters except for `a`, `b`, and `c`. Also, `java RegExDemo "[^b]ox"` `box`

doesn't report a match because b isn't a member of [^b], whereas java RegExDemo "[^b]ox" fox reports a match because f is a member. (The double quotes surrounding [^b]ox are necessary on our Windows 7 platform because ^ is treated specially at the command line.)

- A *range character class* consists of successive literal characters expressed as a starting literal character, followed by the hyphen metacharacter (-), followed by an ending literal character, and matches all characters in this range. For example, [a-z] consists of all characters from a through z. Also, java RegExDemo [h-l]ouse house reports a match because h is a member of the class, whereas java RegExDemo [h-l]ouse mouse doesn't report a match because m lies outside of the range and is therefore not part of the class. You can combine multiple ranges within the same range character class by placing them side by side; for example, [A-Za-z] consists of all uppercase and lowercase Latin letters.
- A *union character class* consists of multiple nested character classes and matches all characters that belong to the resulting union. For example, [abc[u-z]] consists of characters a, b, c, u, v, w, x, y, and z. Also, java RegExDemo [[0-9][A-F][a-f]] e reports a match because e is a hexadecimal character. (We could have alternatively expressed this character class as [0-9A-Fa-f] by combining multiple ranges.)
- An *intersection character class* consists of multiple &&-separated nested character classes and matches all characters that are common to these nested character classes. For example, [a-c&&[c-f]] consists of character c, which is the only character common to [a-c] and [c-f]. Also, java RegExDemo "[aeiouy&&[y]]" y reports a match because y is common to classes [aeiouy] and [y].
- A *subtraction character class* consists of multiple &&-separated nested character classes, where at least one nested character class is a negation character class, and it matches all characters except for those indicated by the negation character class/classes. For example, [a-z&&[^x-z]] consists of characters a through w. (The square brackets surrounding ^x-z are necessary; otherwise, ^ is ignored and the resulting class consists of only x, y, and z.) Also, java RegExDemo "[a-z&&[^aeiou]]"

g reports a match because g is a consonant and only consonants belong to this class. (We're ignoring y, which is sometimes regarded as a consonant and sometimes regarded as a vowel.)

A *predefined character class* is a regex construct for a commonly specified character class. Table 14-1 identifies Pattern's predefined character classes.

Table 14-1. Predefined Character Classes

Predefined Character Class	Description
\d	Matches any digit character. \d is equivalent to [0-9].
\D	Matches any nondigit character. \D is equivalent to [^\d].
\s	Matches any whitespace character. \s is equivalent to [\t\n\x0B\f\r].
\S	Matches any nonwhitespace character. \S is equivalent to [^\s].
\w	Matches any word character. \w is equivalent to [a-zA-Z0-9].
\W	Matches any nonword character. \W is equivalent to [^\w].

For example, the following command reports a match because \w matches the word character a in abc:

```
java RegExDemo \wbc abc
```

Capturing Groups

A *capturing group* saves a match's characters for later recall during pattern matching and is expressed as a character sequence surrounded by parentheses metacharacters (and). All characters within a capturing group are treated as a unit. For example, the (Android) capturing group combines A, n, d, r, o, i, and d into a unit. It matches the Android pattern against all occurrences of Android in the input. Each match replaces the previous match's saved Android characters with the next match's Android characters.

Capturing groups can appear inside other capturing groups. For example, capturing groups (A) and (B(C)) appear inside capturing group ((A)(B(C))), and capturing group (C) appears inside capturing group (B(C)). Each nested or nonnested capturing group receives its own number, numbering starts at 1, and capturing groups are numbered

from left to right. For example, $((A)(B(C)))$ is assigned 1, (A) is assigned 2, $(B(C))$ is assigned 3, and (C) is assigned 4.

A capturing group saves its match for later recall via a *back reference*, which is a backslash character followed by a digit character denoting a capturing group number. The back reference causes the matcher to use the back reference's capturing group number to recall the capturing group's saved match and then use that match's characters to attempt a further match. The following example uses a back reference to determine if the input consists of two consecutive Android patterns:

```
java RegExDemo "(Android) \1" "Android Android"
```

RegExDemo reports a match because the matcher detects `Android`, followed by a space, followed by `Android` in the input.

Boundary Matchers and Zero-Length Matches

A *boundary matcher* is a regex construct for identifying the beginning of a line, a word boundary, the end of text, and other commonly occurring boundaries. See Table 14-2.

Table 14-2. Boundary Matchers

Boundary Matcher	Description
<code>^</code>	Matches the beginning of the line.
<code>\$</code>	Matches the end of the line.
<code>\b</code>	Matches the word boundary.
<code>\B</code>	Matches a nonword boundary.
<code>\A</code>	Matches the beginning of text.
<code>\G</code>	Matches the end of the previous match.
<code>\Z</code>	Matches the end of text except for line terminator (when present).
<code>\z</code>	Matches the end of text.

Consider the following example:

```
java RegExDemo \b\b "I think"
```

This example reports several matches, as revealed in the following output:

```
regex = \b\b
input = I think
Located [] starting at 0 and ending at -1
Located [] starting at 1 and ending at 0
Located [] starting at 2 and ending at 1
Located [] starting at 7 and ending at 6
```

This output reveals several *zero-length matches*. When a zero-length match occurs, the starting and ending indexes are equal, although the output shows the ending index to be one less than the starting index because we specified `end() - 1` in Listing 14-5 (so that a match's end index identifies a non-zero-length match's last character, not the character following the non-zero-length match's last character).

Note A zero-length match occurs in empty input text, at the beginning of input text, after the last character of input text, or between any two characters of that text. Zero-length matches are easy to identify because they always start and end at the same index position.

Quantifiers

The final regex construct we present is the *quantifier*, a numeric value implicitly or explicitly bound to a pattern. Quantifiers are categorized as greedy, reluctant, or possessive:

- A *greedy quantifier* (?*, or +) attempts to find the longest match. Specify $X?$ to find one or no occurrences of X ; X^* to find zero or more occurrences of X ; X^+ to find one or more occurrences of X ; $X\{n\}$ to find n occurrences of X ; $X\{n,\}$ to find at least n (and possibly more) occurrences of X ; and $X\{n,m\}$ to find at least n but no more than m occurrences of X .
- A *reluctant quantifier* (??*, or +?) attempts to find the shortest match. Specify $X??$ to find one or no occurrences of X ; $X^*?$ to find zero or more occurrences of X ; $X^+?$ to find one or more occurrences

of $X; X\{n\}?$ to find n occurrences of X ; $X\{n, \}$? to find at least n (and possibly more) occurrences of X ; and $X\{n, m\}?$ to find at least n but no more than m occurrences of X .

- A *possessive quantifier* ($?+$, $*+$, or $++$) is similar to a greedy quantifier except that a possessive quantifier only makes one attempt to find the longest match, whereas a greedy quantifier can make multiple attempts. Specify $X?+$ to find one or no occurrences of X ; $X*+$ to find zero or more occurrences of X ; $X++$ to find one or more occurrences of X ; $X\{n\}+$ to find n occurrences of X ; $X\{n, \}+$ to find at least n (and possibly more) occurrences of X ; and $X\{n, m\}+$ to find at least n but no more than m occurrences of X .

For an example of a greedy quantifier, execute the following command:

```
java RegExDemo .*end "wend rend end"
```

You'll discover the following output:

```
regex = .*end
input = wend rend end
Located [wend rend end] starting at 0 and ending at 12
```

The greedy quantifier ($.*$) matches the longest sequence of characters that terminates in `end`. It starts by consuming all of the input text and then is forced to back off until it discovers that the input text terminates with these characters.

For an example of a reluctant quantifier, execute the following command:

```
java RegExDemo .*?end "wend rend end"
```

You'll discover the following output:

```
regex = .*?end
input = wend rend end
Located [wend] starting at 0 and ending at 3
Located [ rend] starting at 4 and ending at 8
Located [ end] starting at 9 and ending at 12
```

The reluctant quantifier (`(.*?)`) matches the shortest sequence of characters that terminates in end. It begins by consuming nothing and then slowly consumes characters until it finds a match. It then continues until it exhausts the input text.

For an example of a possessive quantifier, execute the following command:

```
java RegExDemo .*+end "wend rend end"
```

You'll discover the following output:

```
regex = .*+end
input = wend rend end
```

The possessive quantifier (`(.*+)`) doesn't detect a match because it consumes the entire input text, leaving nothing left over to match end at the end of the regex. Unlike a greedy quantifier, a possessive quantifier doesn't back off.

While working with quantifiers, you'll probably encounter zero-length matches. For example, execute the following command:

```
java RegExDemo 1? 101101
```

You should observe the following output:

```
regex = 1?
input = 101101
Located [1] starting at 0 and ending at 0
Located [] starting at 1 and ending at 0
Located [1] starting at 2 and ending at 2
Located [1] starting at 3 and ending at 3
Located [] starting at 4 and ending at 3
Located [1] starting at 5 and ending at 5
Located [] starting at 6 and ending at 5
```

The result of this greedy quantifier is that 1 is detected at locations 0, 2, 3, and 5 in the input text and that nothing is detected (a zero-length match) at locations 1, 4, and 6.

This time, execute the following command:

```
java RegExDemo 1?? 101101
```

You should observe the following output:

```
regex = 1??  
input = 101101  
Located [] starting at 0 and ending at -1  
Located [] starting at 1 and ending at 0  
Located [] starting at 2 and ending at 1  
Located [] starting at 3 and ending at 2  
Located [] starting at 4 and ending at 3  
Located [] starting at 5 and ending at 4  
Located [] starting at 6 and ending at 5
```

This output might look surprising, but remember that a reluctant quantifier looks for the shortest match, which (in this case) is no match at all.

Finally, execute the following command:

```
java RegExDemo 1+? 101101
```

You should observe the following output:

```
regex = 1+?  
input = 101101  
Located [1] starting at 0 and ending at 0  
Located [1] starting at 2 and ending at 2  
Located [1] starting at 3 and ending at 3  
Located [1] starting at 5 and ending at 5
```

This possessive quantifier only matches the locations where 1 is detected in the input text. It doesn't perform zero-length matches.

Practical Regular Expressions

Most of the previous regex examples haven't been practical, except to help you grasp how to use the various regex constructs. In contrast, the following examples reveal a regex that matches phone numbers of the form (ddd) ddd-dddd or ddd-dddd. A single space appears between (ddd) and ddd; there's no space on either side of the hyphen.

```
java RegExDemo "(\(\d{3}\))?\s*\d{3}-\d{4}" "(800) 555-1212"
regex = (\(\d{3}\))?\s*\d{3}-\d{4}
input = (800) 555-1212
Located [(800) 555-1212] starting at 0 and ending at 13

java RegExDemo "(\(\d{3}\))?\s*\d{3}-\d{4}" 555-1212
regex = (\(\d{3}\))?\s*\d{3}-\d{4}
input = 555-1212
Located [555-1212] starting at 0 and ending at 7
```

Note To learn more about regular expressions, check out “Lesson: Regular Expressions” (<http://download.oracle.com/javase/tutorial/essential/regex/index.html>) in The Java Tutorials.

Working withCharsets

In Chapter 12, we briefly introduced the concepts of character set and character encoding. We also referred to some of the types located in the `java.nio.charset` package. In this section, we expand on these topics and explore this package in more detail. We also briefly revisit the `String` class, discussing that part of `String` that’s relevant to the discussion.

A Brief Review of the Fundamentals

Java uses Unicode to represent characters. (*Unicode* is a 16-bit character set standard [actually, more of an encoding standard because some characters are represented by multiple numeric values; each value is known as a *code point*] whose goal is to map all of the world’s significant character sets into an all-encompassing mapping.) Although Unicode makes it much easier to work with characters from different languages, it doesn’t automate everything and you often need to work with charsets. Before we dig into this topic, you should understand the following terms:

- *Character*: A meaningful symbol. For example, “\$” and “E” are characters. These symbols predate the computer era.

- *Character set:* A set of characters. For example, uppercase letters A through Z could be considered to form a character set. No numeric values are assigned to the characters in the set. There is no relationship to Unicode, ASCII, EBCDIC, or any other kind of character set standard.
- *Coded character set:* A character set where each character is assigned a unique numeric value. Standards bodies such as US-ASCII or ISO-8859-1 define mappings from characters to numeric values.
- *Character-encoding scheme:* An encoding of a coded character set's numeric values to sequences of bytes that represent these values. Some encodings are one-to-one. For example, in ASCII, character A is mapped to integer 65 and encoded as integer 65. For some other mappings, encodings are one-to-one or one-to-many. For example, UTF-8 encodes Unicode characters. Each character whose numeric value is less than 128 is encoded as a single byte to be compatible with ASCII. Other Unicode characters are encoded as 2- to 6-byte sequences. See www.ietf.org/rfc/rfc2279.txt for more information.
- *Charset:* A coded character set combined with a character-encoding scheme. Charsets are described by the abstract `java.nio.charset.CharSet` class.

Although Unicode is widely used and increasing in popularity, other character set standards are also used. Because operating systems perform I/O at the byte level, and because files store data as byte sequences, it's necessary to translate between byte sequences and the characters that are encoded into these sequences. Charset and the other classes located in the `java.nio.charset` package address this translation task.

Working withCharsets

Beginning with JDK 1.4, virtual machines were required to support a standard collection of charsets and could support additional charsets. They also support the default charset, which doesn't have to be one of the standard charsets and is obtained when the virtual machine starts running. Table 14-3 identifies and describes the standard charsets.

Table 14-3. StandardCharsets

Charset Name	Description
US-ASCII	The 7-bit ASCII that forms the American English character set. Also known as the basic Latin block in Unicode.
ISO-8859-1	The 8-bit character set used by most European languages. It's a superset of ASCII and includes most non-English European characters.
UTF-8	An 8-bit byte-oriented character encoding for Unicode. Characters are encoded in 1 to 6 bytes.
UTF-16BE	A 16-bit encoding using big-endian order for Unicode. Characters are encoded in 2 bytes with the high-order 8 bits written first.
UTF-16LE	A 16-bit encoding using little-endian order for Unicode. Characters are encoded in 2 bytes with the low-order 8 bits written first.
UTF-16	A 16-bit encoding whose endian order is determined by an optional byte-order mark.

Charset names are case insensitive and are maintained by the Internet Assigned Names Authority (IANA). The names in Table 14-3 are included in IANA's official registry.

Note The probably most often used character encoding is UTF-8.

UTF-16BE and UTF-16LE encode each character as a 2-byte sequence in big-endian or little-endian order, respectively. A decoder for a UTF-16BE- or UTF-16LE-encoded byte sequence needs to know how the byte sequence was encoded. In contrast, UTF-16 relies on a *byte-order mark (BOM)* that appears at the beginning of the sequence. If this mark is absent, decoding proceeds according to UTF-16BE (Java's native byte order). If this mark equals \uFEFF, the sequence is decoded according to UTF-16BE. If this mark equals \UFFFE, the sequence is decoded according to UTF-16LE.

Each charset name is associated with a Charset object, which you obtain by invoking one of this class's factory methods. Listing 14-6 presents an application that shows you how to use this class to obtain the default and standard charsets, which are then used to encode characters into byte sequences.

Listing 14-6. Using Charsets to Encode Characters into Byte Sequences

```
import java.nio.ByteBuffer;
import java.nio.charset.Charset;

public class CharsetDemo {
    public static void main(String[] args) {
        String msg = "façade touché";
        String[] csNames = {
            "US-ASCII",
            "ISO-8859-1",
            "UTF-8",
            "UTF-16BE",
            "UTF-16LE",
            "UTF-16"
        };
        encode(msg, Charset.defaultCharset());
        for (String csName: csNames)
            encode(msg, Charset.forName(csName));
    }

    private static void encode(String msg, Charset cs) {
        System.out.println("Charset: " + cs.toString());
        System.out.println("Message: " + msg);

        ByteBuffer buffer = cs.encode(msg);
        System.out.println("Encoded: ");

        for (int i = 0; buffer.hasRemaining(); i++) {
            int _byte = buffer.get() & 255;
            char ch = (char) _byte;
            if (Character.isWhitespace(ch) || Character.isISOControl(ch))
                ch = '\u0000';
            System.out.printf("%2d: %02x (%c)%n", i, _byte, ch);
        }
        System.out.println();
    }
}
```

[Listing 14-6](#)'s `main()` method first creates a message consisting of two French words and an array of names for the standard collection of charsets. Next, it invokes the `encode()` method to encode the message according to the default charset, which it obtains by calling `Charset`'s `defaultCharset()` factory method. Continuing, `main()` invokes `encode()` for each of the standard charsets. `Charset`'s `Charset.forName(String charsetName)` factory method is used to obtain the `Charset` instance that corresponds to `charsetName`.

Caution `forName()` throws `java.nio.charset.IllegalCharsetNameException` when the specified charset name is illegal and throws `java.nio.charset.UnsupportedCharsetException` when the desired charset isn't supported by the virtual machine.

The `encode()` method first identifies the charset and the message. It then invokes `Charset`'s `ByteBuffer encode(String s)` method to return a new `ByteBuffer` object containing the bytes that encode the characters from `s`.

`main()` next iterates over the bytes in the byte buffer, converting each byte to a character. It uses `java.lang.Character`'s `isWhitespace()` and `isISOControl()` methods to determine if the character is whitespace or a control character (neither is regarded as printable) and converts such a character to Unicode 0 (empty string). (A carriage return or newline would screw up the output, for example.)

Finally, the index of the character, its hexadecimal value, and the character itself are printed to the standard output stream. We chose to use `System.out.printf()` for this task. You'll learn about this method in the next section.

Compile Listing 14-6 as follows:

```
javac CharsetDemo.java
```

Run the application as follows:

```
java CharsetDemo
```

You should observe the following output (abbreviated):

```

Charset: windows-1252
Message: façade touché
Encoded: 0: 66 (f)    1: 61 (a)    2: e7 (ç)    ...
Charset: US-ASCII
Encoded: 0: 66 (f)    1: 61 (a)    2: 3f (?)    ...
Charset: ISO-8859-1
Encoded: 0: 66 (f)    1: 61 (a)    2: e7 (ç)    ...
Charset: UTF-8
Encoded: 0: 66 (f)    1: 61 (a)    2: c3 (Ã)    ...
Charset: UTF-16BE
Encoded: 0: 00 ( )    1: 66 (f)    2: 00 ( )    ...
Charset: UTF-16LE
Encoded: 0: 66 (f)    1: 00 ( )    2: 61 (a)    ...
Charset: UTF-16
Encoded: 0: fe (þ)    1: ff (ÿ)    2: 00 ( )

```

In addition to providing encoding methods such as the aforementioned `ByteBuffer encode(String s)` method, `Charset` provides a complementary `CharBuffer decode(ByteBuffer buffer)` decoding method. The return type is `CharBuffer` because byte sequences are decoded into characters.

Note `ByteBuffer encode(String s)` is a convenience method for specifying `CharBuffer.wrap(s)` and passing the result to the `ByteBuffer encode(CharBuffer buffer)` method.

Charsets and the String Class

The `String` class describes a string as a sequence of characters. It declares constructors that can be passed byte arrays. Because a byte array contains an encoded character sequence, a charset is required to decode them. The following is a partial list of `String` constructors that work with charsets:

- `String(byte[] data)`: Constructs a new `String` instance by decoding the specified array of bytes using the platform's default charset
- `String(byte[] data, int offset, int byteCount)`: Constructs a new `String` instance by decoding the specified subsequence of the byte array using the platform's default charset
- `String(byte[] data, String charsetName)`: Constructs a new `String` instance by decoding the specified array of bytes using the named charset

Furthermore, `String` declares methods that encode its sequence of characters into a byte array with help from the default charset or a named charset. Two of these methods are the following:

- `byte[] getBytes()`: Returns a new byte array containing the characters of this string encoded using the platform's default charset
- `byte[] getBytes(String charsetName)`: Returns a new byte array containing the characters of this string encoded using the named charset

Note that `String(byte[] data, String charsetName)` and `byte[] getBytes(String charsetName)` throw `java.io.UnsupportedEncodingException` when the charset isn't supported.

We've created a small application that demonstrates `String` and charsets. Listing 14-7 presents the source code.

Listing 14-7. Using Charsets with String

```
import java.io.UnsupportedEncodingException;

public class CharsetDemo {
    public static void main(String[] args) throws UnsupportedEncodingException {
        byte[] encodedMsg = {
            0x66, 0x61, (byte) 0xc3, (byte) 0xa7, 0x61, 0x64, 0x65, 0x20, 0x74,
            0x6f, 0x75, 0x63, 0x68, (byte) 0xc3, (byte) 0xa9
        };
        String s = new String(encodedMsg, "UTF-8");
        System.out.println(s);
    }
}
```

```

System.out.println();
byte[] bytes = s.getBytes();
for (byte _byte: bytes)
    System.out.print(Integer.toHexString(_byte & 255) + " ");
System.out.println();
}
}

```

Listing 14-7's `main()` method first creates a byte array containing a UTF-8 encoded message. It then converts this array to a `String` object via the UTF-8 charset. After outputting the resulting `String` object, it extracts this object's bytes into a new byte array and proceeds to output these bytes in hexadecimal format. As demonstrated earlier in this chapter, we bitwise AND each byte value with 255 to remove the 0xFF sign extension bytes for negative integers when the 8-bit byte integer value is converted to a 32-bit integer value. These sign extension bytes would otherwise be output.

Compile Listing 14-7 (`javac CharsetDemo.java`), and run this application (`java CharsetDemo`). You should observe the following output:

façade touché

66 61 e7 61 64 65 20 74 6f 75 63 68 e9

You might be wondering why you observe e7 instead of c3 a7 (Latin small letter c with a *cedilla* [hook or tail]) and e9 instead of c3 a9 (Latin small letter e with an acute accent). The answer is that we invoked the noargument `getBytes()` method to encode the string. This method uses the default charset, which is `windows-1252` on our platform. According to this charset, e7 is equivalent to c3 a7 and e9 is equivalent to c3 a9. The result is a shorter encoded sequence.

Working with Formatter and Scanner

The description for JSR 51 (<http://jcp.org/en/jsr/detail?id=51>) indicates that a simple `printf`-style formatting facility was proposed for inclusion in NIO. If you're familiar with the C language, you've probably worked with the `printf()` family of functions that support formatted output. You've also probably worked with the complementary `scanf()` family that support formatted input.

One feature that makes the `printf()` and `scanf()` functions useful is varargs, which lets you pass a variable number of arguments to these functions.

Working with Formatter

Java 5 introduced the `java.util.Formatter` class as an interpreter for `printf()`-style format strings. This class provides support for layout justification and alignment; common formats for numeric, string, and date/time data; and more. Commonly used Java types (such as `byte` and `BigDecimal`) are supported. Also, limited formatting customization for arbitrary user-defined types is provided through the associated `java.util.Formattable` interface and `java.util.FormattableFlags` class.

`Formatter` declares several constructors for creating `Formatter` objects. These constructors let you specify where you want formatted output to be sent. For example, `Formatter()` writes formatted output to an internal `StringBuilder` instance and `Formatter(OutputStream os)` writes formatted output to the specified output stream. You can access the destination by calling `Formatter`'s `Appendable out()` method.

Note The `java.lang.Appendable` interface describes an object to which `char` values and character sequences can be appended. Classes (such as `StringBuilder`) whose instances are to receive formatted output (via the `Formatter` class) implement `Appendable`. This interface declares methods such as `Appendable append(char c)`—append `c`'s character to this appendable. When an I/O error occurs, this method throws `IOException`.

After creating a `Formatter` object, you would call a `format()` method to format a varying number of values. For example, `Formatter format(String format, Object... args)` formats the `args` array according to the string of format specifiers passed to the `format` parameter, and it returns a reference to the invoking `Formatter` so that you can chain `format()` calls together (for convenience).

Each format specifier has one of the following syntaxes:

- `%[argument_index$][flags][width][.precision]conversion`
- `%[argument_index$][flags][width]conversion`
- `%[flags][width]conversion`

The first syntax describes a format specifier for general, character, and numeric types. The second syntax describes a format specifier for types that are used to represent dates and times. The third syntax describes a format specifier that doesn't correspond to arguments.

The optional *argument_index* is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by 1\$, the second argument is referenced by 2\$, and so on.

The optional *flags* are a set of characters that modify the output format. The set of valid flags depends on the conversion.

The optional *width* is a positive decimal integer indicating the minimum number of characters to be written to the output.

The optional *precision* is a nonnegative decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.

The required conversion depends on the syntax. For the first syntax, it's a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type. For the second syntax, it's a two-character sequence. The first character is t or T. The second character indicates the format to be used. For the third syntax, it's a character indicating content to be inserted in the output.

Conversions are divided into six categories: general, character, numeric (integer or floating-point), date/time, percent, and line separator. The following list identifies a few example format specifiers and their conversions:

- %d: Formats argument as a decimal integer
- %x: Formats argument as a hexadecimal integer
- %c: Formats argument as a character
- %f: Formats argument as a decimal number
- %s: Formats argument as a string
- %n: Outputs a platform-specific line separator
- %10.2f: Formats argument as a decimal number with 10 as the minimum number of characters to be written (leading spaces are written when the number is smaller than the width) and 2 as the number of characters to be written after the decimal point

- %05d: Formats argument as a decimal integer with 5 as the minimum number of characters to be written (leading 0s are written when the number is smaller than the width)

When you're finished with the formatter, you might want to invoke the `void flush()` method to ensure that any buffered output in the destination is written to the underlying stream. You would typically invoke `flush()` when the destination is a file.

Continuing, invoke the formatter's `void close()` method. In addition to closing the formatter, this method also closes the underlying output destination when this destination's class implements the `java.io.Closeable` interface. If the formatter has been closed, this method has no effect. Attempting to format after calling `close()` results in `java.util.FormatterClosedException`.

[Listing 14-8](#) provides a simple demonstration of `Formatter` using the aforementioned format specifiers.

Listing 14-8. Demonstrating the `Formatter` Class

```
import java.util.Formatter;

public class FormatterDemo {
    public static void main(String[] args) {
        Formatter formatter = new Formatter();
        formatter.format("%d", 123);
        System.out.println(formatter.toString());
        formatter.format("%x", 123);
        System.out.println(formatter.toString());
        formatter.format("%c", 'X');
        System.out.println(formatter.toString());
        formatter.format("%f", 0.1);
        System.out.println(formatter.toString());
        formatter.format("%s%n", "Hello, World");
        System.out.println(formatter.toString());
        formatter.format("%10.2f", 98.375);
        System.out.println(formatter.toString());
        formatter.format("%05d", 123);
        System.out.println(formatter.toString());
        formatter.format("%1$d %1$d", 123);
```

```
        System.out.println(formatter.toString());
        formatter.format("%d %d", 123);
        System.out.println(formatter.toString());
        formatter.close();
    }
}
```

Listing 14-8's `main()` method first creates a `Formatter` object via the `Formatter()` constructor, which sends formatted output to a `StringBuilder` instance. It then demonstrates the aforementioned format specifiers by invoking a `format()` method, followed by the `toString()` method to obtain the formatted content, which is subsequently output.

The `formatter.format("%1$d %1$d", 123);` method call accesses the single data item argument to be formatted (123) twice by referencing this argument via `1$`. Without this reference, which is demonstrated via `formatter.format("%d %d", 123);`, an exception would be thrown because there must be a separate argument for each format specifier unless you use an argument index.

Lastly, the `formatter` is closed.

Compile Listing 14-8 as follows:

```
javac FormatterDemo.java
```

Run this application as follows:

```
java FormatterDemo
```

You should observe the following output:

```
123
1237b
1237bX
1237bX0.100000
1237bX0.100000Hello, World
1237bX0.100000Hello, World
      98.38
1237bX0.100000Hello, World
      98.3800123
```

```
1237bX0.100000Hello, World  
98.3800123123 123  
Exception in thread "main" java.util.MissingFormatArgumentException:  
Format specifier 'd'
```

The first thing to notice about the output is that each `format()` call appends formatted output to the previously formatted output. The second thing to notice is that `java.util.MissingFormatArgumentException` is thrown when you don't specify a needed argument.

Note `MissingFormatArgumentException` is one of several formatter exception types. These exception types subtype the `java.util.IllegalFormatException` type.

If you aren't happy with this concatenated output, there are two ways to solve the problem:

- Instantiate a new `Formatter` instance, as in `formatter = new Formatter();`, before calling `format()`. This ensures that a new default and empty string builder is created.
- Create your own `StringBuilder` instance and pass it to a constructor such as `Formatter(Appendable a)`. After outputting the formatted content, invoke `StringBuilder's void setLength(int newLength)` method with 0 as the argument to erase previous content.

It's cumbersome to have to create and manage a `Formatter` object when all you want to do is to achieve something equivalent to the C language's `printf()` function. Java addresses this situation by adding formatter support to the `java.io.PrintStream` class.

Of the various formatter-oriented methods added to `PrintStream`, you'll often invoke `PrintStream printf(String format, Object... args)`. After sending its formatted content to the print stream, this method returns a reference to this stream so that you can chain method calls together.

Listing 14-9 provides a small `printf()` demonstration.

Listing 14-9. Formatting via printf()

```
public class FormatterDemo {
    public static void main(String[] args) {
        System.out.printf("%04X%n", 478);
        System.out.printf("Current date: %1$tb %1$te, %1$tY%n",
                           System.currentTimeMillis());
    }
}
```

Listing 14-9’s `main()` method invokes `System.out.printf()` twice. The first invocation formats 32-bit integer 478 into a four-digit hexadecimal string with a leading zero and uppercase hexadecimal digits. The second invocation formats the current millisecond value returned from `System.currentTimeMillis()` into a date. The `tb` conversion specifies an abbreviated month name (such as Jan), the `te` conversion specifies the day of the month (such as 1 through 31), and the `tY` conversion specifies the year (formatted with at least four digits, with leading 0s as necessary).

Compile Listing 14-9 (`javac FormatterDemo.java`), and run the application (`java FormatterDemo`). You should observe output similar to the output shown here:

```
01DE
Current date: Jan 14, 2014
```

Note For more information on `Formatter` and its supported format specifiers, we refer you to `Formatter`’s Java documentation. You might also want to check out the documentation on the `Formattable` interface and `FormattableFlags` class to learn about customizing `Formatter`.

Working with Scanner

Java 5 introduced the `java.util.Scanner` class to parse input characters into primitive types, strings, and big integers/big decimals with the help of regular expressions. `Scanner` declares several constructors for scanning content originating from diverse

sources. For example, `Scanner(InputStream source)` creates a scanner for scanning the specified input stream, whereas `Scanner(String source)` creates a scanner for scanning the specified string.

A `Scanner` instance uses a *delimiter pattern*, which matches whitespace by default, to break its input into discrete values. After creating this instance, you can call one of the “`hasNext`” methods to verify that an anticipated character sequence is present for scanning. For example, you could call `boolean hasNextDouble()` to determine whether or not the next sequence of characters can be scanned into a double-precision floating-point value.

When the value is present, you would call the appropriate “`next`” method to scan the value. For example, you would call `double nextDouble()` to scan this sequence and return a `double` containing its value.

When you’re finished with the scanner, invoke its `void close()` method. Beyond closing the scanner, this method also closes the underlying input source when this source’s class implements the `Closeable` interface. If the scanner has been closed, this method has no effect. Any attempt to scan after calling this method will result in `IllegalStateException`.

The following example shows you how to create a scanner for scanning values from standard input and then scanning an integer followed by a double-precision floating-point value:

```
Scanner sc = new Scanner(System.in);
if (sc.hasNextInt())
    i = sc.nextInt();
if (sc.hasNextDouble())
    d = sc.nextDouble();
```

EXERCISES

The following exercises are designed to test your understanding of Chapter 14’s content:

1. Define new I/O.
2. What is a buffer?
3. Identify a buffer’s four properties.

4. What happens when you invoke Buffer's array() method on a buffer backed by a read-only array?
5. What happens when you invoke Buffer's flip() method on a buffer?
6. What happens when you invoke Buffer's reset() method on a buffer where a mark has not been set?
7. True or false: Buffers are thread-safe.
8. How do you create a byte buffer?
9. Define view buffer.
10. How is a view buffer created?
11. How do you create a read-only view buffer?
12. Identify ByteBuffer's methods for storing a single byte in a byte buffer and fetching a single byte from a byte buffer.
13. What causes BufferOverflowException or BufferUnderflowException to occur?
14. True or false: Calling flip() twice returns you to the original state.
15. What is the difference between Buffer's clear() and reset() methods?
16. What does ByteBuffer's compact() method accomplish?
17. What is the purpose of the ByteOrder class?
18. Define direct byte buffer.
19. How do you obtain a direct byte buffer?
20. What is a channel?
21. What capabilities does the Channel interface provide?
22. True or false: A channel that implements InterruptibleChannel is asynchronously closeable.
23. Define file channel.
24. Define exclusive lock and shared lock.
25. What is the fundamental difference between FileChannel's lock() and tryLock() methods?

26. What does the `FileLock lock()` method do when either a lock is already held that overlaps this lock request or another thread is waiting to acquire a lock that will overlap with this request?
27. Specify the pattern that you should adopt to ensure that an acquired file lock is always released.
28. What method does `FileChannel` provide for mapping a region of a file into memory?
29. True or false: Socket channels are selectable and can function in nonblocking mode.
30. True or false: Datagram channels are not thread-safe.
31. Why do socket channels support nonblocking mode?
32. How would you obtain a socket channel's associated socket?
33. How do you obtain a server socket channel?
34. Define selector.
35. Define regular expression.
36. What does the `Pattern` class accomplish?
37. What do `Pattern`'s `compile()` methods do when they discover illegal syntax in their regular expression arguments?
38. What does the `Matcher` class accomplish?
39. What is the difference between `Matcher`'s `matches()` and `lookingAt()` methods?
40. Define character class.
41. Identify the various kinds of character classes.
42. Define capturing group.
43. What is a zero-length match?
44. Define quantifier.
45. What is the difference between a greedy quantifier and a reluctant quantifier?
46. How do possessive and greedy quantifiers differ?

47. Identify the two main classes that contribute to the NIO printf-style formatting facility.
 48. What does the %n format specifier accomplish?
 49. Refactor Listing 12–7 (Chapter 12’s Copy application) to use the ByteBuffer and FileChannel classes in partnership with FileInputStream and FileOutputStream.
 50. Create a ReplaceText application that takes input text, a pattern that specifies text to replace, and replacement text command-line arguments, and uses Matcher’s String replaceAll(String replacement) method to replace all matches of the pattern with the replacement text (passed to replacement). For example, java ReplaceText "too many embedded spaces" "\s+ " " should output too many embedded spaces with only a single space character between successive words.
-

Summary

Java 1.4 introduced a more powerful I/O architecture that supports memory-mapped file I/O, readiness selection, file locking, and more. This architecture largely consists of buffers, channels, selectors, regular expressions, and charsets, and it is commonly known as new I/O (NIO).

NIO is based on buffers. A buffer is an object that stores a fixed amount of data to be sent to or received from an I/O service (a means for performing input/output). It sits between an application and a channel that writes the buffered data to the service or reads the data from the service and deposits it into the buffer. Java supports buffers by providing the Buffer class, assorted subclasses, and the ByteOrder type-safe enumeration.

Channels partner with buffers to achieve high-performance I/O. A channel is an object that represents an open connection to a hardware device, a file, a network socket, an application component, or another entity that’s capable of performing write, read, and other I/O operations. Channels efficiently transfer data between byte buffers and I/O service sources or destinations. Java supports channels by providing the Channel interface and related types.

Selectors let you achieve readiness selection in a Java context. Readiness selection offloads to the operating system the work involved in checking for I/O stream readiness

to perform write, read, and other operations. The operating system is instructed to observe a group of channels and return some indication of which channels are ready to perform a specific operation (such as read) or operations (such as accept and read). This capability lets a thread multiplex a potentially huge number of active channels by using the readiness information provided by the operating system. In this way, network servers can handle large numbers of network connections; they are vastly scalable. Java supports selectors by offering the `SelectableChannel`, `SelectionKey`, and `Selector` classes.

Text-processing applications often need to match text against patterns (character strings that concisely describe sets of strings that are considered to be matches). For example, an application might need to locate all occurrences of a specific word pattern in a text file so that it can replace those occurrences with another word. NIO includes regular expressions to help text-processing applications perform pattern matching with high performance. Java supports regular expressions by providing the `Pattern` and `Matcher` classes.

Charsets combine coded character sets with character-encoding schemes. They're used to translate between byte sequences and the characters that are encoded into these sequences. Java supports charsets by providing `CharSet` and related classes.

The description for JSR 51 (the NIO JSR) indicates that a simple `printf`-style formatting facility was proposed for inclusion in NIO. This facility consists of formatted output and formatted input.

Chapter 15 focuses on database access. You first encounter the Apache Derby and SQLite database products and then learn how to use the JDBC API to create/access their databases.

CHAPTER 15

Accessing Databases

Applications often need to access databases to store and retrieve various kinds of data. A *database* (<http://en.wikipedia.org/wiki/Database>) is an organized collection of data. Although there are many kinds of databases (such as hierarchical, object-oriented, and relational), *relational databases*, which organize data into tables that can be related to each other, are common.

Note In a relational database, each row stores a single item (such as an employee) and each column stores a single item attribute (such as an employee's name).

Except for the most trivial of databases (such as a flat file database based on a single data file), databases are created and managed through a *database management system (DBMS)*—see http://en.wikipedia.org/wiki/Database_management_system. Relational DBMSs (RDBMSs) support *Structured Query Language (SQL)* for working with tables and more.

Note For brevity, we assume that you're familiar with SQL. If not, you might want to check out Wikipedia's “SQL” entry (<http://en.wikipedia.org/wiki/SQL>) for an introduction.

Java supports database access and creation (and more) via its relational database-oriented JDBC (Java Database Connectivity) API. Because you need an RDBMS before you can explore JDBC, this chapter first introduces you to Apache Derby (formerly Java

DB and up to JRE 7 included with the JDK), followed by the popular SQLite (<http://en.wikipedia.org/wiki/Sqlite>). SQLite is also included with Android. This chapter then focuses on JDBC.

Note Android offers an alternative to JDBC via its `android.database` and `android.database.sqlite` packages, which are the preferred means for accessing databases from an Android application. Although Android supports JDBC by including this API and an undocumented JDBC driver (we discuss JDBC drivers later in this chapter), you should focus on using Android's database access alternative when developing an Android application that requires database access. Because you still might find JDBC useful, especially when creating a non-Android application, we present JDBC in this chapter.

Introducing Apache Derby

First introduced by Sun Microsystems as part of JDK 6 to give developers an RDBMS to test their JDBC code, a DBMS named *Java DB* was a distribution of Apache's open source Derby product, which was based on IBM's Cloudscape RDBMS code base. This pure-Java RDBMS was also bundled with JDK 7. Beginning with JDK 8, Java DB was removed from the JDK and also returned to the Apache Derby team. It's secure, supports JDBC and SQL (including transactions, stored procedures, and concurrency), and has a small footprint—its core engine and JDBC driver occupy approximately 3.5MB.

Note A *JDBC driver* is a class file plug-in for communicating with a database. We'll have more to say about JDBC drivers when we introduce JDBC later in this chapter.

Apache Derby is capable of running in an embedded environment or in a client/server environment. In an embedded environment, where an application accesses the database engine via Java DB's *embedded driver*, the database engine runs in the same virtual machine as the application. Figure 15-1 illustrates the embedded environment architecture, where the database engine is embedded in the application.

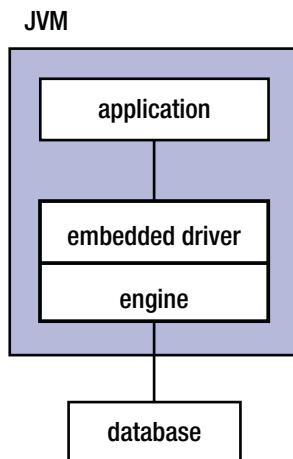


Figure 15-1. No separate processes are required to start up or shut down an embedded database engine

In a client/server environment, client applications and the database engine run in separate virtual machines. A client application accesses the network server through Derby's *client driver*. The network server, which runs in the same virtual machine as the database engine, accesses the database engine through the embedded driver. Figure 15-2 illustrates this architecture.

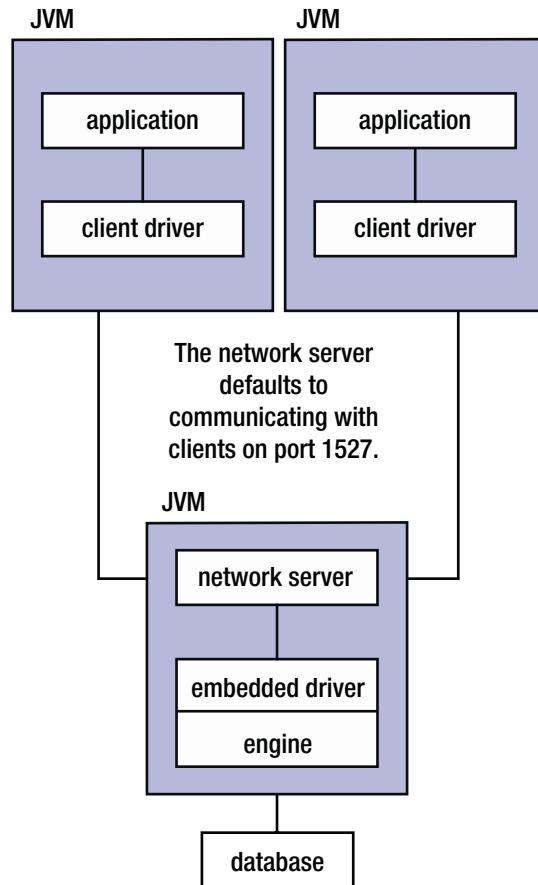


Figure 15-2. Multiple clients communicate with the same database engine through the network server

Apache Derby implements the database portion of the architectures shown in Figures 15-1 and 15-2 as a directory with the same name as the database.

Note Apache Derby doesn't provide an SQL command to *drop* (destroy) a database. Destroying a database requires that you manually delete its directory structure.

Apache Derby Installation and Configuration

Download Apache Derby from https://db.apache.org/derby/derby_downloads.html and extract it into some folder. For the rest of this chapter, we refer to this folder as DERBY_INST.

Note We focus on Apache Derby 10.14.2.0 in this chapter.

Before you can run the tools and start/stop the network server, you must set the DERBY_HOME environment variable. Set this variable for Windows via set DERBY_HOME=DERBY_INST (replace by the installation folder) and for Unix (bash shell) via export DERBY_HOME=DERBY_INST (replace by the installation folder). This setting will not persist past the current command shell session unless you make it permanent.

In order to run Apache derby in an embedded environment (e.g., from inside a Java program), you must set the CLASSPATH environment variable. Use or look at the set*CP scripts inside the bin folder for that aim.

Apache Derby Demos

The Apache Derby distribution contains demo applications inside the demo folder. There you will also find detailed descriptions about all demos as HTML files. A good starting point is the programs/simple demo.

Apache Derby Command-Line Tools

The DERBY_INST/bin directory contains, among others, sysinfo, ij, and dblook Windows and Unix/Linux script files for launching command-line tools:

- Run sysinfo to view the Java environment/Java DB configuration.
- Run ij to run scripts that execute ad hoc SQL commands and perform repetitive tasks.
- Run dblook to view all or part of a database's data definition language (DDL).

If you experience trouble with Java DB (such as not being able to connect to a database), you can run `sysinfo` to find out if the problem is configuration-related. This tool reports various settings under the Java Information, Derby Information, and Locale Information headings. The documentation tells you more about the tool.

The `ij` script is useful for creating a database and initializing a user's *schema* (a namespace that logically organizes tables and other database objects) by running a script file that specifies the appropriate DDL statements. For example, you've created an `EMPLOYEES` table with its `NAME` and `PHOTO` columns and you have created a `create_emp_schema.sql` script file in the current directory that contains the following line:

```
CREATE TABLE EMPLOYEES(NAME VARCHAR(30), PHOTO BLOB);
```

You can use any text editor to create that file (on Windows better don't use Notepad—a developer editor like Notepad++ is the better choice). The following embedded `ij` script session creates the `employees` database and `EMPLOYEES` table:

```
C:\db>ij
ij version 10.8
ij> connect 'jdbc:derby:employees;create=true';
ij> run 'create_emp_schema.sql';
ij>
ij> disconnect;
ij> exit;
C:>\db>
```

This assumes that you added `ij`'s folder to the system's PATH environment variable, or that you use that folder as the user's current directory and the SQL script lies inside the same folder. The `connect` command causes the `employees` database to be created—we'll have more to say about this command's syntax when we introduce JDBC later in this chapter. The `run` command causes `create_emp_schema.sql` to execute, and the subsequent pair of lines is generated as a result.

The `CREATE TABLE EMPLOYEES(NAME VARCHAR(30), PHOTO BLOB);` line is an SQL statement for creating a table named `EMPLOYEES` with `NAME` and `PHOTO` columns. Data items entered into the `NAME` column are of SQL type `VARCHAR` (a varying number of characters—a string) with a maximum of 30 characters, and data items entered into the `PHOTO` column are of SQL type `BLOB` (a binary large object, such as an image).

Note We specify SQL statements in uppercase, but you can also specify them in lowercase or mixed case.

After run 'create_emp_schema.sql' finishes, the specified EMPLOYEES table is added to the newly created employees database. To verify the table's existence, run dblook against the employees directory, as the following session demonstrates:

```
C:\db>dblook -d jdbc:derby:employees
-- Timestamp: 2012-11-25 16:13:42.693
-- Source database is: employees
-- Connection URL is: jdbc:derby:employees
-- appendLogs: false

-----
-- DDL Statements for tables
-----

CREATE TABLE "APP"."EMPLOYEES" ("NAME" VARCHAR(30), "PHOTO"
BLOB(2147483647));
```

C:\db>

All database objects (such as tables and indexes) are assigned to user and system schemas, which logically organize these objects in the same way that packages logically organize classes. When a user creates or accesses a database, Apache Derby uses the specified username as the namespace name for newly added database objects. In the absence of a username, Derby chooses APP, as the preceding session output shows.

Starting an Apache Derby Server

In order to start an Apache Derby database for connection in client/server mode, all you have to do is to run the script startNetworkServer inside the bin folder. Upon successful startup, the console output should be similar to

```
Mon Apr 06 11:27:25 CEST 2020 : Apache Derby Network Server - 10.15.2.0 -
(1873585) started and ready to accept connections on port 1527
```

You can now use JDBC to connect to the database. We'll talk about that in the following texts.

Embedded Apache Derby Example

Although described in the Apache Derby documentation and exemplified in the "simple" demo, we briefly describe how to use Apache Derby in an embedded mode (no DB server started), to simplify getting acquainted to using Derby.

To run Derby in embedded mode, add the following JARs to the CLASSPATH: derby.jar, derbyclient.jar, derbyshared.jar, derbytools.jar, and derbyoptionaltools.jar. You can find all of them in the lib folder of the Derby installation.

Inside your application, you then add the following code:

```
Connection conn = null;
// ... more database objects like statements, prepared statements,
// resultsets.

try {
    Properties props = new Properties(); // connection properties
    // providing a username and password is optional in the embedded mode
    props.put("user", "user1");
    props.put("password", "user1");

    String dbName = "derbyDB"; // the name of the database
    conn = DriverManager.getConnection("jdbc:derby:" + dbName +
        ";create=true", props);

    // ... from here start working with the connection

} catch (SQLException sqle) {
    // handle the exception
} finally {
    // close everything (statements, resultsets, connections)
    // ...
    try {
        if (conn != null) conn.close();
    } catch (SQLException sqle) { }
}
```

You can see that we define some database product-specific properties and then use the `DriverManager` class to open a connection.

Introducing SQLite

`SQLite` (<http://sqlite.org/>) is a very simple and popular RDBMS. Basically, it implements a self-contained, serverless, zero-configuration, transactional SQL database engine and is considered to be the most widely deployed database engine in the world. For example, SQLite is found in Mozilla Firefox, Google Chrome, and other web browsers. It's also found in Google Android, Apple iOS, and other mobile operating systems.

Note To learn what sets SQLite apart from other RDBMs, visit the “Distinctive Features of SQLite” page at <http://sqlite.org/different.html>. As well as learning about features such as the aforementioned zero-configuration, you’ll learn about features such as *manifest typing*, in which you can store any value of any data type in any column regardless of the declared type of that column.

To introduce yourself to SQLite, visit the SQLite home page at <http://sqlite.org/>. You can explore online documentation (<http://sqlite.org/docs.html>), download SQLite software (<http://sqlite.org/download.html>), and so on. Regarding downloads, you can download source code, documentation, and precompiled binaries for the Linux, Mac OS X (x86), and Windows platforms.

You can specify `sqlite3` with a database file name argument (such as `sqlite3 employees`) to create the database file (e.g., `employees`) when it doesn’t exist (you must create a table at least) or open the existing file and enter this tool’s shell from where you can execute `sqlite3`-specific, dot-prefixed commands and SQL statements. As Figure 15–3 shows, you can also specify `sqlite3` without an argument and enter the shell.

```
C:\prj\dev\lfad2\ch14\code>sqlite3
SQLite version 3.7.14.1 2012-10-04 19:37:12
Enter ".help" for instructions
Enter SQL statements terminated with a ";"

sqlite> .help
.backup ?DB? FILE      Backup DB <default "main"> to FILE
.bail ON|OFF            Stop after hitting an error. Default OFF
.databases              List names and files of attached databases
.dump ?TABLE? ...       Dump the database in an SQL text format
                        If TABLE specified, only dump tables matching
                        LIKE pattern TABLE.
.echo ON|OFF             Turn command echo on or off
.exit                   Exit this program
.explain ?ON|OFF?        Turn output mode suitable for EXPLAIN on or off.
                        With no args, it turns EXPLAIN on.
.header<s> ON|OFF       Turn display of headers on or off
.help                   Show this message
.import FILE TABLE      Import data from FILE into TABLE
.indices ?TABLE?        Show names of all indices
                        If TABLE specified, only show indices for tables
                        matching LIKE pattern TABLE.
.load FILE ?ENTRY?     Load an extension library
.log FILE{off            Turn logging on or off. FILE can be stderr/stdout
.mode MODE ?TABLE?      Set output mode where MODE is one of:
                        csv   Comma-separated values
                        column Left-aligned columns. <See .width>
                        html  HTML <table> code
                        insert SQL insert statements for TABLE
                        line  One value per line
                        list  Values delimited by .separator string
                        tabs  Tab-separated values
                        tcl   TCL list elements
```

Figure 15-3. *sqlite3* is invoked without a database file name argument

Figure 15-3 reveals the prologue that greets you after entering the *sqlite3* shell, which is indicated by the *sqlite>* prompt from where you enter commands. It also reveals part of the help text that's presented when you type the *sqlite3*-specific *.help* command.

Tip You can create a database after specifying *sqlite3* without an argument by entering the appropriate SQL statements to create and populate desired tables (and possibly create indexes) and then invoking *.backup file name* (where *file name* identifies the file that stores the database) before exiting *sqlite3*.

While discussing Apache Derby command-line tools, we presented a small employee-oriented database example consisting of an *employees* database and a *create_emp_schema.sql* script file that contains the following SQL statement for creating an *EMPLOYEES* table (consisting of names and photos):

```
CREATE TABLE EMPLOYEES(NAME VARCHAR(30), PHOTO BLOB);
```

Let's find out how to create this database and table with sqlite3.

At the command line, execute `sqlite3 employees`. At the resulting `sqlite>` command prompt, execute the aforementioned SQL statement, and then execute `.quit` to quit sqlite3. You should now observe an `employees` file in the same directory as `sqlite3`.

Reexecute `sqlite3 employees`. At the `sqlite>` command prompt, execute `.tables`. You should observe a single output line consisting of `EMPLOYEES`. Now execute `.schema employees` (case isn't significant) and you should see the aforementioned `CREATE TABLE` statement.

You can continue to play with `sqlite3` and the `employees` database/`EMPLOYEES` table. For example, you could insert a single row of data into the `EMPLOYEES` table via the following `INSERT` statement and then select/output this row via the following `SELECT` statement:

```
INSERT INTO EMPLOYEES VALUES('Duke', null);
SELECT * FROM EMPLOYEES;
```

You should observe the following line as the result—nothing appears for the photo because of its `null` value:

```
DUKE |
```

Accessing Databases via JDBC

JDBC is an API (associated with the `java.sql` and `javax.sql` packages—we mainly focus on `java.sql` in this chapter) for communicating with RDBMSs in an RDBMS-independent manner. You can use JDBC to perform various database operations, such as submitting SQL statements that tell the RDBMS to create a table and to update or query tabular data.

Data Sources, Drivers, and Connections

Although JDBC is typically used to communicate with RDBMSs, it also could be used to communicate with a flat file database. For this reason, JDBC uses the term *data source* (a data-storage facility ranging from a simple file to a complex relational database managed by an RDBMS) to abstract the source of data.

Because data sources are accessed in different ways, JDBC uses *drivers* (class file plug-ins) to abstract over their implementations. This abstraction lets you write an application that can be adapted to an arbitrary data source without having to change a single line of code (in most cases). Drivers are implementations of the `java.sql.Driver` interface.

JDBC recognizes four types of drivers.

- *Type 1 drivers* implement JDBC as a mapping to another data-access API (such as Open Database Connectivity, or ODBC—see <http://en.wikipedia.org/wiki/ODBC>). The driver converts JDBC method calls into function calls on the other library. The JDBC-ODBC Bridge driver is an example and isn't supported by Oracle. It was commonly used in the early days of JDBC when other kinds of drivers were uncommon.
- *Type 2 drivers* are written partly in Java and partly in native code. They interact with a data source-specific native client library and are not portable for this reason. Oracle's OCI (Oracle Call Interface) client-side driver is an example.
- *Type 3 drivers* don't depend on native code and communicate with a *middleware server* (a server that sits between the application client and the data source) via an RDBMS-independent protocol. The middleware server then communicates the client's requests to the data source.
- *Type 4 drivers* don't depend on native code and implement the network protocol for a specific data source. The client connects directly to the data source instead of going through a middleware server.

Before you can communicate with a data source, you need to establish a connection. JDBC provides the `java.sql.DriverManager` class and the `javax.sql.DataSource` interface for this purpose.

- `DriverManager` lets an application connect to a data source by specifying a URL. When this class first attempts to establish a connection, it automatically loads any JDBC 4.x drivers located via the classpath. (Pre-JDBC 4.x drivers must be loaded manually.)

- `DataSource` hides connection details from the application to promote data source portability and is preferred over `DriverManager` for this reason. Because a discussion of `DataSource` is somewhat involved (and is typically used in a Java EE/Jakarta EE context), we focus on `DriverManager` in this chapter.

Before letting you obtain a data source connection, early JDBC versions required you to explicitly load a suitable driver by specifying `Class.forName()` with the name of the class that implements the `Driver` interface. For example, the JDBC-ODBC Bridge driver was loaded via `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")`. Later JDBC versions relaxed this requirement by letting you specify a list of drivers to load via the `jdbc.drivers` system property. `DriverManager` would attempt to load all of these drivers during its initialization.

Under Java 7, `DriverManager` first loads all drivers identified by the `jdbc.drivers` system property. It then uses the `java.util.ServiceLoader`-based service provider mechanism to load all drivers from accessible driver JAR files so that you don't have to explicitly load drivers. This mechanism requires a driver to be packaged into a JAR file that includes `META-INF/services/java.sql.Driver`. The `java.sql.Driver` text file must contain a single line that names the driver's implementation of the `Driver` interface.

Each loaded driver instantiates and registers itself with `DriverManager` via `DriverManager`'s `void registerDriver(Driver driver)` class method. When invoked, a `getConnection()` method walks through registered drivers, returning an implementation of the `java.sql.Connection` interface from the first driver that recognizes `getConnection()`'s JDBC URL. (You might want to check out `DriverManager`'s source code to see how this is done.)

Note To maintain data source independence, much of JDBC consists of interfaces. Each driver provides implementations of the various interfaces.

To connect to a data source and obtain a `Connection` instance, call one of `DriverManager`'s `Connection getConnection(String url)`, `Connection getConnection(String url, Properties info)`, or `Connection getConnection(String url, String user, String password)` methods. With either method, the `url` argument specifies a string-based URL that starts with the `jdbc:` prefix and continues with data source-specific syntax.

Consider Apache Derby. The URL syntax varies depending on the driver. For the embedded driver (when you want to access a local database), this syntax is as follows:

```
jdbc:derby:databaseName;URLAttributes
```

For the client driver (when you want to access a remote database, although you can also access a local database with this driver), this syntax is as follows:

```
jdbc:derby://host:port/databaseName;URLAttributes
```

With either syntax, *URLAttributes* is an optional sequence of semicolon-delimited *name-value* pairs. For example, *create=true* tells Derby to create a new database in case the database doesn't exist yet.

The following example demonstrates the first syntax by telling JDBC to load the Derby embedded driver and create the database named *testdb* on the local host:

```
Connection con = DriverManager.getConnection("jdbc:derby:testdb;create=true");
```

The following example demonstrates the second syntax by telling JDBC to load the Derby client driver and create the database named *testdb* on port 8500 of the *xyz* host:

```
Connection con;
con = DriverManager.getConnection("jdbc:derby://xyz:8500/
testdb;create=true");
```

Consider SQLite. The Xerial project (www.xerial.org/trac/Xerial) provides the SQLite JDBC driver (www.xerial.org/trac/Xerial/wiki/SQLiteJDBC) for testing JDBC with SQLite. Point your browser to <https://bitbucket.org/xerial/sqlite-jdbc/downloads> and download an appropriate driver JAR file (such as *sqlite-jdbc-3.7.2.jar*).

For creating an actual file in which to store the database, the URL syntax for the Xerial SQLite driver is as follows:

```
jdbc:sqlite:databaseName
```

The following examples demonstrate this syntax for connecting to a database file (which is created when it doesn't exist) named *sample.db*:

```
Connection con1 = DriverManager.getConnection("jdbc:sqlite:sample.db");
Connection con2 = DriverManager.getConnection("jdbc:sqlite:C:/temp/
sample.db");
```

The first example obtains a connection to the current directory's `sample.db` file; the second example obtains a connection to a `sample.db` file in the `C:\temp` directory.

SQLite also supports in-memory database management, which doesn't create any database files. The following example shows you how to connect to an existing in-memory database:

```
Connection con = DriverManager.getConnection("jdbc:sqlite::memory:");
```

The following example shows you how to create and obtain a connection to an in-memory database:

```
Connection con = DriverManager.getConnection("jdbc:sqlite:");
```

Note For the most part, this chapter's applications can be used with either the Java DB embedded driver connection syntax or the non-in-memory SQLite driver connection syntax.

Statements

After obtaining a connection to a data source, an application interacts with the data source by issuing SQL statements (such as `CREATE TABLE`, `INSERT`, `SELECT`, `UPDATE`, `DELETE`, and `DROP TABLE`). JDBC supports SQL statements via the `java.sql.Statement`, `java.sql.PreparedStatement`, and `java.sql.CallableStatement` interfaces. Furthermore, `Connection` declares various `createStatement()`, `prepareStatement`, and `prepareCall()` methods that return `Statement`, `PreparedStatement`, or `CallableStatement` implementation instances, respectively.

Statement and ResultSet

`Statement` is the easiest-to-use interface, and `Connection`'s `Statement` `createStatement()` method is the easiest-to-use method for obtaining a `Statement` instance. After calling this method, you can execute various SQL statements by invoking `Statement` methods such as the following:

- `ResultSet executeQuery(String sql)` executes a `SELECT` statement and (assuming no exception is thrown) provides access to its results via a `java.sql.ResultSet` instance.

- `int executeUpdate(String sql)` executes a CREATE TABLE, INSERT, UPDATE, DELETE, or DROP TABLE statement and (assuming no exception is thrown) typically returns the number of table rows affected by this statement.

We've created a `JDBC` application that demonstrates these methods. Listing 15-1 presents its source code.

Listing 15-1. Creating, Inserting Values into, Querying, and Dropping an EMPLOYEES Table

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCdemo {
    // Running Derby in embedded mode. For client/server mode, you'd
    // use something like
    // jdbc:derby://localhost:1527/employee;create=true
    final static String URL1 = "jdbc:derby:employee;create=true";

    final static String URL2 = "jdbc:sqlite:employee";

    public static void main(String[] args) {
        String url = null;
        if (args.length != 1) {
            System.err.println("usage 1: java JDBCdemo javadb");
            System.err.println("usage 2: java JDBCdemo sqlite");
            return;
        }
        if (args[0].equals("javadb"))
            url = URL1;
        else
            if (args[0].equals("sqlite"))
                url = URL2;
```

```
else {
    System.err.println("invalid command-line argument");
    return;
}
Connection con = null;
try {
    if (args[0].equals("sqlite"))
        Class.forName("org.sqlite.JDBC");
    con = DriverManager.getConnection(url);
    Statement stmt = null;
    try {
        stmt = con.createStatement();
        String sql = "CREATE TABLE EMPLOYEES(ID INTEGER, NAME
VARCHAR(30))";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO EMPLOYEES VALUES(1, 'John Doe')";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO EMPLOYEES VALUES(2, 'Sally Smith')";
        stmt.executeUpdate(sql);
        ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEES");
        while (rs.next())
            System.out.println(rs.getInt("ID") + " " +
rs.getString("NAME"));
        stmt.executeUpdate("DROP TABLE EMPLOYEES");
    } catch (SQLException sqlex) {
        while (sqlex != null) {
            System.err.println("SQL error : " + sqlex.getMessage());
            System.err.println("SQL state : " + sqlex.getSQLState());
            System.err.println("Error code: " + sqlex.getErrorCode());
            System.err.println("Cause: " + sqlex.getCause());
            sqlex = sqlex.getNextException();
        }
    } finally {
```

```

        if (stmt != null)
            try {
                stmt.close();
            } catch (SQLException sqle) {
            }
        }

    } catch (ClassNotFoundException cnfe) {
        System.err.println("unable to load sqlite driver");
    } catch (SQLException sqlex) {
        while (sqlex != null) {
            System.err.println("SQL error : " + sqlex.getMessage());
            System.err.println("SQL state : " + sqlex.getSQLState());
            System.err.println("Error code: " + sqlex.getErrorCode());
            System.err.println("Cause: " + sqlex.getCause());
            sqlex = sqlex.getNextException();
        }
    } finally {
        if (con != null)
            try {
                con.close();
            } catch (SQLException sqle) {
            }
    }
}
}

```

After successfully establishing a connection to the employee data source, `main()` creates a statement and uses it to execute SQL statements for creating, inserting values into, querying, and dropping an `EMPLOYEES` table.

The `executeQuery()` method returns a `ResultSet` object that provides access to a query's tabular results. Each result set is associated with a *cursor* that provides access to a specific row of data. The cursor initially points before the first row; call `ResultSet`'s `boolean next()` method to advance the cursor to the next row. As long as there's a next row, this method returns true; it returns false when there are no more rows to examine.

`ResultSet` also declares various methods for returning the current row's column values based on their types. For example, `int getInt(String columnLabel)` returns the integer value corresponding to the INTEGER-based column identified by `columnLabel`. Similarly, `String getString(String columnLabel)` returns the string value corresponding to the VARCHAR-based column identified by `columnLabel`.

Tip If you don't have column names but have one-based column indexes, call `ResultSet` methods such as `int getInt(int columnIndex)` and `String getString(int columnIndex)`. However, the best practice is to call `int getInt(String columnLabel)`.

Compile Listing 15-1 and run this application. You should observe the following output:

```
1 John Doe
2 Sally Smith
```

SQL's INTEGER and VARCHAR types map to Java's `int` and `java.lang.String` types. Table 15-1 presents a more complete list of type mappings.

Table 15-1. SQL Type/Java Type Mappings

SQL Type	Java Type
ARRAY	<code>java.sql.Array</code>
BIGINT	<code>Long</code>
BINARY	<code>byte[]</code>
BIT	<code>Boolean</code>
BLOB	<code>java.sql.Blob</code>
BOOLEAN	<code>Boolean</code>
CHAR	<code>java.lang.String</code>
CLOB	<code>java.sql.Clob</code>
DATE	<code>java.sql.Date</code>
DECIMAL	<code>java.math.BigDecimal</code>

(continued)

Table 15-1. (continued)

SQL Type	Java Type
DOUBLE	Double
FLOAT	Double
INTEGER	Int
LONGVARBINARY	byte[]
LONGVARCHAR	java.lang.String
NUMERIC	java.math.BigDecimal
REAL	Float
REF	java.sql.Ref
SMALLINT	Short
STRUCT	java.sql.Struct
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
TINYINT	Byte
VARBINARY	byte[]
VARCHAR	java.lang.String

Check out <http://docs.oracle.com/javase/1.5.0/docs/guide/jdbc/getstart/mapping.html> for more information on type mappings.

PreparedStatement

`PreparedStatement` is the next easiest-to-use interface, and `Connection`'s `PreparedStatement prepareStatement()` method is the easiest-to-use method for obtaining a `PreparedStatement` instance—`PreparedStatement` is a subinterface of `Statement`.

Unlike a regular statement, a *prepared statement* represents a precompiled SQL statement. The SQL statement is compiled to improve performance and prevent *SQL injection* (see http://en.wikipedia.org/wiki/SQL_injection), and the compiled result is stored in a `PreparedStatement` implementation instance.

You typically obtain this instance when you want to execute the same prepared statement multiple times. For example, you want to execute an SQL INSERT statement multiple times to populate a database table. Consider Listing 15-2.

Listing 15-2. Creating, Inserting Values via a Prepared Statement into, Querying, and Dropping an EMPLOYEES Table

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCDemo {
    final static String URL1 = "jdbc:derby:employee;create=true";
    final static String URL2 = "jdbc:sqlite:employee";

    public static void main(String[] args) {
        String url = null;
        if (args.length != 1) {
            System.err.println("usage 1: java JDBCDemo derby");
            System.err.println("usage 2: java JDBCDemo sqlite");
            return;
        }
        if (args[0].equals("derby"))
            url = URL1;
        else if (args[0].equals("sqlite"))
            url = URL2;
        else {
            System.err.println("invalid command-line argument");
            return;
        }
        Connection con = null;
        try {
            if (args[0].equals("sqlite"))
                Class.forName("org.sqlite.JDBC");
            con = DriverManager.getConnection(url);
```

```
Statement stmt = null;
try {
    stmt = con.createStatement();
    String sql = "CREATE TABLE EMPLOYEES(ID INTEGER, NAME
VARCHAR(30))";
    stmt.executeUpdate(sql);
    PreparedStatement pstmt = null;
    try {
        pstmt = con.prepareStatement("INSERT INTO EMPLOYEES
VALUES(?, ?)");
        String[] empNames = { "John Doe", "Sally Smith" };
        for (int i = 0; i < empNames.length; i++) {
            pstmt.setInt(1, i+1);
            pstmt.setString(2, empNames[i]);
            pstmt.executeUpdate();
        }
        ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEES");
        while (rs.next())
            System.out.println(rs.getInt("ID") + " " +
rs.getString("NAME"));
        stmt.executeUpdate("DROP TABLE EMPLOYEES");
    } catch (SQLException sqlex) {
        while (sqlex != null) {
            System.err.println("SQL error : " + sqlex.getMessage());
            System.err.println("SQL state : " + sqlex.getSQLState());
            System.err.println("Error code: " + sqlex.
getErrorCode());
            System.err.println("Cause: " + sqlex.getCause());
            sqlex = sqlex.getNextException();
        }
    } finally {
        if (pstmt != null)
            try {
                pstmt.close();
            } catch (SQLException sqle) {
                sqle.printStackTrace();
            }
    }
}
```

```
        }
    }
} catch (SQLException sqlex) {
    while (sqlex != null) {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
} finally {
    if (stmt != null)
        try {
            stmt.close();
        } catch (SQLException sqle) {
        }
}
} catch (ClassNotFoundException cnfe) {
    System.err.println("unable to load sqlite driver");
} catch (SQLException sqlex) {
    while (sqlex != null) {
        System.err.println("SQL error : " + sqlex.getMessage());
        System.err.println("SQL state : " + sqlex.getSQLState());
        System.err.println("Error code: " + sqlex.getErrorCode());
        System.err.println("Cause: " + sqlex.getCause());
        sqlex = sqlex.getNextException();
    }
} finally {
    if (con != null)
        try {
            con.close();
        } catch (SQLException sqle) {
        }
}
}
```

Listing 15-2 creates a `String` object that specifies an SQL `INSERT` statement. Each “?” character serves as a placeholder for a value that’s specified before the statement is executed.

After the `PreparedStatement` implementation instance has been obtained, this interface’s `void setInt(int parameterIndex, int x)` and `void setString(int parameterIndex, String x)` methods are called on this instance to provide these values (the first argument passed to each method is a 1-based integer column index into the table associated with the statement—1 corresponds to the leftmost column), and then `PreparedStatement`’s `int executeUpdate()` method is called to execute this SQL statement. The end result is that a pair of rows containing John Doe, Sally Smith, and their respective identifiers is added to the `EMPLOYEES` table.

CallableStatement

`CallableStatement` is the most specialized of the statement interfaces; it extends `PreparedStatement`. You use this interface to execute SQL stored procedures in which a *stored procedure* is a list of SQL statements that perform a specific task (such as fire an employee). Java DB differs from other RDBMSs in that a stored procedure’s body is implemented as a `public static` Java method. Furthermore, the class in which this method is declared must be `public`.

Note SQLite doesn’t support stored procedures.

You create a stored procedure by executing an SQL statement that typically begins with `CREATE PROCEDURE` and then continues with RDBMS-specific syntax. For example, the Apache Derby syntax for creating a stored procedure, as specified on the web page at <http://db.apache.org/derby/docs/10.8/ref/rrefcreateprocedurestatement.html>, is as follows:

```
CREATE PROCEDURE procedure-name ([ procedure-parameter [, procedure-parameter] ]*)  
[ procedure-element ]*
```

procedure-name is expressed as

```
[ schemaName .] SQL92Identifier
```

procedure-parameter is expressed as

```
[{ IN | OUT | INOUT }] [ parameter-Name ] DataType
```

procedure-element is expressed as

```
{
| [ DYNAMIC ] RESULT SETS INTEGER
| LANGUAGE { JAVA }
| DeterministicCharacteristic
| EXTERNAL NAME string
| PARAMETER STYLE JAVA
| EXTERNAL SECURITY { DEFINER | INVOKER }
| { NO SQL | MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA }
}
```

Anything between [] is optional, the * to the right of [] indicates that anything between these metacharacters can appear zero or more times, the {} metacharacters surround a list of items, and | separates possible items—only one of these items can be specified.

For example, CREATE PROCEDURE FIRE(IN ID INTEGER) PARAMETER STYLE JAVA LANGUAGE JAVA DYNAMIC RESULT SETS 0 EXTERNAL NAME 'JDBCDemo.fire' creates a stored procedure named FIRE. This procedure specifies an input parameter named ID and is associated with a public static method named fire in a public class named JDBCDemo.

After creating the stored procedure, you need to obtain a CallableStatement implementation instance in order to call that procedure, and you do so by invoking one of Connection's prepareCall() methods, for example, CallableStatement prepareCall(String sql).

The string passed to prepareCall() is an *escape clause* (RDBMS-independent syntax) consisting of an open {, followed by the word call, followed by a space, followed by the name of the stored procedure, followed by a parameter list with "?" placeholder characters for the arguments that will be passed, followed by a closing }.

Note Escape clauses are JDBC’s way of smoothing out some of the differences in how different RDBMS vendors implement SQL. When a JDBC driver detects escape syntax, it converts it into the code that the particular RDBMS understands. This makes escape syntax RDBMS independent.

Once you have a `CallableStatement` reference, you pass arguments to these parameters in the same way as with `PreparedStatement`. The following example demonstrates:

```
CallableStatement cstmt = null;
try {
    cstmt = con.prepareCall("{ call FIRE(?) }")
    cstmt.setInt(1, 2);
    cstmt.execute();
} catch (SQLException sqle) {
    // handle the exception
} finally {
    // close the callable statement
}
```

The `cstmt.setInt(1, 2)` method call assigns 2 to the leftmost stored procedure parameter—parameter index 1 corresponds to the leftmost parameter (or to a single parameter when there’s only one). The `cstmt.execute()` method call executes the stored procedure, which results in a callback to the application’s `public static void fire(int id)` method.

Metadata

A data source is typically associated with *metadata* (data about data) that describes the data source. When the data source is an RDBMS, this data is typically stored in a collection of tables.

Metadata includes a list of *catalogs* (RDBMS databases whose tables describe RDBMS objects such as *base tables* [tables that physically exist], *views* [virtual tables], and *indexes* [files that improve the speed of data retrieval operations]), *schemas*

(namespaces that partition database objects), and additional information (such as version numbers, identification strings, and limits).

To access a data source's metadata, invoke `Connection`'s `DatabaseMetaData getMetaData()` method. This method returns an implementation instance of the `java.sql.DatabaseMetaData` interface.

We've created another `JDBC Demo` application that demonstrates `getMetaData()` and various `DatabaseMetaData` methods in the context of Java DB. Listing 15-3 presents `MetaData`'s source code.

Listing 15-3. Obtaining Metadata from an employee Data Source

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBC Demo {
    public static void main(String[] args) {
        String url = "jdbc:derby:employee;create=true";
        Connection con = null;
        try {
            con = DriverManager.getConnection(url);
            dump(con.getMetaData());
        } catch (SQLException sqlex) {
            while (sqlex != null) {
                System.err.println("SQL error : " + sqlex.getMessage());
                System.err.println("SQL state : " + sqlex.getSQLState());
                System.err.println("Error code: " + sqlex.getErrorCode());
                System.err.println("Cause: " + sqlex.getCause());
                sqlex = sqlex.getNextException();
            }
        } finally {
```

```
if (con != null)
    try {
        con.close();
    } catch (SQLException sqle) {
    }
}

private static void dump(DatabaseMetaData dbmd) throws SQLException {
    System.out.println("DB Major Version = " + dbmd.
        getDatabaseMajorVersion());
    System.out.println("DB Minor Version = " + dbmd.
        getDatabaseMinorVersion());
    System.out.println("DB Product = " + dbmd.getDatabaseProductName());
    System.out.println("Driver Name = " + dbmd.getDriverName());
    System.out.println("Numeric function names for escape clause = " +
        dbmd.getNumericFunctions());
    System.out.println("String function names for escape clause = " +
        dbmd.getStringFunctions());
    System.out.println("System function names for escape clause = " +
        dbmd.getSystemFunctions());
    System.out.println("Time/date function names for escape clause = " +
        dbmd.getTimeDateFunctions());
    System.out.println("Catalog term: " + dbmd.getCatalogTerm());
    System.out.println("Schema term: " + dbmd.getSchemaTerm());
    System.out.println();
    System.out.println("Catalogs");
    System.out.println("-----");
    ResultSet rsCat = dbmd.getCatalogs();
    while (rsCat.next())
        System.out.println(rsCat.getString("TABLE_CAT"));
    System.out.println();
    System.out.println("Schemas");
    System.out.println("-----");
    ResultSet rsSchem = dbmd.getSchemas();
    while (rsSchem.next())
```

```

        System.out.println(rsSchema.getString("TABLE_SCHEM"));
        System.out.println();
        System.out.println("Schema/Table");
        System.out.println("-----");
        rsSchema = dbmd.getschemas();
        while (rsSchema.next()) {
            String schema = rsSchema.getString("TABLE_SCHEM");
            ResultSet rsTab = dbmd.getTables(null, schema, "%", null);
            while (rsTab.next())
                System.out.println(schema + " " + rsTab.getString("TABLE_NAME"));
        }
    }
}

```

Listing 15-3's `dump()` method invokes various methods on its `dbmd` argument to output assorted metadata.

The `int getDatabaseMajorVersion()` and `int getDatabaseMinorVersion()` methods return the major (such as 10) and minor (such as 8) parts of Derby's version number. Similarly, `String getDatabaseProductName()` returns the name of this product (such as Apache Derby), and `String getDriverName()` returns the name of the driver (such as Apache Derby Embedded JDBC Driver).

SQL defines various functions that can be invoked as part of SELECT and other statements. For example, you can specify `SELECT COUNT(*) AS TOTAL FROM EMPLOYEES` to return a one-row-by-one-column result set with the column named `TOTAL` and the row value containing the number of rows in the `EMPLOYEES` table.

Because not all RDBMSs adopt the same syntax for specifying function calls, JDBC uses a *function escape clause*, consisting of `{ fn functionname(arguments) }`, to abstract over differences. For example, `SELECT {fn UCASE(NAME)} FROM EMPLOYEES` selects all `NAME` column values from `EMPLOYEES` and uppercases their values in the result set.

The `String getNumericFunctions()`, `String getStringFunctions()`, `String getSystemFunctions()`, and `String getTimeDateFunctions()` methods return lists of function names that can appear in function escape clauses. For example, `getNumericFunctions()` returns ABS, ACOS, ASIN, ATAN, ATAN2, CEILING, COS, COT, DEGREE S, EXP, FLOOR, LOG, LOG10, MOD, PI, RADIANS, RAND, SIGN, SIN, SQRT, TAN for Java DB 10.8.

Not all vendors use the same terminology for catalog and schema. For this reason, the String `getCatalogTerm()` and String `getSchemaTerm()` methods are present to return the vendor-specific terms, which happen to be CATALOG and SCHEMA for Derby.

The `ResultSet getCatalogs()` method returns a result set of catalog names, which are accessible via the result set's TABLE_CAT column. This result set is empty for Derby, which divides a single default catalog into various schemas.

The `ResultSet getSchemas()` method returns a result set of schema names, which are accessible via the result set's TABLE_SCHEM column. This column contains APP, NULLID, SQLJ, SYS, SYSCAT, SYSCS_DIAG, SYSCS_UTIL, SYSFUN, SYSIBM, SYSPROC, and SYSSTAT values for Derby. APP is the default schema in which a user's database objects are stored.

The `ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)` method returns a result set containing table names (in the TABLE_NAME column) and other table-oriented metadata that match the specified catalog, schemaPattern, tableNamePattern, and types. To obtain a result set of all tables for a specific schema, pass null to catalog and types, the schema name to schemaPattern, and the % wildcard to tableNamePattern.

For example, the SYS schema stores SYSALIASES, SYSCHECKS, SYSCOLPERMS, SYSCOLUMNS, SYSCONGLOMERATES, SYSCONSTRAINTS, SYSDEPENDS, SYSFILES, SYSFOREIGNKEYS, SYSKEYS, SYSPERMS, SYSROLES, SYSROUTINEPERMS, SYSSCHEMAS, SYSSEQUENCES, SYSSTATEMENTS, SYSSTATISTICS, SYSTABLEPERMS, SYSTABLES, SYSTRIGGERS, and SYSVIEWS tables.

EXERCISES

The following exercises are designed to test your understanding of Chapter 15's content:

1. Define database.
2. What is a relational database?
3. Define database management system.
4. True or false: Apache Derby's client driver causes the database engine to run in the same virtual machine as the application.
5. What is JDBC?

6. A JDBC driver implements what interface?
 7. True or false: There are three kinds of JDBC drivers.
 8. Describe a Type 3 JDBC driver.
 9. How do you obtain a connection to an Apache Derby DB data source via the embedded driver?
 10. What is a SQL state error code?
 11. What is the difference between `SQLNonTransientException` and `SQLTransientException`?
 12. Identify JDBC's three statement types.
 13. Which Statement method do you call to execute an SQL SELECT statement?
 14. What does a result set's cursor accomplish?
 15. To which Java type does the SQL FLOAT type map?
 16. What does a prepared statement represent?
 17. True or false: CallableStatement extends PreparedStatement.
 18. Define stored procedure.
 19. How do you call a stored procedure?
 20. Define metadata.
 21. What does metadata include?
-

Summary

A database is an organized collection of data. Although there are many kinds of databases (such as hierarchical, object-oriented, and relational), relational databases, which organize data into tables that can be related to each other, are common.

Except for the most trivial of databases (such as a flat file database based on a single data file), databases are created and managed through a database management system. Relational DBMSs support SQL for working with tables and more.

First introduced by Sun Microsystems as part of JDK 6 (and not included in the JRE) to give developers an RDBMS to test their JDBC code, Java DB is a distribution of Apache's open source Derby product, which is based on IBM's Cloudscape RDBMS code base.

Apache Derby is capable of running in an embedded environment or in a client/server environment. In an embedded environment, where an application accesses the database engine via Derby's embedded driver, the database engine runs in the same virtual machine as the application.

In a client/server environment, client applications and the database engine run in separate virtual machines. A client application accesses the network server through Apache Derby's client driver. The network server, which runs in the same virtual machine as the database engine, accesses the database engine through the embedded driver.

SQLite is a self-contained, serverless, zero-configuration, transactional SQL database engine; it is considered to be the most widely deployed database engine in the world. For example, SQLite is found in Mozilla Firefox, Google Chrome, and other web browsers. It's also found in Google Android, Apple iOS, and other mobile operating systems.

The `sqlite3` executable offers a command-line shell for accessing and modifying SQLite databases. You can specify `sqlite3` with a database file name argument to create the database file when it doesn't exist (you must create a table at least) or open the existing file, and enter this tool's shell from where you can execute `sqlite3`-specific, dot-prefixed commands and SQL statements.

JDBC is an API for performing various database operations, such as submitting SQL statements that tell the RDBMS to create a table and to update or query tabular data. Although JDBC is typically used to communicate with RDBMSs, it also could be used to communicate with a flat file database. For this reason, JDBC uses the term data source to abstract the source of data.

Because data sources are accessed in different ways, JDBC uses drivers to abstract over their implementations. This abstraction lets you write an application that can be adapted to an arbitrary data source without having to change a single line of code (in most cases). Drivers are implementations of the `java.sql.Driver` interface. JDBC recognizes four types of drivers.

To connect to a data source and obtain a `Connection` instance, call one of `DriverManager`'s `getConnection()` methods. With either method, the `url` argument

specifies a string-based URL that starts with the `jdbc:` prefix and continues with data source-specific syntax.

`DriverManager`'s `getConnection()` methods (and other JDBC methods in the various JDBC interfaces) throw `SQLException` or one of its subclasses when something goes wrong. Instances of this class provide vendor codes, SQL state strings, and other kinds of information.

After obtaining a connection to a data source, an application interacts with the data source by issuing SQL statements. JDBC supports SQL statements via the `Statement`, `PreparedStatement`, and `CallableStatement` interfaces.

The `executeQuery()` methods return a `ResultSet` object that provides access to a query's tabular results. Each result set is associated with a cursor that provides access to a specific row of data. The cursor initially points before the first row.

`ResultSet` also declares various methods for returning the current row's column values based on their types. For example, `int getInt(String columnLabel)` returns the integer value corresponding to the `INTEGER`-based column identified by `columnLabel`.

A prepared statement represents a precompiled SQL statement. The SQL statement is compiled to improve performance and prevent SQL injection, and the compiled result is stored in a `PreparedStatement` implementation instance.

A callable statement is a special kind of prepared statement for executing SQL stored procedures in which a stored procedure is a list of SQL statements that perform a specific task. The argument passed to a callable statement's `prepareCall()` method is specified using escape syntax.

A data source is typically associated with metadata that describes the data source. When the data source is an RDBMS, this data is typically stored in a collection of tables. Metadata includes a list of catalogs, base tables, views, indexes, schemas, and additional information.

Databases can store XML documents, which are a convenient way to exchange data. Chapter 16 introduces you to XML and SON and shows you how to parse, create, and transform XML and JSON documents.

CHAPTER 16

Working with XML and JSON Documents

Applications commonly use XML and JSON documents to store and exchange data. In Chapter 16, we introduce XML and JSON for the benefit of those who are unfamiliar with these technologies.

Java supports XML via the SAX, DOM, StAX, XPath, and XSLT APIs. After introducing XML, we also introduce these APIs, except for StAX, which Android doesn't support.

Java JSON processing gets handled by JSR 374 (or the older JSR 353). It is not part of the JSE though, but you find it in Java EE and Jakarta EE, and you can add an implementation in the form of a library so it can be used for JSE as well. Although Android provides its own JSON handling library, we will briefly introduce the Java standard API, so you can get familiar with the basics of JSON processing.

What Is XML?

XML (Extensible Markup Language) is a *metalinguage* (a language used to describe other languages) for defining *vocabularies* (custom markup languages), which is key to XML's importance and popularity. XML-based vocabularies (such as XHTML) let you describe documents in a meaningful way.

XML vocabulary documents are like HTML (see <http://en.wikipedia.org/wiki/HTML>) documents in that they are text-based and consist of *markup* (encoded descriptions of a document's logical structure) and *content* (document text not interpreted as markup). Markup is evidenced via *tags* (angle bracket-delimited syntactic constructs) and each tag has a name. Furthermore, some tags have *attributes* (name-value pairs).

If you haven't previously encountered XML, you might be surprised by its simplicity and how closely its vocabularies resemble HTML. You don't need to be a rocket scientist to learn how to create an XML document. To prove this to yourself, check out Listing 16-1.

Listing 16-1. XML-Based Recipe for a Grilled Cheese Sandwich

```
<recipe>
  <title>
    Grilled Cheese Sandwich
  </title>
  <ingredients>
    <ingredient qty="2">
      bread slice
    </ingredient>
    <ingredient>
      cheese slice
    </ingredient>
    <ingredient qty="2">
      margarine pat
    </ingredient>
  </ingredients>
  <instructions>
    Place frying pan on element and select medium heat. For each bread
    slice, smear one pat of margarine on one side of bread slice.
    Place cheese slice between bread slices with margarine-smeared sides
    away from the cheese. Place sandwich in frying pan with one
    margarine-smeared side in contact with pan. Fry for a couple of
    minutes and flip. Fry other side for a minute and serve.
  </instructions>
</recipe>
```

Listing 16-1 presents an XML document that describes a recipe for making a grilled cheese sandwich. This document is reminiscent of an HTML document in that it consists of tags, attributes, and content. However, that's where the similarity ends. Instead of presenting HTML tags such as `<html>`, `<head>`, ``, and `<p>`, this informal recipe language presents its own `<recipe>`, `<ingredients>`, and other tags.

Note Although Listing 16-1's <title> and </title> tags are also found in HTML, they differ from their HTML counterparts. Web browsers typically display the content between these tags in their title bars. In contrast, the content between Listing 16-1's <title> and </title> tags might be displayed as a header, spoken aloud, or presented in some other way, depending on the application that parses this document.

XML documents are based on the XML declaration, elements and attributes, character references and CDATA sections, namespaces, and comments and processing instructions. After learning about these fundamentals, you'll learn what it means for an XML document to be well formed. You will also learn what it means for an XML document to be valid.

XML Declaration

An XML document will typically begin with the *XML declaration*, a special markup that informs an XML parser that the document is XML. The absence of the XML declaration in Listing 16-1 reveals that this special markup isn't mandatory. When the XML declaration is present, nothing can appear before it.

The XML declaration minimally looks like <?xml version="1.0"?> in which the nonoptional *version* attribute identifies the version of the XML specification to which the document conforms. The initial version of this specification (1.0) was introduced in 1998 and is widely implemented.

Note The World Wide Web Consortium (W3C), which maintains XML, released version 1.1 in 2004. This version mainly supports the use of line-ending characters used on EBCDIC platforms (see <http://en.wikipedia.org/wiki/EBCDIC>) and the use of scripts and characters that are absent from Unicode 3.2 (see <http://en.wikipedia.org/wiki/Unicode>). Unlike XML 1.0, XML 1.1 isn't widely implemented and should be used only by those needing its unique features.

XML supports Unicode, which means that XML documents consist entirely of characters taken from the Unicode character set. The document's characters are encoded into bytes for storage or transmission, and the encoding is specified via the XML declaration's optional encoding attribute. One common encoding is *UTF-8* (see <http://en.wikipedia.org/wiki/UTF-8>), which is a variable-length encoding of the Unicode character set. UTF-8 is a strict superset of ASCII (see <http://en.wikipedia.org/wiki/Ascii>), which means that pure ASCII text files are also UTF-8 documents.

Note In the absence of the XML declaration or when the XML declaration's encoding attribute isn't present, an XML parser typically looks for a special character sequence at the start of a document to determine the document's encoding. This character sequence is known as the *byte-order mark (BOM)* and is created by an editor program (such as Microsoft Windows Notepad) when it saves the document according to UTF-8 or some other encoding. For example, the hexadecimal sequence EF BB BF signifies UTF-8 as the encoding. Similarly, FE FF signifies UTF-16 big endian (see <http://en.wikipedia.org/wiki/UTF-16/UCS-2>), FF FE signifies UTF-16 little endian, 00 00 FE FF signifies UTF-32 big endian (see <http://en.wikipedia.org/wiki/UTF-32/UCS-4>), and FF FE 00 00 signifies UTF-32 little endian. UTF-8 is assumed when no BOM is present.

If you'll never use characters apart from the ASCII character set, you can probably forget about the encoding attribute. However, when your native language isn't English or when you're called to create XML documents that include non-ASCII characters, you need to specify encoding properly. For example, when your document contains ASCII plus characters from a non-English Western European language (such as ç, the cedilla used in French, Portuguese, and other languages), you might want to choose ISO-8859-1 as the encoding attribute's value; the document will probably have a smaller size when encoded in this manner than when encoded with UTF-8. Listing 16-2 shows you the resulting XML declaration.

Listing 16-2. An Encoded Document Containing Non-ASCII Characters

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<movie>
    <name>Le Fabuleux Destin d'Amélie Poulain</name>
    <language>français</language>
</movie>
```

The final attribute that can appear in the XML declaration is `standalone`. This optional attribute determines whether the XML document relies on an external DTD (discussed later in this chapter) or not: its value is `no` when relying on an external DTD, or `yes` when not relying on an external DTD. The value defaults to `no`, implying that there is an external DTD. However, because there's no guarantee of a DTD, `standalone` is rarely used and won't be discussed further.

Elements and Attributes

Following the XML declaration is a *hierarchical* (tree) structure of elements, where an *element* is a portion of the document delimited by a *start tag* (such as `<name>`) and an *end tag* (such as `</name>`), or is an *empty-element tag* (a stand-alone tag whose name ends with a forward slash [/], such as `<break/>`). Start tags and end tags surround content and possibly other markup, whereas empty-element tags don't surround anything. Figure 16-1 reveals Listing 16-1's XML document tree structure.

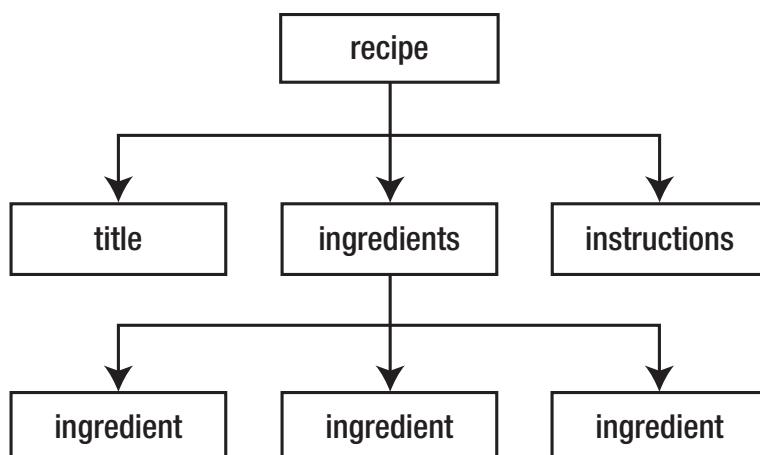


Figure 16-1. Listing 16-1's tree structure is rooted in the `recipe` element

As with HTML document structure, the structure of an XML document is anchored in a *root element* (the topmost element). In HTML, the root element is `html` (the `<html>` and `</html>` tag pair). Unlike in HTML, you can choose the root element for your XML documents. Figure 16-1 shows the root element to be `recipe`.

Unlike the other elements that have parent elements, `recipe` has no parent. Also, `recipe` and `ingredients` have child elements: `recipe`'s children are `title`, `ingredients`, and `instructions`; and `ingredients`' children are three instances of `ingredient`. The `title`, `instructions`, and `ingredient` elements don't have child elements.

Elements can contain child elements, content, or *mixed content* (a combination of child elements and content). Listing 16-2 reveals that the `movie` element contains `name` and `language` child elements, and it also reveals that each of these child elements contains content (e.g., `language` contains `français`). Listing 16-3 presents another example that demonstrates mixed content along with child elements and content.

Listing 16-3. An abstract Element Containing Mixed Content

```
<?xml version="1.0"?>
<article title="The Rebirth of JavaFX" lang="en">
    <abstract>
        JavaFX 2.0 marks a significant milestone in the history of
        JavaFX. Now that Sun Microsystems has passed the torch to Oracle,
        we have seen the demise of JavaFX Script and the emerge of Java APIs
        (such as <code-inline>javafx.application.Application</code-inline>)
        for interacting with this technology. This article introduces you
        to this new flavor of JavaFX, where you learn about JavaFX 2.0
        architecture and key APIs.
    </abstract>
    <body>
        </body>
</article>
```

This document's root element is `article`, which contains `abstract` and `body` child elements. The `abstract` element mixes content with a `code-in-line` element, which contains content. In contrast, the `body` element is empty.

Note As with Listings 16-1 and 16-2, Listing 16-3 also contains *whitespace* (invisible characters such as spaces, tabs, carriage returns, and line feeds). The XML specification permits whitespace to be added to a document. Whitespace appearing within content (such as spaces between words) is considered part of the content. In contrast, the parser typically ignores whitespace appearing between an end tag and the next start tag. Such whitespace isn't considered part of the content.

An XML element's start tag can contain one or more attributes. For example, Listing 16-1's `<ingredient>` tag has a `qty` (quantity) attribute and Listing 16-3's `<article>` tag has `title` and `lang` attributes. Attributes provide additional information about elements. For example, `qty` identifies the amount of an ingredient that can be added, `title` identifies an article's title, and `lang` identifies the language in which the article is written (`en` for English). Attributes can be optional. For example, when `qty` isn't specified, a default value of 1 is assumed.

Note Element and attribute names may contain any alphanumeric character from English or another language, and they may also include the underscore (`_`), hyphen (`-`), period (`.`), and colon (`:`) punctuation characters. The colon should only be used with namespaces (discussed later in this chapter), and names cannot contain whitespace.

Character References and CDATA Sections

Certain characters cannot appear literally in the content that appears between a start tag and an end tag, or within an attribute value. For example, you cannot place a literal `<` character between a start tag and an end tag because doing so would confuse an XML parser into thinking that it had encountered another tag.

One solution to this problem is to replace the literal character with a *character reference*, which is a code that represents the character. Character references are classified as numeric character references or character entity references.

- A *numeric character reference* refers to a character via its Unicode code point and adheres to the format `&#nnnn;` (not restricted to four positions) or `&#xhhhh;` (not restricted to four positions), where *nnnn* provides a decimal representation of the code point and *hhhh* provides a hexadecimal representation. For example, `Σ` and `Σ` represent the Greek capital letter sigma. Although XML mandates that the `x` in `&#xhhhh;` be lowercase, it's flexible in that the leading zero is optional in either format and in allowing you to specify an uppercase or lowercase letter for each *h*. As a result, `Σ`, `Σ`, and `Σ` are also valid representations of the Greek capital letter sigma.
- A *character entity reference* refers to a character via the name of an *entity* (aliased data) that specifies the desired character as its replacement text. Character entity references are predefined by XML and have the format `&name;`, in which *name* is the entity's name. XML predefines five-character entity references: `<` (`<`), `>` (`>`), `&` (`&`), `'` (`'`), and `"` (`"`).

Consider `<expression>6 < 4</expression>`. You could replace the `<` with numeric reference `<`, yielding `<expression>6 < 4</expression>`, or better yet with `<`, yielding `<expression>6 < 4</expression>`. The second choice is clearer and easier to remember.

Suppose you want to embed an HTML or XML document within an element. To make the embedded document acceptable to an XML parser, you would need to replace each literal `<` (start of tag) and `&` (start of entity) character with its `<` and `&` predefined character entity reference, a tedious and possibly error-prone undertaking since you might forget to replace one of these characters. To save you from tedium and potential errors, XML provides an alternative in the form of a CDATA (character data) section.

A *CDATA section* is a section of literal HTML or XML markup and content surrounded by the `<! [CDATA[prefix and the]]>` suffix. You don't need to specify predefined character entity references within a CDATA section, as demonstrated in Listing 16-4.

Listing 16-4. Embedding an XML Document in Another Document's CDATA Section

```
<?xml version="1.0"?>
<svg-examples>
    <example>
        The following Scalable Vector Graphics document describes a blue-
        filled and
        black-stroked rectangle.
        <![CDATA[<svg width="100%" height="100%" version="1.1"
            xmlns="http://www.w3.org/2000/svg">
            <rect width="300" height="100"
                style="fill:rgb(0,0,255);stroke-width:1; stroke:rgb(0,0,0)"/>
        </svg>]]>
    </example>
</svg-examples>
```

Listing 16-4 embeds a Scalable Vector Graphics (SVG) (see <http://en.wikipedia.org/wiki/Svg>) XML document within the example element of an SVG examples document. The SVG document is placed in a CDATA section, obviating the need to replace all < characters with < predefined character entity references.

Namespaces

It's common to create XML documents that combine features from different XML languages. Namespaces are used to prevent name conflicts when elements and other XML language features appear. Without namespaces, an XML parser couldn't distinguish between same-named elements or other language features that mean different things, such as two same-named title elements from two different languages.

Note Namespaces aren't part of XML 1.0. They arrived about a year after this specification was released. To ensure backward compatibility with XML 1.0, namespaces take advantage of colon characters, which are legal characters in XML names. Parsers that don't recognize namespaces return names that include colons.

A *namespace* is a Uniform Resource Identifier (URI) -based container that helps differentiate XML vocabularies by providing a unique context for its contained identifiers. The namespace URI is associated with a *namespace prefix* (an alias for the URI) by specifying, typically on an XML document's root element, either the `xmlns` attribute by itself (which signifies the default namespace) or the `xmlns:prefix` attribute (which signifies the namespace identified as *prefix*), and assigning the URI to this attribute.

Note A namespace's scope starts at the element where it's declared and is applied to all of the element's content unless overridden by another namespace declaration with the same prefix name.

When *prefix* is specified, it and a colon character are prepended to the name of each element tag that belongs to that namespace (see Listing 16-5).

Listing 16-5. Introducing a Pair of Namespaces

```
<?xml version="1.0"?>
<h:html xmlns:h="http://www.w3.org/1999/xhtml"
         xmlns:r="http://www.tutortutor.ca/">
    <h:head>
        <h:title>
            Recipe
        </h:title>
    </h:head>
    <h:body>
        <r:recipe>
            <r:title>
                Grilled Cheese Sandwich
            </r:title>
            <r:ingredients>
                <h:ul>
                    <h:li>
                        <r:ingredient qty="2">
                            bread slice
                        </r:ingredient>
                    </h:li>
                </h:ul>
            </r:ingredients>
        </r:recipe>
    </h:body>
</h:html>
```

```
</r:ingredient>
</h:li>
<h:li>
<r:ingredient>
    cheese slice
</r:ingredient>
</h:li>
<h:li>
<r:ingredient qty="2">
    margarine pat
</r:ingredient>
</h:li>
</h:ul>
</r:ingredients>
<h:p>
<r:instructions>
    Place frying pan on element and select medium heat. For each bread
    slice, smear one pat of margarine on one side of bread slice.
    Place cheese slice between bread slices with margarine-smeared
    sides away from the cheese. Place sandwich in frying pan with one
    margarine-smeared side in contact with pan. Fry for a couple of
    minutes and flip. Fry other side for a minute and serve.
</r:instructions>
</h:p>
</r:recipe>
</h:body>
</h:html>
```

Listing 16-5 describes a document that combines elements from the XHTML language (see <http://en.wikipedia.org/wiki/XHTML>) with elements from the recipe language. All element tags that associate with XHTML are prefixed with h:, and all element tags that associate with the recipe language are prefixed with r:.

The `h:` prefix associates with the www.w3.org/1999/xhtml URI, and the `r:` prefix associates with the www.tutortutor.ca URI. XML doesn't mandate that URIs point to document files. It only requires that they be unique to guarantee unique namespaces.

This document's separation of the recipe data from the XHTML elements makes it possible to preserve this data's structure while also allowing an XHTML-compliant web browser (such as Google Chrome) to present the recipe via a web page (see Figure 16-2).

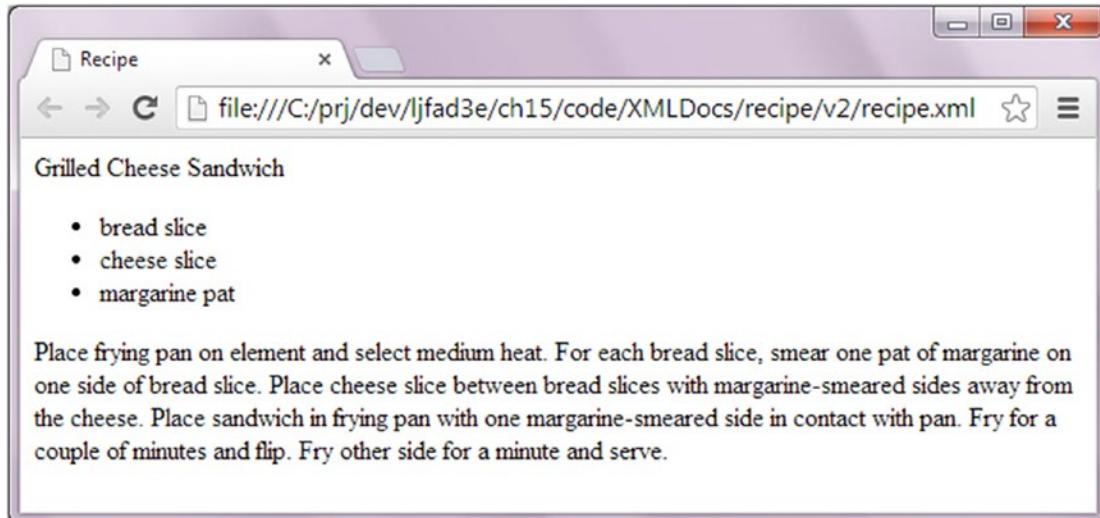


Figure 16-2. Google Chrome presents the recipe data via XHTML tags

A tag's attributes don't need to be prefixed when those attributes belong to the element. For example, `qty` isn't prefixed in `<r:ingredient qty="2">`. However, a prefix is required for attributes belonging to other namespaces. For example, suppose you want to add an XHTML style attribute to the document's `<r:title>` tag to provide styling for the recipe title when displayed via an application. You can accomplish this task by inserting an XHTML attribute into the `title` tag, as follows:

```
<r:title h:style="font-family: sans-serif;">
```

The XHTML style attribute has been prefixed with `h:` because this attribute belongs to the XHTML language namespace and not to the recipe language namespace.

When multiple namespaces are involved, it can be convenient to specify one of these namespaces as the default namespace to reduce the tedium in entering namespace prefixes. Consider Listing 16-6.

Listing 16-6. Specifying a Default Namespace

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:r="http://www.tutortutor.ca/">
  <head>
    <title>
      Recipe
    </title>
  </head>
  <body>
    <r:recipe>
      <r:title>
        Grilled Cheese Sandwich
      </r:title>
      <r:ingredients>
        <ul>
          <li>
            <r:ingredient qty="2">
              bread slice
            </r:ingredient>
          </li>
          <li>
            <r:ingredient>
              cheese slice
            </r:ingredient>
          </li>
          <li>
            <r:ingredient qty="2">
              margarine pat
            </r:ingredient>
          </li>
        </ul>
      </r:ingredients>
      <p>
        <r:instructions>
```

```

Place frying pan on element and select medium heat. For each bread
slice, smear one pat of margarine on one side of bread slice.
Place cheese slice between bread slices with margarine-smeared
sides away from the cheese. Place sandwich in frying pan with one
margarine-smeared side in contact with pan. Fry for a couple of
minutes and flip. Fry other side for a minute and serve.

</r:instructions>
</p>
</r:recipe>
</body>
</html>
```

Listing 16-6 specifies a default namespace for the XHTML language. No XHTML element tag needs to be prefixed with h:. However, recipe language element tags must still be prefixed with the r: prefix.

Comment and Processing Instructions

XML documents can contain *comments*, which are character sequences beginning with <!-- and ending with -->. For example, you might place <!-- Todo --> in Listing 16-3's body element to remind yourself that you need to finish coding this element.

Comments are used to clarify portions of a document. They can appear anywhere after the XML declaration except within tags; they cannot be nested, cannot contain a double hyphen (--) because doing so might confuse an XML parser that the comment has been closed, and shouldn't contain a hyphen (-) for the same reason, and they are typically ignored during processing. Comments are not content.

XML also permits processing instructions to be present. A *processing instruction* is an instruction that's made available to the application parsing the document. The instruction begins with <? and ends with ?>. The <? prefix is followed by a name known as the *target*. This name typically identifies the application to which the processing instruction is intended. The rest of the processing instruction contains text in a format appropriate to the application. Two examples of processing instructions are <?xml-stylesheet href="modern.xsl" type="text/xml"?> (associate an eXtensible

Stylesheet Language [XSL] style sheet [see <http://en.wikipedia.org/wiki/XSL>] with an XML document) and <?php /* PHP code */ ?> (pass a PHP [see <http://en.wikipedia.org/wiki/Php>] code fragment to the application). Although the XML declaration looks like a processing instruction, this isn't the case.

Note The XML declaration isn't a processing instruction.

Well-Formed Documents

HTML is a sloppy language in which elements can be specified out of order, end tags can be omitted, and so on. The complexity of a web browser's page layout code is partly due to the need to handle these special cases. In contrast, XML is a much stricter language. To make XML documents easier to parse, XML mandates that XML documents follow certain rules:

- *All elements must either have start and end tags or consist of empty-element tags.* For example, unlike the HTML `<p>` tag that's often specified without a `</p>` counterpart, `</p>` must also be present from an XML document perspective.
- *Tags must be nested correctly.* For example, while you'll probably get away with specifying `<i>Android</i>` in HTML, an XML parser would report an error. In contrast, `<i>Android</i>` doesn't result in an error.
- *All attribute values must be quoted.* Either single quotes ('') or double quotes ("") are permissible (although double quotes are the more commonly specified quotes). It's an error to omit these quotes.
- *Empty elements must be properly formatted.* For example, HTML's `
` tag would have to be specified as `
` in XML. You can specify a space between the tag's name and the / character, although the space is optional.
- *Be careful with case.* XML is a case-sensitive language in which tags differing in case (such as `<Author>` and `<Author>`) are considered different. It's an error to mix start and end tags of different cases, for example, `<Author>` with `</Author>`.

XML parsers that are aware of namespaces enforce two additional rules:

- All element and attribute names must not include more than one colon character.
- No entity names, processing instruction targets, or notation names (discussed later) can contain colons.

An XML document that conforms to these rules is *well formed*. The document has a logical and clean appearance and is much easier to process. XML parsers will only parse well-formed XML documents.

Valid Documents

It's not always enough for an XML document to be well formed; in many cases, the document must also be valid. A *valid* document adheres to constraints. For example, a constraint could be placed upon Listing 16-1's recipe document to ensure that the `ingredients` element always precedes the `instructions` element; perhaps an application must first process `ingredients`.

Note XML document validation is similar to a compiler analyzing source code to make sure that the code makes sense in a machine context. For example, each of `int`, `count`, `=`, `1`, and `;` are valid Java character sequences but `1 count ;` `int =` isn't a valid Java construct (whereas `int count = 1;` is a valid Java construct).

Some XML parsers perform validation, whereas other parsers don't because validating parsers are harder to write. A parser that performs validation compares an XML document to a grammar document. Any deviation from this document is reported as an error to the application; the document isn't valid. The application may choose to fix the error or reject the document. Unlike well-formedness errors, validity errors aren't necessarily fatal and the parser can continue to parse the document.

Note Validating XML parsers often don't validate by default because validation can be time-consuming. They must be instructed to perform validation.

Grammar documents are written in a special language. Two commonly used grammar languages are document type definition and XML Schema.

Document Type Definition

Document type definition (DTD) is the oldest grammar language for specifying an XML document's grammar. DTD grammar documents (known as DTDs) are written in accordance with a strict syntax that states what elements may be present and in what parts of a document. It also states what is contained within elements (child elements, content, or mixed content) and what attributes may be specified. For example, a DTD may specify that a `recipe` element must have an `ingredients` element followed by an `instructions` element.

Introducing DTD is beyond the scope of the book. If you need to know more of DTD, please have a look at https://en.wikipedia.org/wiki/Document_type_definition as a starting point.

XML Schema

XML Schema is a grammar language for declaring the structure, content, and *semantics* (meaning) of an XML document. This language's grammar documents are known as *schemas* that are themselves XML documents. Schemas must conform to the XML Schema DTD (see www.w3.org/2001/XMLSchema.dtd).

XML Schema was introduced by the W3C to overcome limitations with DTD, such as DTD's lack of support for namespaces. Also, XML Schema provides an object-oriented approach to declaring an XML document's grammar. This grammar language provides a much larger set of primitive types than DTD's CDATA and PCDATA types. For example, you'll find integer, floating-point, various date and time, and string types to be part of XML Schema.

Note XML Schema predefines 19 primitive types, which are expressed via the following identifiers: `anyURI`, `base64Binary`, `boolean`, `date`, `dateTime`, `decimal`, `double`, `duration`, `float`, `hexBinary`, `gDay`, `gMonth`, `gMonthDay`, `gYear`, `gYearMonth`, `NOTATION`, `QName`, `string`, and `time`.

Further introducing schemas is beyond the scope of this book. Have a look at https://en.wikipedia.org/wiki/XML_schema to learn more about schemas.

Parsing XML Documents with SAX

Simple API for XML (SAX) is an event-based API for parsing an XML document sequentially from start to finish. As a SAX-oriented parser encounters an item from the document's *infoset* (an abstract data model describing an XML document's information; see http://en.wikipedia.org/wiki/XML_Information_Set), it makes this item available to an application as an *event* by calling one of the methods in one of the application's *handlers* (an object whose methods are called by the parser to make event information available), which the application has previously registered with the parser. The application can then *consume* this event by processing the infoset item in some manner.

Note According to its official website (www.saxproject.org), SAX originated as an XML parsing API for Java. However, SAX isn't exclusive to Java. Microsoft also supports SAX for its .NET framework (see <http://saxdotnet.sourceforge.net>).

After taking you on a tour of the SAX API, we provide a simple demonstration of this API to help you become familiar with its event-based parsing paradigm. We then show you how to create a custom entity resolver.

Exploring the SAX API

SAX exists in two major versions. Java implements SAX 1 through the `javax.xml.parsers` package's abstract `SAXParser` and `SAXParserFactory` classes, and it implements SAX 2 through the `org.xml.sax` package's `XMLReader` interface and through the `org.xml.sax.helpers` package's `XMLReaderFactory` class. The `org.xml.sax`, `org.xml.sax.ext`, and `org.xml.sax.helpers` packages provide various types that augment both Java implementations.

Note We explore only the SAX 2 implementation because SAX 2 makes available additional infoset items about an XML document (such as comments and CDATA section notifications).

Classes that implement the `XMLReader` interface describe SAX 2-based parsers. Instances of these classes are obtained by calling the `XMLReaderFactory` class's `createXMLReader()` methods. For example, the following code fragment invokes this class's `XMLReader createXMLReader()` class method to create and return an `XMLReader` instance:

```
XMLReader xmlr = XMLReaderFactory.createXMLReader();
```

This method call returns an instance of an `XMLReader`-implementing class and assigns its reference to `xmlr`.

Note Behind the scenes, `createXMLReader()` attempts to create an `XMLReader` instance from system defaults according to a lookup procedure that first examines the `org.xml.sax.driver` system property to see if it has a value. If so, this property's value is used as the name of the class that implements `XMLReader`. Furthermore, an attempt to instantiate this class and return the instance is made. An instance of the `org.xml.sax.SAXException` class is thrown when `createXMLReader()` cannot obtain an appropriate class or instantiate the class.

The returned `XMLReader` object makes available several methods for configuring the parser and parsing a document's content. For the methods, please consult the SAX API documentation.

After obtaining an `XMLReader` instance, you can configure that instance by setting its features and properties. A *feature* is a name-value pair that describes a parser mode, such as validation. In contrast, a *property* is a name-value pair that describes some other aspect of the parser interface, such as a lexical handler that augments the content handler by providing callback methods for reporting on comments, CDATA delimiters, and a few other syntactic constructs.

Features and properties have names, which must be absolute URIs beginning with the `http://` prefix. A feature's value is always a Boolean true/false value. In contrast, a property's value is an arbitrary object. The following example demonstrates setting a feature and a property:

```
xmlr.setFeature("http://xml.org/sax/features/validation", true);
xmlr.setProperty("http://xml.org/sax/properties/lexical-handler",
    new LexicalHandler() { /* ... */});
```

The `setFeature()` call enables the validation feature so that the parser will perform validation. Feature names are prefixed with <http://xml.org/sax/features/>.

Note Parsers must support the namespaces and namespace-prefixes features. `namespaces` decides whether URLs and local names are passed to `ContentHandler`'s `startElement()` and `endElement()` methods. It defaults to `true`; these names are passed. The parser can pass empty strings when `false`. `namespace-prefixes` decides whether a namespace declaration's `xmlns` and `xmlns:prefix` attributes are included in the `Attributes` list passed to `startElement()`, and it also decides whether qualified names are passed as the method's third argument; a *qualified name* is a prefix plus a local name. It defaults to `false`, meaning that `xmlns` and `xmlns:prefix` aren't included, and meaning that parsers don't have to pass qualified names. No properties are mandatory. The JDK documentation's `org.xml.sax` package page lists standard SAX 2 features and properties.

The `setProperty()` call assigns an instance of a class that implements the `org.xml.sax.ext.LexicalHandler` interface to the `lexical-handler` property so that interface methods can be called to report on comments, CDATA sections, and so on. Property names are prefixed with <http://xml.org/sax/properties/>.

Note Unlike `ContentHandler`, `DTDHandler`, `EntityResolver`, and `ErrorHandler`, `LexicalHandler` is an extension (it's not part of the core SAX API), which is why `XMLReader` doesn't declare a `void setLexicalHandler(LexicalHandler handler)` method. If you want to install a lexical handler, you must use `XMLReader`'s `setProperty()` method to install the handler as the value of the <http://xml.org/sax/properties/lexical-handler> property.

Features and properties can be read-only or read-write. (In some rare cases, a feature or property might be write-only.) When setting or reading a feature or property, `SAXNotSupportedException` or `SAXNotRecognizedException` might be thrown. For example, if you try to modify a read-only feature/property, an instance of the `SAXNotSupportedException` class is thrown. This exception could also be thrown if

you call `setFeature()` or `setProperty()` during parsing. Trying to set the validation feature for a parser that doesn't perform validation is a scenario where an instance of the `SAXNotRecognizedException` class is thrown.

The handlers installed by `setContentHandler()`, `setDTDHandler()`, and `setErrorHandler()`, the entity resolver installed by `setEntityResolver()`, and the handler installed by the lexical-handler property/`LexicalHandler` interface provide various callback methods that you need to understand before you can codify them to respond effectively to parsing events. For details, see the API documentation.

Demonstrating the SAX API

[Listing 16-7](#) presents the source code to `SAXDemo`, an application that demonstrates the SAX API. The application consists of a `SAXDemo` entry-point class and a `Handler` subclass of `DefaultHandler2`.

Listing 16-7. `SAXDemo`

```
import java.io.FileReader;
import java.io.IOException;

import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;

import org.xml.sax.helpers.XMLReaderFactory;

public class SAXDemo {
    public static void main(String[] args) {
        if (args.length < 1 || args.length > 2) {
            System.err.println("usage: java SAXDemo xmlfile [v]");
            return;
        }
        try {
            XMLReader xmlr = XMLReaderFactory.createXMLReader();
            if (args.length == 2 && args[1].equals("v"))
                xmlr.setFeature("http://xml.org/sax/features/validation", true);
            xmlr.setFeature("http://xml.org/sax/features/namespace-prefixes",
                           true);
        }
    }
}
```

```

        Handler handler = new Handler();
        xmlr.setContentHandler(handler);
        xmlr.setDTDHandler(handler);
        xmlr.setEntityResolver(handler);
        xmlr.setErrorHandler(handler);
        xmlr.setProperty("http://xml.org/sax/properties/lexical-handler",
                         handler);
        xmlr.parse(new InputSource(new FileReader(args[0])));
    } catch (IOException ioe) {
        System.err.println("IOE: " + ioe);
    } catch (SAXException saxe) {
        System.err.println("SAXE: " + saxe);
    }
}
}
}

```

SAXDemo's main() method first verifies that one or two command-line arguments (the name of an XML document optionally followed by lowercase letter v, which tells SAXDemo to create a validating parser) have been specified. It then creates an XMLReader instance; conditionally enables the validation feature and enables the namespace-prefixes feature; instantiates the companion Handler class; installs this Handler instance as the parser's content handler, DTD handler, entity resolver, and error handler; installs this Handler instance as the value of the lexical-handler property; creates an input source to read the document from a file; and parses the document.

The Handler class's source code is presented in Listing 16-8.

Listing 16-8. Handler

```

import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.Locator;
import org.xml.sax.SAXParseException;

import org.xml.sax.ext.DefaultHandler2;

public class Handler extends DefaultHandler2 {
    private Locator locator;
}

```

```
@Override
public void characters(char[] ch, int start, int length) {
    System.out.print("characters() [");
    for (int i = start; i < start + length; i++)
        System.out.print(ch[i]);
    System.out.println("]");
}

@Override
public void comment(char[] ch, int start, int length) {
    System.out.print("characters() [");
    for (int i = start; i < start + length; i++)
        System.out.print(ch[i]);
    System.out.println("]");
}

@Override
public void endCDATA() {
    System.out.println("endCDATA()");
}

@Override
public void endDocument() {
    System.out.println("endDocument()");
}

@Override
public void endDTD() {
    System.out.println("endDTD()");
}

@Override
public void endElement(String uri, String localName, String qName) {
    System.out.print("endElement() ");
    System.out.print("uri=[ " + uri + "], ");
    System.out.print("localName=[ " + localName + "], ");
    System.out.println("qName=[ " + qName + "]");
}
```

```
@Override
public void endEntity(String name) {
    System.out.print("endEntity() ");
    System.out.println("name=[ " + name + " ]");
}

@Override
public void endPrefixMapping(String prefix) {
    System.out.print("endPrefixMapping() ");
    System.out.println("prefix=[ " + prefix + " ]");
}

@Override
public void error(SAXParseException saxpe) {
    System.out.println("error() " + saxpe);
}

@Override
public void fatalError(SAXParseException saxpe) {
    System.out.println("fatalError() " + saxpe);
}

@Override
public void ignorableWhitespace(char[] ch, int start, int length) {
    System.out.print("ignorableWhitespace() [ ");
    for (int i = start; i < start + length; i++)
        System.out.print(ch[i]);
    System.out.println("] ");
}

@Override
public void notationDecl(String name, String publicId, String systemId) {
    System.out.print("notationDecl() ");
    System.out.print("name=[ " + name + " ]");
    System.out.print("publicId=[ " + publicId + " ]");
    System.out.println("systemId=[ " + systemId + " ]");
}
```

```
@Override
public void processingInstruction(String target, String data) {
    System.out.print("processingInstruction() [");
    System.out.println("target=[" + target + "]");
    System.out.println("data=[" + data + "]");
}

@Override
public InputSource resolveEntity(String publicId, String systemId) {
    System.out.print("resolveEntity() ");
    System.out.print("publicId=[" + publicId + "]");
    System.out.println("systemId=[" + systemId + "]");
    // Do not perform a remapping.
    InputSource is = new InputSource();
    is.setPublicId(publicId);
    is.setSystemId(systemId);
    return is;
}

@Override
public void setDocumentLocator(Locator locator) {
    System.out.print("setDocumentLocator() ");
    System.out.println("locator=[" + locator + "]");
    this.locator = locator;
}

@Override
public void skippedEntity(String name) {
    System.out.print("skippedEntity() ");
    System.out.println("name=[" + name + "]");
}

@Override
public void startCDATA() {
    System.out.println("startCDATA()");
}
```

```
@Override
public void startDocument() {
    System.out.println("startDocument()");
}

@Override
public void startDTD(String name, String publicId, String systemId) {
    System.out.print("startDTD() ");
    System.out.print("name=[" + name + "]");
    System.out.print("publicId=[" + publicId + "]");
    System.out.println("systemId=[" + systemId + "]");
}

@Override
public void startElement(String uri, String localName, String qName,
                        Attributes attributes)
{
    System.out.print("startElement() ");
    System.out.print("uri=[" + uri + "], ");
    System.out.print("localName=[" + localName + "], ");
    System.out.println("qName=[" + qName + "]");
    for (int i = 0; i < attributes.getLength(); i++)
        System.out.println(" Attribute: " + attributes.getLocalName(i) +
                           ", " + attributes.getValue(i));
    System.out.println("Column number=[" + locator.getColumnNumber() + "]");
    System.out.println("Line number=[" + locator.getLineNumber() + "]");
}

@Override
public void startEntity(String name) {
    System.out.print("startEntity() ");
    System.out.println("name=[" + name + "]");
}
```

```
@Override
public void startPrefixMapping(String prefix, String uri) {
    System.out.print("startPrefixMapping() ");
    System.out.print("prefix=[" + prefix + "]");
    System.out.println("uri=[" + uri + "]");
}

@Override
public void unparsedEntityDecl(String name, String publicId,
                               String systemId, String notationName)
{
    System.out.print("unparsedEntityDecl() ");
    System.out.print("name=[" + name + "]");
    System.out.print("publicId=[" + publicId + "]");
    System.out.print("systemId=[" + systemId + "]");
    System.out.println("notationName=[" + notationName + "]");
}

@Override
public void warning(SAXParseException saxpe) {
    System.out.println("warning() " + saxpe);
}
}
```

The Handler subclass is pretty straightforward; it outputs every possible piece of information about an XML document, subject to feature and property settings. You'll find this class handy for exploring the order in which events occur along with various features and properties.

After compiling `SAXDemo.java` and `Handler.java` (`javac SAXDemo.java`), execute the following command to parse Listing 16-4's `svg-examples.xml` document:

```
java SAXDemo svg-examples.xml
```

SAXDemo responds by presenting the following output (the hash code may be different):

```
setDocumentLocator() locator=[com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$LocatorProxy@1395ddba]
startDocument()
startElement() uri=[], localName=[svg-examples], qName=[svg-examples]
Column number=[15]
Line number=[2]
characters() [
    ]
startElement() uri=[], localName=[example], qName=[example]
Column number=[13]
Line number=[3]
characters() [
    The following Scalable Vector Graphics document describes a blue-
    filled and ]
characters() [
    black-stroked rectangle.
    ]
startCDATA()
characters() [<svg width="100%" height="100%" version="1.1"
    xmlns="http://www.w3.org/2000/svg">
    <rect width="300" height="100"
        style="fill:rgb(0,0,255);stroke-width:1;
        stroke:rgb(0,0,0)"/>
    </svg>]
endCDATA()
characters() [
    ]
endElement() uri=[], localName=[example], qName=[example]
characters() [
    ]
endElement() uri=[], localName=[svg-examples], qName=[svg-examples]
endDocument()
```

The first output line proves that `setDocumentLocator()` is called first. It also identifies the `Locator` instance whose `getColumnNumber()` and `getLineNumber()` methods are called to output the parser location when `startElement()` is called; these methods return column and line numbers starting at 1.

Perhaps you're curious about the three instances of the following output:

```
characters() [
]
```

The instance of this output that follows the `endCDATA()` output is reporting a carriage return/line feed combination that wasn't included in the preceding `characters()` method call, which was passed the contents of the CDATA section minus these line terminator characters. In contrast, the instances of this output that follow the `startElement()` call for `svg-examples` and follow the `endElement()` call for `example` are somewhat curious. There's no content between `<svg-examples>` and `<example>`, and between `</example>` and `</svg-examples>`, or is there?

You can satisfy this curiosity by modifying `svg-examples.xml` to include an internal DTD. Place the following DTD (which indicates that an `svg-examples` element contains one or more `example` elements, and that an `example` element contains parsed character data) between the XML declaration and the `<svg-examples>` start tag:

```
<!DOCTYPE svg-examples [
<!ELEMENT svg-examples (example+)
<!ELEMENT example (#PCDATA)
]>
```

Continuing, execute the following command:

```
java SAXDemo svg-examples.xml
```

This time, you should see the following output (although the hash code will probably differ):

```
setDocumentLocator() locator=[com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser$LocatorProxy@540fe861]
startDocument()
startDTD() name=[svg-examples]publicId=[null]systemId=[null]
endDTD()
```

CHAPTER 16 WORKING WITH XML AND JSON DOCUMENTS

```
startElement() uri=[], localName=[svg-examples], qName=[svg-examples]
Column number=[15]
Line number=[6]
ignorableWhitespace() [
    ]
startElement() uri=[], localName=[example], qName=[example]
Column number=[13]
Line number=[7]
characters() [
    The following Scalable Vector Graphics document describes a blue-
    filled and
    black-stroked rectangle.]
characters() [
    ]
startCDATA()
characters() [<svg width="100%" height="100%" version="1.1"
    xmlns="http://www.w3.org/2000/svg">
    <rect width="300" height="100"
        style="fill:rgb(0,0,255);stroke-width:1;
        stroke:rgb(0,0,0)"/>
</svg>]
endCDATA()
characters() [
    ]
endElement() uri=[], localName=[example], qName=[example]
ignorableWhitespace() [
]
endElement() uri=[], localName=[svg-examples], qName=[svg-examples]
endDocument()
```

This output reveals that the `ignorableWhitespace()` method was called after `startElement()` for `svg-examples` and after `endElement()` for `example`. The former two calls to `characters()` that produced the strange output were reporting ignorable whitespace.

Recall that we previously defined *ignorable whitespace* as whitespace located between tags where the DTD doesn't allow mixed content. For example, the DTD indicates that `svg-examples` shall contain only `example` elements, not `example` elements and parsed character data. However, the line terminator following the `<svg-examples>` tag and the leading whitespace before `<example>` are parsed character data. The parser now reports these characters by calling `ignorableWhitespace()`.

This time, there are only two occurrences of the following output:

```
characters() [
]
```

The first occurrence reports the line terminator separately from the `example` element's text (before the CDATA section); it didn't do so previously, which proves that `characters()` is called with either all or part of an element's content. Once again, the second occurrence reports the line terminator that follows the CDATA section.

Let's validate `svg-examples.xml` without the previously presented internal DTD. You'll do so by executing the following command; don't forget to include the `v` command-line argument or the document won't validate:

```
java SAXDemo svg-examples.xml v
```

Among its output are a couple of `error()`-prefixed lines that are similar to those shown here:

```
error() org.xml.sax.SAXParseException; lineNumber: 2; columnNumber: 14;
Document is invalid: no grammar found.
error() org.xml.sax.SAXParseException; lineNumber: 2; columnNumber: 14;
Document root element "svg-examples", must match DOCTYPE root "null".
```

These lines reveal that a DTD grammar hasn't been found. Furthermore, the parser reports a mismatch between `svg-examples` (it considers the first encountered element to be the root element) and `null` (it considers `null` to be the name of the root element in the absence of a DTD). Neither violation is considered to be fatal, which is why `error()` is called instead of `fatalError()`.

Add the internal DTD to `svg-examples.xml` and reexecute `java SAXDemo svg-examples.xml v`. This time, you should see no `error()`-prefixed lines in the output.

Tip SAX 2 validation defaults to validating against a DTD. To validate against an XML Schema-based schema instead, add the `schemaLanguage` property with the `http://www.w3.org/2001/XMLSchema` value to the `XMLReader` instance. Accomplish this task for `SAXDemo` by specifying `xmlr.setProperty("http://java.sun.com/xml/jaxp/properties/schemaLanguage", "http://www.w3.org/2001/XMLSchema");` before `xmlr.parse(new InputSource(new FileReader(args[0])));`.

Parsing and Creating XML Documents with DOM

Document Object Model (DOM) is an API for parsing an XML document into an in-memory tree of nodes and for creating an XML document from a tree of nodes. After a DOM parser has created a document tree, an application uses the DOM API to navigate over and extract infoset items from the tree's nodes.

Note DOM originated as an object model for the Netscape Navigator 3 and Microsoft Internet Explorer 3 web browsers. Collectively, these implementations are known as DOM Level 0. Because each vendor's DOM implementation was only slightly compatible with the other, the W3C subsequently took charge of DOM's development to promote standardization and has so far released DOM Levels 1, 2, and 3 (with Level 4 under development). Java 7 and newer versions of Android support all three DOM levels through their DOM APIs.

DOM has two big advantages over SAX. First, DOM permits random access to a document's infoset items, whereas SAX only permits serial access. Second, DOM lets you also create XML documents, whereas you can only parse documents with SAX. However, SAX is advantageous over DOM in that it can parse documents of arbitrary size, whereas the size of documents parsed or created by DOM is limited by the amount of available memory for storing the document's node-based tree structure.

In this section, we first introduce you to DOM's tree structure. We then take you on a tour of the DOM API; you learn how to use this API to parse and create XML documents.

A Tree of Nodes

DOM views an XML document as a tree that's composed of several kinds of nodes. This tree has a single root node, and all nodes except for the root have a parent node. Also, each node has a list of child nodes. When this list is empty, the child node is known as a *leaf node*.

Note DOM permits nodes to exist that are not part of the tree structure. For example, an element node's attribute nodes are not regarded as child nodes of the element node. Also, nodes can be created but not inserted into the tree; they can also be removed from the tree.

Each node has a *node name*, which is the complete name for nodes that have names (such as an element's or an attribute's prefixed name), and `#node-type` for unnamed nodes, where *node-type* is one of `cdata-section`, `comment`, `document`, `document-fragment`, or `text`. Nodes also have *local names* (names without prefixes), prefixes, and namespace URIs (although these attributes may be null for certain kinds of nodes, such as comments). Finally, nodes have string values, which happen to be the content of text nodes, comment nodes, and similar text-oriented nodes; normalized values of attributes; and null for everything else.

DOM classifies nodes into 12 types, of which 7 types can be considered part of a DOM tree. All of these types are described as follows:

- *Attribute node*: One of an element's attributes. It has a name, a local name, a prefix, a namespace URI, and a normalized string value. The value is *normalized* by resolving any entity references and by converting sequences of whitespace to a single whitespace character. An attribute node has children, which are the text and any entity reference nodes that form its value. Attribute nodes are not regarded as children of their associated element nodes.
- *CDATA section node*: The contents of a CDATA section. Its name is `#cdata-section`, and its value is the CDATA section's text.

- *Comment node*: A document comment. Its name is #comment, and its value is the comment text. A comment node has a parent, which is the node that contains the comment.
- *Document node*: The root of a DOM tree. Its name is #document, it always has a single element node child, and it will also have a document type child node when the document has a document type declaration. Furthermore, it can have additional child nodes describing comments or processing instructions that appear before or after the root element's start tag. There can be only one document node in the tree.
- *Document fragment node*: An alternative root node. Its name is #document-fragment, and it contains anything that an element node can contain (such as other element nodes and even comment nodes). A parser never creates this kind of a node. However, an application can create a document fragment node when it extracts part of a DOM tree to be moved somewhere else. Document fragment nodes let you work with subtrees.
- *Document type node*: A document type declaration. Its name is the name specified by the document type declaration for the root element. Also, it has a (possibly null) public identifier, a required system identifier, an internal DTD subset (which is possibly null), a parent (the document node that contains the document type node), and lists of DTD-declared notations and general entities. Its value is always set to null.
- *Element node*: A document's element. It has a name, a local name, a (possibly null) prefix, and a namespace URI, which is null when the element doesn't belong to any namespace. An element node contains children, including text nodes, and even comment and processing instruction nodes.
- *Entity node*: The parsed and unparsed entities that are declared in a document's DTD. When a parser reads a DTD, it attaches a map of entity nodes (indexed by entity name) to the document type node. An entity node has a name and a system identifier, and it can also have a

public identifier if one appears in the DTD. Finally, when the parser reads the entity, the entity node is given a list of read-only child nodes that contain the entity's replacement text.

- *Entity reference node*: A reference to a DTD-declared entity. Each entity reference node has a name, and it is included in the tree when the parser doesn't replace entity references with their values. The parser never includes entity reference nodes for character references (such as & or Σ) because they're replaced by their respective characters and included in a text node.
- *Notation node*: A DTD-declared notation. A parser that reads the DTD attaches a map of notation nodes (indexed by notation name) to the document type node. Each notation node has a name and a public identifier or a system identifier, whichever identifier was used to declare the notation in the DTD. Notation nodes don't have children.
- *Processing instruction node*: A processing instruction that appears in the document. It has a name (the instruction's target), a string value (the instruction's data), and a parent (its containing node).
- *Text node*: Document content. Its name is #text, and it represents a portion of an element's content when an intervening node (such as a comment) must be created. Characters such as < and & that are represented in the document via character references are replaced by the literal characters they represent. When these nodes are written to a document, these characters must be escaped.

Although these node types store considerable information about an XML document, there are limitations (such as not exposing whitespace outside of the root element). In contrast, most DTD or schema information, such as element types (`<!ELEMENT...>`) and attribute types (`<xs:attribute...>`), cannot be accessed through the DOM.

DOM Level 3 addresses some of the DOM's various limitations. For example, although DOM doesn't provide a node type for the XML declaration, DOM Level 3 makes it possible to access the XML declaration's version, encoding, and standalone attribute values via attributes of the document node.

Note Nonroot nodes never exist in isolation. For example, it's never the case for an element node not to belong to a document or to a document fragment. Even when such nodes are disconnected from the main tree, they remain aware of the document or document fragment to which they belong.

Exploring the DOM API

Java implements DOM through the `javax.xml.parsers` package's abstract `DocumentBuilder` and `DocumentBuilderFactory` classes along with the nonabstract `FactoryConfigurationError` and `ParserConfigurationException` classes. The `org.w3c.dom`, `org.w3c.dom.bootstrap` (not supported by Android), `org.w3c.dom.events` (not supported by Android), and `org.w3c.dom.ls` packages provide various types that augment this implementation.

The first step in working with DOM is to instantiate `DocumentBuilderFactory` by calling one of its `newInstance()` methods. For example, the following code fragment invokes `DocumentBuilderFactory`'s `DocumentBuilderFactory newInstance()` class method:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```

Behind the scenes, `newInstance()` follows an ordered lookup procedure to identify the `DocumentBuilderFactory` implementation class to load. This procedure first examines the `javax.xml.parsers.DocumentBuilderFactory` system property and lastly chooses the Java platform's default `DocumentBuilderFactory` implementation class when no other class is found. If an implementation class isn't available (perhaps the class identified by the `javax.xml.parsers.DocumentBuilderFactory` system property doesn't exist) or cannot be instantiated, `newInstance()` throws an instance of the `FactoryConfigurationError` class. Otherwise, it instantiates the class and returns its instance.

After obtaining a `DocumentBuilderFactory` instance, you can call various configuration methods to configure the factory. For example, you could call `DocumentBuilderFactory`'s `void setNamespaceAware(boolean awareness)` method with a true argument to tell the factory that any returned parser (known as a *document builder* to DOM) must provide support for XML namespaces. You can also call `void setValidating(boolean validating)` with true as the argument to validate documents against their DTDs, or call `void setSchema(Schema schema)` to validate documents against the `javax.xml.validation.Schema` instance identified by `schema`.

VALIDATION API

Schema is a member of the Validation API, which decouples document parsing from validation, making it easier for applications to take advantage of specialized validation libraries that support additional schema languages (such as RELAX NG—see http://en.wikipedia.org/wiki/RELAX_NG), and also making it easier to specify the location of a schema.

The Validation API is associated with the javax.xml.validation package, which also includes SchemaFactory, SchemaFactoryLoader, TypeInfoProvider, Validator, and ValidatorHandler. Schema is the central class, and it represents an immutable in-memory representation of a grammar.

DOM supports the Validation API via DocumentBuilderFactory's void setSchema(Schema schema) and Schema getSchema() methods. Similarly, SAX 1.0 supports Validation via SAXParserFactory's void setSchema(Schema schema) and Schema getSchema() methods. SAX 2.0 and StAX don't support the Validation API.

The following example provides a demonstration of the Validation API in a DOM context:

```
// Parse an XML document into a DOM tree.  
DocumentBuilder parser =  
    DocumentBuilderFactory.newInstance().newDocumentBuilder();  
Document document = parser.parse(new File("instance.xml"));  
// Create a SchemaFactory capable of understanding W3C XML Schema (WXS).  
SchemaFactory factory =  
    SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);  
// Load a WXS schema, represented by a Schema instance.  
Source schemaFile = new StreamSource(new File("mySchema.xsd"));  
Schema schema = factory.newSchema(schemaFile);  
// Create a Validator instance, which is used to validate an XML document.  
Validator validator = schema.newValidator();  
// Validate the DOM tree.  
try {  
    validator.validate(new DOMSource(document));  
} catch (SAXException saxe) {  
    // XML document is invalid!  
}
```

This example refers to XSLT types such as Source. We explore XSLT later in this chapter.

After the factory has been configured, call its `DocumentBuilder newDocumentBuilder()` method to return a document builder that supports the configuration, as demonstrated here:

```
DocumentBuilder db = dbf.newDocumentBuilder();
```

If a document builder cannot be returned (perhaps the factory cannot create a document builder that supports XML namespaces), this method throws a `ParserConfigurationException` instance.

Assuming that you've successfully obtained a document builder, what happens next depends upon whether you want to parse or create an XML document.

Parsing XML Documents

`DocumentBuilder` provides several overloaded `parse()` methods for parsing an XML document into a node tree. These methods differ in how they obtain the document. For example, `Document parse(String uri)` parses the document that's identified by its string-based URI argument.

Note Each `parse()` method throws `java.lang.IllegalArgumentException` when `null` is passed as the method's first argument, `IOException` when an input/output problem occurs, and `SAXException` when the document cannot be parsed. This last exception type implies that `DocumentBuilder`'s `parse()` methods rely on SAX to take care of the actual parsing work. Because they are more involved in building the node tree, DOM parsers are commonly referred to as *document builders*.

The returned `org.w3c.dom.Document` object provides access to the parsed document through methods such as `DocumentType getDoctype()`, which makes the document type declaration available through the `org.w3c.dom.DocumentType` interface. Conceptually, `Document` is the root of the document's node tree.

Note Apart from `DocumentBuilder`, `DocumentBuilderFactory`, and a few other classes, DOM is based on interfaces, of which `Document` and `DocumentType` are examples. Behind the scenes, DOM methods (such as the `parse()` methods) return objects whose classes implement these interfaces.

Document and all other `org.w3c.dom` interfaces that describe different kinds of nodes are subinterfaces of the `org.w3c.dom.Node` interface. As such, they inherit `Node`'s constants and methods.

`Node` declares 12 constants that represent the various kinds of nodes; `ATTRIBUTE_NODE` and `ELEMENT_NODE` are examples. When you want to identify the kind of node represented by a given `Node` object, call `Node`'s short `getNodeType()` method and compare the returned value to one of these constants.

Note The rationale for using `getNodeType()` and these constants, instead of using `instanceof` and a class name, is that DOM (the object model, not the Java DOM API) was designed to be language independent, and languages such as AppleScript don't have the equivalent of `instanceof`.

`Node` declares several methods for getting and setting common node properties. These methods include `String getNodeName()`, `String getLocalName()`, `String getNamespaceURI()`, `String getPrefix()`, `void setPrefix(String prefix)`, `String getNodeValue()`, and `void setNodeValue(String nodeValue)`, which let you get and (for some properties) set a node's name (such as `#text`), local name, namespace URI, prefix, and normalized string value properties.

Note Various `Node` methods (such as `setPrefix()` and `getNodeValue()`) throw an instance of the `org.w3c.dom.DOMException` class when something goes wrong. For example, `setPrefix()` throws this exception when the prefix argument contains an illegal character, the node is read-only, or the argument is malformed. Similarly, `getNodeValue()` throws `DOMException` when `getNodeValue()` would return more characters than can fit into a `DOMString` (a W3C type) variable on the implementation platform. `DOMException` declares a series of constants (such as `DOMSTRING_SIZE_ERR`) that classify the reason for the exception.

Node declares several methods for navigating the node tree. Three of its navigation methods are as follows:

- `boolean hasChildNodes()` returns true when a node has child nodes.
- `Node getChildNode()` returns the node's first child.
- `Node getLastChild()` returns the node's last child.

For nodes with multiple children, you'll find the `NodeList getChildNodes()` method to be handy. This method returns an `org.w3c.dom.NodeList` instance whose `int getLength()` method returns the number of nodes in the list and whose `Node item(int index)` method returns the node at the `index`th position in the list (or null when `index`'s value isn't valid; it's less than 0 or greater than or equal to `getLength()`'s value).

Node declares four methods for modifying the tree by inserting, removing, replacing, and appending child nodes:

- `Node insertBefore(Node newChild, Node refChild)` inserts `newChild` before the existing node specified by `refChild` and returns `newChild`.
- `Node removeChild(Node oldChild)` removes the child node identified by `oldChild` from the tree and returns `oldChild`.
- `Node replaceChild(Node newChild, Node oldChild)` replaces `oldChild` with `newChild` and returns `oldChild`.
- `Node appendChild(Node newChild)` adds `newChild` to the end of the current node's child nodes and returns `newChild`.

Finally, Node declares several utility methods, including `Node cloneNode(boolean deep)` (create and return a duplicate of the current node, recursively cloning its subtree when `true` is passed to `deep`) and `void normalize()` (descend the tree from the given node and merge all adjacent text nodes, deleting those text nodes that are empty).

Tip To obtain an element node's attributes, first call Node's `NamedNodeMap getAttributes()` method. This method returns an `org.w3c.dom.NamedNodeMap` implementation when the node represents an element; otherwise,

it returns null. Along with declaring methods for accessing these nodes by name (such as `Node getNamedItem(String name)`), `NamedNodeMap` declares `int getLength()` and `Node item(int index)` methods for returning all attribute nodes by `index`. You would then obtain the `Node`'s name by calling a method such as `getnodeName()`.

Beyond inheriting `Node`'s constants and methods, `Document` declares its own methods. For example, you can call `Document`'s `String getXmlEncoding()`, `boolean getXmlStandalone()`, and `String getXmlVersion()` methods to return the XML declaration's encoding, standalone, and version attribute values, respectively.

`Document` declares three methods for locating one or more elements:

- `Element getElementById(String elementId)` returns the element that has an `id` attribute (as in ``) matching the value specified by `elementId`.
- `NodeList getElementsByTagName(String tagname)` returns a nodelist of a document's elements (in document order) matching the specified `tagName`.
- `NodeList getElementsByTagNameNS(String namespaceURI, String localName)` is essentially the same as the second method except that only elements matching the given `localName` and `namespaceURI` are returned in the nodelist. Pass "*" to `namespaceURI` to match all namespaces; pass "*" to `localName` to match all local names.

The returned element node and each element node in the list implement the `org.w3c.dom.Element` interface. This interface declares methods to return nodelists of descendant elements in the tree, attributes associated with the element, and more. For example, `String getAttribute(String name)` returns the value of the attribute identified by `name`, whereas `Attr getAttributeNode(String name)` returns an attribute node by `name`. The returned node is an implementation of the `org.w3c.dom.Attr` interface.

You now have enough information to explore an application for parsing an XML document and outputting the element and attribute information from the resulting DOM tree. Listing 16-9 presents this application's source code.

Listing 16-9. DOMDemo (Version 1)

```
import java.io.IOException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Attr;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class DOMDemo {

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage: java DOMDemo xmlfile");
            return;
        }
        try {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            dbf.setNamespaceAware(true);
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.parse(args[0]);
            System.out.printf("Version = %s%n", doc.getXmlVersion());
            System.out.printf("Encoding = %s%n", doc.getXmlEncoding());
            System.out.printf("Standalone = %b%n%n", doc.getXmlStandalone());
            if (doc.hasChildNodes()) {
                NodeList nl = doc.getChildNodes();
                for (int i = 0; i < nl.getLength(); i++) {
                    Node node = nl.item(i);
                    if (node.getNodeType() == Node.ELEMENT_NODE)
                        dump((Element) node);
                }
            }
        } catch (Exception e) {
```

```

        e.printStackTrace(System.err);
    }
}

private static void dump(Element e) {
    System.out.printf("Element: %s, %s, %s, %s%n",
                      e.getNodeName(),
                      e.getLocalName(), e.getPrefix(),
                      e.getNamespaceURI());
    NamedNodeMap nnm = e.getAttributes();
    if (nnm != null)
        for (int i = 0; i < nnm.getLength(); i++) {
            Node node = nnm.item(i);
            Attr attr = e.getAttributeNode(node.getNodeName());
            System.out.printf(" Attribute %s = %s%n",
                              attr.getName(),
                              attr.getValue());
        }
    NodeList nl = e.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (node instanceof Element)
            dump((Element) node);
    }
}
}

```

DOMDemo's main() method first verifies that one command line argument (the name of an XML document) has been specified. It then creates a document builder factory, informs the factory that it wants a namespace-aware document builder, and has the factory return this document builder.

Continuing, main() parses the document into a node tree; outputs the XML declaration's version number, encoding, and stand-alone attribute values; and recursively dumps all element nodes (starting with the root node) and their attribute values.

Notice the use of getNode Type() in one part of this listing and instanceof in another part. The getNode Type() method call isn't necessary (it's only present for demonstration) because instanceof can be used instead. However, the cast from Node type to Element type in the dump() method calls is necessary.

Compile this source code (`javac DOMDemo.java`), and run the application to dump Listing 16-3's article XML content as follows:

```
java DOMDemo article.xml
```

You should observe the following output:

```
Version = 1.0
Encoding = null
Standalone = false

Element: article, article, null, null
    Attribute lang = en
        Attribute title = The Rebirth of JavaFX
Element: abstract, abstract, null, null
Element: code-inline, code-inline, null, null
Element: body, body, null, null
```

Each Element-prefixed line outputs the node name, followed by the local name, followed by the namespace prefix, followed by the namespace URI. The node and local names are identical because namespaces aren't being used. For the same reason, the namespace prefix and namespace URI are null.

Continuing on, execute the following command line to dump Listing 16-5's recipe content:

```
java DOMDemo recipe.xml
```

This time, you observe the following output, which includes namespace information:

```
Version = 1.0
Encoding = null
Standalone = false

Element: h:html, html, h, http://www.w3.org/1999/xhtml
    Attribute xmlns:h = http://www.w3.org/1999/xhtml
    Attribute xmlns:r = http://www.tutortutor.ca/
Element: h:head, head, h, http://www.w3.org/1999/xhtml
Element: h:title, title, h, http://www.w3.org/1999/xhtml
Element: h:body, body, h, http://www.w3.org/1999/xhtml
Element: r:recipe, recipe, r, http://www.tutortutor.ca/
```

```
Element: r:title, title, r, http://www.tutortutor.ca/
Element: r:ingredients, ingredients, r, http://www.tutortutor.ca/
Element: h:ul, ul, h, http://www.w3.org/1999/xhtml
Element: h:li, li, h, http://www.w3.org/1999/xhtml
Element: r:ingredient, ingredient, r, http://www.tutortutor.ca/
    Attribute qty = 2
Element: h:li, li, h, http://www.w3.org/1999/xhtml
Element: r:ingredient, ingredient, r, http://www.tutortutor.ca/
Element: h:li, li, h, http://www.w3.org/1999/xhtml
Element: r:ingredient, ingredient, r, http://www.tutortutor.ca/
    Attribute qty = 2
Element: h:p, p, h, http://www.w3.org/1999/xhtml
Element: r:instructions, instructions, r, http://www.tutortutor.ca/
```

Creating XML Documents

DocumentBuilder declares the abstract Document newDocument() method for creating a document tree. The returned Document object declares various “create” and other methods for creating this tree. For example, Element createElement(String tagName) creates an element named by tagName, returning a new Element object with the specified name but with its local name, prefix, and namespace URI set to null.

Listing 16-10 presents another version of the DOMDemo application that briefly demonstrates the creation of a document tree.

Listing 16-10. DOMDemo (Version 2)

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;
```

```
public class DOMDemo {  
    public static void main(String[] args) {  
        try {  
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();  
            DocumentBuilder db = dbf.newDocumentBuilder();  
            Document doc = db.newDocument();  
            // Create the root element.  
            Element root = doc.createElement("movie");  
            doc.appendChild(root);  
            // Create name child element and add it to the root.  
            Element name = doc.createElement("name");  
            root.appendChild(name);  
            // Add a text element to the name element.  
            Text text = doc.createTextNode("Le Fabuleux Destin d'Amélie  
Poulain");  
            name.appendChild(text);  
            // Create language child element and add it to the root.  
            Element language = doc.createElement("language");  
            root.appendChild(language);  
            // Add a text element to the language element.  
            text = doc.createTextNode("français");  
            language.appendChild(text);  
            System.out.printf("Version = %s%n", doc.getXmlVersion());  
            System.out.printf("Encoding = %s%n", doc.getXmlEncoding());  
            System.out.printf("Standalone = %b%n%n", doc.getXmlStandalone());  
            NodeList nl = doc.getChildNodes();  
            for (int i = 0; i < nl.getLength(); i++) {  
                Node node = nl.item(i);  
                if (node.getNodeType() == Node.ELEMENT_NODE)  
                    dump((Element) node);  
            }  
        } catch (Exception e) {  
            e.printStackTrace(System.err);  
        }  
    }  
}
```

```

private static void dump(Element e) {
    System.out.printf("Element: %s, %s, %s, %s%n", e.getNodeName(),
                      e.getLocalName(), e.getPrefix(),
                      e.getNamespaceURI());
    NodeList nl = e.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (node instanceof Element)
            dump((Element) node);
        else if (node instanceof Text)
            System.out.printf("Text: %s%n", ((Text) node).getWholeText());
    }
}
}

```

`DOMDemo` creates Listing 16-2's movie document. It uses `Document`'s `createElement()` method to create the root `movie` element and `movie`'s `name` and `language` child elements. It also uses `Document`'s `Text` `createTextNode(String data)` method to create text nodes that are attached to the `name` and `language` nodes. Notice the calls to `Node`'s `appendChild()` method to append child nodes (such as `name`) to parent nodes (such as `movie`).

After creating this tree, `DOMDemo` outputs the tree's element nodes and other information. This output appears as follows:

```

Version = 1.0
Encoding = null
Standalone = false

Element: movie, null, null, null
Element: name, null, null, null
Text: Le Fabuleux Destin d'Amélie Poulain
Element: language, null, null, null
Text: français

```

The output is pretty much as expected, but there's one problem: the XML declaration's `encoding` attribute hasn't been set to `ISO-8859-1`. You cannot accomplish this task via the DOM API. Instead, you need to use the XSLT API. While exploring XSLT, you'll learn how to set the `encoding` attribute, and you'll also learn how to output this tree to an XML document file.

What Is JSON?

JSON (JavaScript Object Notation) is a data format used often today for exchanging data between servers and web pages, but it can also be used for server-to-server communication. The main differences between XML and JSON are as follows:

- In JSON, no tags are used; container elements are delimited by {} (objects with child elements) and [] (arrays).
- JSON elements cannot have attributes.
- JSON does not have a namespace concept.
- JSON does not have a validation mechanism.

A JSON document typically refers to a root object, denoted by a surrounding {} pair, and looks as follows:

```
{
  "name" : "Linda Green",
  "id" : 746,
  "rank" : 36.7,
  "active" : true,
  "children" : [
    747,
    748
  ],
  "address" : {
    "country" : "US",
    "state" : "Texas",
    "city" : "Austin"
  },
  "plan" : null
}
```

You can see that object elements follow a "name" : value syntax, while the contents of an array just are a comma-separated list. Numbers are allowed and are just entered literally. As Boolean values, we can use true and false, and null denotes “unspecified” or “unknown”.

JSON Processing in Java

JSON processing is not part of the JSE (standard edition), but you can add it easily by providing appropriate implementation libraries. In order to do so, add the following JARs to the CLASSPATH: javax.json-api-1.1.jar, javax.json-1.1.jar, javax.json.bind-api-1.0.jar, yasson-1.0.1.jar, cdi-api-2.0.jar, javax.el-api-3.0.0.jar, javax.interceptor-api-1.2.jar, and javax.inject-1.jar. You can download them from <https://mvnrepository.com>, or, if you have a Maven or Gradle project, directly use the following coordinates (group:artifact:version):

```
javax.json:javax.json-api:1.1
org.glassfish:javax.json:1.1
javax.json.bind:javax.json.bind-api:1.0
org.eclipse:yasson:1.0.1
```

Yasson is the reference implementation for JSONB (JSON Binding), which handles the conversion between Java objects and their JSON representation.

Generating JSON

With all the JSON processing and binding library JARs added, we can create JSON files. To, for example, create a JSON file using the builder pattern, write as follows:

```
Map<String, Object> properties = new HashMap<String, Object>() {{
    put(JsonGenerator.PRETTY_PRINTING, true);
}};
JsonGeneratorFactory jgf = Json.createGeneratorFactory(properties);
StringWriter sw = new StringWriter();
JsonGenerator jg = jgf.createGenerator(sw);

jg.writeStartObject()
    .write("name", "Linda Green")
    .write("id", 746)
    .write("rank", 36.7)
    .write("active", true)
    .writeStartArray("children")
        .write(747)
        .write(748)
```

```

    .writeEnd()
    .writeStartObject("address")
        .write("country", "US")
        .write("state", "Texas")
        .write("city", "Austin")
    .writeEnd()
    .writeNull("plan")
.writeEnd()
.close();

System.out.println(sw.toString());

```

Because of the `JsonGenerator.PRETTY_PRINTING` property, the output will use line breaks and indentations. Run the program, and in the console, you will see:

```

{
    "name": "Linda Green",
    "id": 746,
    "rank": 36.7,
    "active": true,
    "children": [
        747,
        748
    ],
    "address": {
        "country": "US",
        "state": "Texas",
        "city": "Austin"
    },
    "plan": null
}

```

In case we instead need to convert a Java object to a JSON presentation, we must use the JSONB technology. As an example, consider the following class:

```

public class Car {
    static public class Manufacturer {
        public String name;

```

```

    public String country;
}
public long id;
public int make;
public String name;
public BigDecimal price;
public double weightPds;
public Manufacturer manufacturer;
}

```

Of course you can add constructors to the class and its inner class, but for our purpose the public fields will do. Now to convert an instance of the class to its JSON representation, you can write:

```

Car c = new Car() {{
    id = 25;
    make = 2004;
    name = "Angelwing";
    price = new BigDecimal("35999.00");
    weightPds = 4500.0;
    manufacturer = new Manufacturer() {{
        name = "Frog Cars";
        country = "Argentine";
    }};
};

String carJson = JsonbBuilder.create(
    new JsonbConfig().withFormatting(true)).toJson(c);
System.out.println(carJson);

```

Because of the formatting configuration property added, the output will include line breaks and indentations. More precisely, the output of this code would read:

```
{
    "id": 25,
    "make": 2004,
```

```

"manufacturer":  

{  

    "country":  

        "Argentine",  

    "name":  

        "Frog Cars"  

},  

"name":  

    "Angelwing",  

"price":  

    35999.00,  

"weightPds":  

    4500.0
}

```

JSONB-specific annotations allow to exclude fields from serialization, or to specify formatting properties. They all begin with “@Jsonb”, and the API documentation at <http://json-b.net/docs/user-guide.html> tells you more details about it. In order, for example, to change the “name” property to “carName” and to remove the ID property from the output, you can write:

```

public class Car {  

    static public class Manufacturer {  

        public String name;  

        public String country;  

    }  

    @JsonbTransient public long id;  

    public int make;  

    @JsonbProperty("carName") public String name;  

    public BigDecimal price;  

    public double weightPds;  

    public Manufacturer manufacturer;  

}

```

Parsing JSON

To convert a JSON string to a Java object using JSON-P and JSONB, you have two options. You can either convert a JSON string to a generic Java object tree representation, or you can perform a conversion to an object tree with predefined Java classes.

Let us first create a generic object tree from JSON. We don't need JSONB for that and the procedure basically goes as follows:

```
String json = "{\n" +
    "    \"carName\": \n" +
    "    \"Angelwing\", \n" +
    "    \"make\": \n" +
    "    2004, \n" +
    "    \"manufacturer\": \n" +
    "    {\n" +
    "        \"country\": \n" +
    "        \"Argentine\", \n" +
    "        \"name\": \n" +
    "        \"Frog Cars\" \n" +
    "    }, \n" +
    "    \"price\": \n" +
    "    35999.00, \n" +
    "    \"weightPds\": \n" +
    "    4500.0\n" +
"}";
```

```
try {
    JsonReader rdr = Json.createReader(new StringReader(json));
    JsonObject obj = rdr.readObject();
    System.out.print(obj.getString("carName"));
} catch(Exception e) {
    e.printStackTrace(System.err);
}
```

Instead of a Reader, you can also use an `InputStream` as a source for the JSON data, for example, from a URL:

```
URL url = new URL("http://example.com/example");
JsonReader rdr = Json.createReader(url.openStream());
...

```

To generate an object tree with predefined Java classes, the procedure is to create a `Jsonb` instance and then use its `fromJson()` method:

```
public class Car {
    static public class Manufacturer {
        public String name;
        public String country;
    }
    @JsonbTransient public long id;
    public int make;
    @JsonbProperty("carName") public String name;
    public BigDecimal price;
    public double weightPds;
    public Manufacturer manufacturer;
}
...
String json = "{\n" +
    "    \"carName\": \n" +
    "    \"Angelwing\", \n" +
    "    \"make\": \n" +
    "    2004, \n" +
    "    \"manufacturer\": \n" +
    "    {\n" +
    "        \"country\": \n" +
    "        \"Argentine\", \n" +
    "        \"name\": \n" +
    "        \"Frog Cars\" \n" +
    "    }, \n" +
    "    \"price\": \n" +
    "    35999.00, \n" +

```

```
"      \"weightPds\": \n" +
"      4500.0\n" +
"}";
try {
    Jsonb jsonb = JsonbBuilder.create();
    Car car = jsonb.fromJson(new StringReader(json), Car.class);
    System.out.print(car.name);
} catch(Exception e) {
    e.printStackTrace(System.err);
}
```

EXERCISES

The following exercises are designed to test your understanding of Chapter 16's content:

1. Define XML.
2. What is the XML declaration?
3. Identify the XML declaration's three attributes. Which attribute is nonoptional?
4. True or false: An element always consists of a start tag followed by content followed by an end tag.
5. Following the XML declaration, an XML document is anchored in what kind of element?
6. What is mixed content?
7. What is a character reference? Identify the two kinds of character references.
8. What is a CDATA section? Why would you use it?
9. Define namespace.
10. What is a namespace prefix?
11. True or false: A tag's attributes don't need to be prefixed when those attributes belong to the element.
12. What is a comment? Where can a comment appear in an XML document?
13. Define processing instruction.

CHAPTER 16 WORKING WITH XML AND JSON DOCUMENTS

14. Identify the rules that an XML document must follow to be considered well formed.
15. What does it mean for an XML document to be valid?
16. A parser that performs validation compares an XML document to a grammar document. Identify the two commonly used grammar languages.
17. Define SAX.
18. How do you obtain a SAX 2-based parser?
19. What is the purpose of the `XMLReader` interface?
20. How do you tell a SAX parser to perform validation?
21. Define ignorable whitespace.
22. What is the purpose of the `org.xml.sax.helpers.DefaultHandler` class?
23. What is an entity? What is an entity resolver?
24. Define DOM.
25. How do you obtain a document builder?
26. How do you use a document builder to parse an XML document?
27. How do you use a document builder to create a new XML document?
28. When creating a new XML document, can you use the DOM API to specify the XML declaration's encoding attribute?
29. Define XPath.
30. Define XSLT.
31. Define JSON.
32. Build the JSON string equivalent (disregarding all attributes) of the XML from Listing 16-1.
33. Use a builder to create the same JSON as for the preceding exercise.
34. Take the JSON from the preceding exercise and replace each ingredient by an object { "name": "...", "quantity": Int-Value }.

35. Create a class Recipe (plus an inner class Ingredient) representing the recipe from the preceding exercise. Build an instance and use JSONB to create a JSON string from it.
 36. Parse the JSON string from the preceding exercise to build a new Recipe instance from it.
-

Summary

Applications often use XML documents to store and exchange data. Before you can understand these documents, you need to understand XML. This understanding requires knowledge of the XML declaration, elements and attributes, character references and CDATA sections, namespaces, and comments and processing instructions. It also involves learning what it means for a document to be well formed, and what it means for a document to be valid in terms of DTDs and XML Schema-based schemas.

You also need to learn how to process XML documents via the SAX and DOM APIs. SAX is used to parse documents via a callback paradigm, while DOM is used to parse documents into and create documents from node trees. XPath is used to search node trees in a more succinct manner than that offered by the DOM API, and XSLT (with help from XPath) is used to transform XML content to XML, HTML, or another format.

With JSON a more lightweight data exchange format compared to XML gets described. You learn how to generate JSON using a builder syntax from a Java object tree and how to generate object trees from a JSON representation.

This chapter largely wraps up the coverage of Java APIs that are supported by Android or stay in relation to Android application development. In chapter 17 we talk about the date and time API.

CHAPTER 17

Date and Time

Handling dates and times in a computer system is a considerably big challenge. We need to take care of different calendar systems, different time zones, daylight saving switches, and language- and country-dependent date and time formatting.

Java includes a sophisticated date and time handling, which is up to JDK version 7 in the majority of use cases boiled down to using classes `java.util.Date`, `java.util.Calendar`, `java.text.SimpleDateFormat`, and the function `System.currentTimeMillis()`.

Only starting with JDK version 8 and also available in Android (API > 25) a major rewriting of the date and time API took place, with the new date and time classes residing inside package `java.time`.

The old API is not deprecated though, and despite some shortcomings fixed by the new API, many developers still use the old classes, which is why we describe both of them.

As usual, we don't cover each and every class and feature, so the reader is asked to consult the official API documentation (obviously the one after JDK 7) to learn more about the details.

The Traditional Date and Time API

The original date and time API exists in Java since its very early times (version 1.0 and 1.1). You can use it to format and parse times and dates and to perform calendar operations.

About Dates and Calendars

Applications must properly handle dates, time zones, and calendars. A *date* is a recorded temporal moment, a *time zone* is a set of geographical regions that share a common number of hours relative to Greenwich Mean Time (GMT), and a *calendar* is a system of organizing the passage of time.

Note GMT identifies the standard geographical location from where all time is measured. UTC, which stands for Coordinated Universal Time, is often specified in place of GMT.

Java 1.0 introduced the `java.util.Date` class as its first attempt to describe calendars. However, `Date` was not amenable to internationalization because of its English-oriented nature and because of its inability to represent dates prior to midnight January 1, 1970 GMT, which is known as the *Unix epoch* (the date when Unix began to be used).

`Date` was eventually refactored to make it more useful by allowing `Date` instances to represent dates before the epoch as well as after the epoch, and by deprecating most of this class's constructors and methods; deprecated methods have been replaced by more appropriate API classes. Table 17-1 describes the more useful `Date` class methods and constructors.

Table 17-1. Date Constructors and Methods

Method	Description
<code>Date()</code>	Allocates a <code>Date</code> object and initializes it to the current time by calling <code>System.currentTimeMillis()</code> .
<code>Date(long date)</code>	Allocates a <code>Date</code> object and initializes it to the time represented by date milliseconds. A negative value indicates a time before the epoch, 0 indicates the epoch, and a positive value indicates a time after the epoch.
<code>boolean after(Date date)</code>	Returns true when this date occurs after date. This method throws <code>NullPointerException</code> when date is null.
<code>boolean before(Date date)</code>	Returns true when this date occurs before date. This method throws <code>NullPointerException</code> when date is null.

(continued)

Table 17-1. (continued)

Method	Description
Object clone()	Returns a copy of this object.
int compareTo(Date date)	Compares this date with date. Returns 0 when this date equals date, a negative value when this date comes before date, and a positive value when this date comes after date. This method throws NullPointerException when date is null.
boolean equals(Object obj)	Compares this date with the Date object represented by obj. Returns true if and only if obj isn't null and is a Date object that represents the same point in time (to the millisecond) as this date.
long getTime()	Returns the number of milliseconds that must elapse before the epoch (a negative value) or have elapsed since the epoch (a positive value).
int hashCode()	Returns this date's hash code. The result is the exclusive OR of the two halves of the long integer value returned by getTime(). That is, the hash code is the value of expression (int) (this.getTime() ^ (this.getTime() >>> 32)).
void setTime(long time)	Sets this date to represent the point in time specified by time milliseconds (a negative value refers to before the epoch; a positive value refers to after the epoch).
String toString()	Returns a String object containing this date's representation as dow mon dd hh:mm:ss zzz yyyy, where dow is the day of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat), mon is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec), dd is the two decimal-digit day of the month (01 through 31), hh is the two decimal-digit hour of the day (00 through 23), mm is the two decimal-digit minute within the hour (00 through 59), ss is the two decimal-digit second within the minute (00 through 61, where 60 and 61 account for leap seconds), zzz is the (possibly empty) time zone (and may reflect daylight saving time), and yyyy is the four decimal-digit year.

Listing 17-1 provides a small demonstration of the Date class.

Listing 17-1. Exploring the Date Class

```
import java.util.Date;

public class DateDemo {
    public static void main(String[] args) {
        Date now = new Date();
        System.out.println(now);
        Date later = new Date(now.getTime() + 86400);
        System.out.println(later);
        System.out.println(now.after(later));
        System.out.println(now.before(later));
    }
}
```

Listing 17-1's main() method creates a pair of Date objects (now and later) and outputs their dates, formatted according to Date's implicitly called `toString()` method. main() then demonstrates `after()` and `before()`, proving that now comes before later, which is one second in the future.

Compile Listing 17-1 as follows:

```
javac DateDemo.java
```

Run this application as follows:

```
java DateDemo
```

You should observe output similar to the following:

```
Sat Jan 04 13:25:20 CST 2020
Sat Jan 04 13:26:47 CST 2020
false
true
```

Date's `toString()` method reveals that a time zone is part of a date. Java provides the abstract `java.util.TimeZone` entry-point class for obtaining instances of `TimeZone` subclasses. This class declares a pair of class methods for obtaining these instances:

- `TimeZone getDefault()`
- `TimeZone getTimeZone(String ID)`

The latter method returns a `TimeZone` instance for the time zone whose `String` identifier (such as "CST") is passed to `ID`.

Note Some time zones take into account *daylight saving time*, the practice of temporarily advancing clocks so that afternoons have more daylight and mornings have less, for example, Central Daylight Time (CDT). Check out Wikipedia's "Daylight saving time" entry (http://en.wikipedia.org/wiki/Daylight_saving_time) to learn more about daylight saving time.

When you need to introduce a new time zone or modify an existing time zone, perhaps to deal with changes to a time zone's daylight saving time policy, you can work directly with `TimeZone`'s `java.util.SimpleTimeZone` concrete subclass. `SimpleTimeZone` describes a raw offset from GMT and provides rules for specifying the start and end of daylight saving time.

Java 1.1 introduced the Calendar API with its abstract `java.util.Calendar` entry-point class as a replacement for `Date`. `Calendar` is intended to represent any kind of calendar. However, time constraints meant that only the Gregorian calendar could be implemented (via the concrete `java.util.GregorianCalendar` subclass) for version 1.1.

`Calendar` declares the following class methods for obtaining calendars:

- `Calendar getInstance()`
- `Calendar getInstance(Locale locale)`
- `Calendar getInstance(TimeZone zone)`
- `Calendar getInstance(TimeZone zone, Locale locale)`

The first and third methods return calendars for the default locale; the second and fourth methods take the specified locale into account. Also, calendars returned by the first two methods are based on the current time in the default time zone; calendars returned by the last two methods are based on the current time in the specified time zone.

Calendar declares various constants, including YEAR, MONTH, DAY_OF_MONTH, DAY_OF_WEEK, LONG, and SHORT. These constants identify the year (four digits), month (0 represents January), current month day (1 through the month's last day), and current weekday (1 represents Sunday) calendar fields, and display styles (such as January vs. Jan).

The first four constants are used with Calendar's various set() methods to set calendar fields to specific values (e.g., set the year field to 2012). They're also used with Calendar's int get(int field) method to return field values, along with other field-oriented methods such as void clear(int field) (unset a field).

The latter two constants are used with Calendar's String getDisplayName(int field, int style, Locale locale) and Map<String, Integer> getDisplayNames(int field, int style, Locale locale) methods, which return short (e.g., Jan) or long (e.g., January) localized String representations of various field values.

Listing 17-2 shows how to use various Calendar constants and methods to output calendar pages according to the en_US and fr_FR locales.

Listing 17-2. Outputting Calendar Pages

```
import java.util.Calendar;
import java.util.Iterator;
import java.util.Locale;
import java.util.Map;
import java.util.Set;

public class CalendarDemo {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.err.println("usage: java CalendarDemo yyyy mm [f|F]");
            return;
        }
        try {
            int year = Integer.parseInt(args[0]);
            int month = Integer.parseInt(args[1]);
            Locale locale = Locale.US;
            if (args.length == 3 && args[2].equalsIgnoreCase("f"))
                locale = Locale.FRANCE;
            showPage(year, month, locale);
        }
    }
}
```

```
        } catch (NumberFormatException nfe) {
            System.err.print(nfe);
        }
    }

private static void showPage(int year, int month, Locale locale) {
    if (month < 1 || month > 12)
        throw new IllegalArgumentException("month [" + month + "] out of " +
                                           "range [1, 12]");
    Calendar cal = Calendar.getInstance(locale);
    cal.set(Calendar.YEAR, year);
    cal.set(Calendar.MONTH, --month);
    cal.set(Calendar.DAY_OF_MONTH, 1);
    displayMonthAndYear(cal, locale);
    displayWeekdayNames(cal, locale);
    int daysInMonth = cal.getActualMaximum(Calendar.DAY_OF_MONTH);
    int firstRowGap = cal.get(Calendar.DAY_OF_WEEK)-1; // 0 = Sunday
    for (int i = 0; i < firstRowGap; i++)
        System.out.print("    ");
    for (int i = 1; i <= daysInMonth; i++) {
        if (i < 10)
            System.out.print(' ');
        System.out.print(i);
        if ((firstRowGap + i) % 7 == 0)
            System.out.println();
        else
            System.out.print(' ');
    }
    System.out.println();
}

private static void displayMonthAndYear(Calendar cal, Locale locale) {
    System.out.println(cal.getDisplayName(Calendar.MONTH, Calendar.LONG,
                                         locale) + " " +
                       cal.get(Calendar.YEAR));
}
```

```

private static void displayWeekdayNames(Calendar cal, Locale locale) {
    Map<String, Integer> weekdayNamesMap;
    weekdayNamesMap = cal.getDisplayNames(Calendar.DAY_OF_WEEK,
                                           Calendar.SHORT, locale);
    String[] names = new String[weekdayNamesMap.size()];
    int[] indexes = new int[weekdayNamesMap.size()];
    Set<Map.Entry<String, Integer>> weekdayNamesEntries;
    weekdayNamesEntries = weekdayNamesMap.entrySet();
    Iterator<Map.Entry<String, Integer>> iter;
    iter = weekdayNamesEntries.iterator();
    while (iter.hasNext()) {
        Map.Entry<String, Integer> entry = iter.next();
        names[entry.getValue() - 1] = entry.getKey();
        indexes[entry.getValue() - 1] = entry.getValue();
    }
    for (int i = 0; i < names.length; i++)
        for (int j = i; j < names.length; j++)
            if (indexes[j] == i + 1){
                System.out.print(names[j].substring(0, 2) + " ");
                continue;
            }
    System.out.println();
}
}

```

Listing 17-2 is pretty straightforward with the exception of `displayWeekdayNames()`. This method calls `Calendar's getDisplayNames()` method to return a map of localized weekday names. Instead of returning a map where the keys are `java.lang.Integers` and the values are localized `Strings`, this map's keys are the localized `Strings`.

This would be fine if the keys were ordered (as in Sunday first and Saturday last, or lundi first and dimanche last). However, they're not ordered. To output these names in order, it's necessary to obtain a set of map entries, iterate over these entries and populate parallel arrays, and then iterate over these arrays to output the weekday names.

Note A French calendar begins the week on lundi (Monday) and ends it on dimanche (Sunday). However, Calendar doesn't take this ordering into account.

Compile Listing 17-2 as follows:

```
javac CalendarDemo.java
```

Run this application with an appropriate year, month, and locale:

```
java CalendarDemo 2014 01
```

When we specify the previous command line for the en_CA locale, we observe the following calendar page:

January 2014

Su	Mo	Tu	We	Th	Fr	Sa
1	2	3	4			
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

If you would like to see this page in the fr_FR locale, specify either of the following command lines:

```
java CalendarDemo 2014 01 f
```

```
java CalendarDemo 2014 01 F
```

You should then observe the following calendar page:

janvier 2014

di	lu	ma	me	je	ve	sa
1	2	3	4			
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Note `Calendar` declares a `Date getTime()` method that returns a calendar's time representation as a `Date` instance. `Calendar` also declares a `void setTime(Date date)` method that sets a calendar's time representation to the specified date.

Date and Time Formatters

The abstract `java.text.DateFormat` entry-point class (a `Format` subclass) provides access to formatters that format `Date` instances as dates or time values (and also to parse such values). This class declares the following class methods:

- `DateFormat getDateInstance()`
- `DateFormat getDateInstance(int style)`
- `DateFormat getDateInstance(int style, Locale locale)`
- `DateFormat getDateTimeInstance()`
- `DateFormat getDateTimeInstance(int dateStyle, int timeStyle)`
- `DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale locale)`
- `DateFormat getInstance()`
- `DateFormat getTimeInstance()`
- `DateFormat getTimeInstance(int style)`
- `DateFormat getTimeInstance(int style, Locale locale)`

The `getDateInstance()` class methods' formatters generate only date information, the `getTimeInstance()` class methods' formatters generate only time information, and the `getDateTimeInstance()` class methods' formatters generate date and time information.

The `dateStyle` and `timeStyle` fields determine how that information will be presented according to the following `DateFormat` constants:

- `SHORT` is completely numeric, such as `12.13.52` or `3:30pm`.
- `MEDIUM` is longer, such as `Jan 12, 1952`.

- LONG is longer still, such as January 12, 1952 or 3:30:32pm.
- FULL is pretty completely specified, such as Tuesday, April 12, 1952 AD or 3:30:42pm PST.

Listing 17-3 shows you how to format a Date instance that represents the Unix epoch according to the local time zone and the UTC time zone.

Listing 17-3. Formatting the Unix Epoch

```
import java.text.DateFormat;
import java.util.Date;
import java.util.Locale;
import java.util.TimeZone;

public class DateFormatDemo {
    public static void main(String[] args) {
        Date d = new Date(0); // Unix epoch
        System.out.println(d);
        DateFormat df = DateFormat.getDateInstance(DateFormat.LONG,
                                                    DateFormat.LONG,
                                                    Locale.US);
        System.out.println("Default format: " + df.format(d));
        df.setTimeZone(TimeZone.getTimeZone("UTC"));
        System.out.println("Taking UTC into account: " + df.format(d));
    }
}
```

Compile Listing 17-3 as follows:

```
javac DateFormatDemo.java
```

Run this application as follows:

```
java DateFormatDemo
```

We observe the following output for the CST time zone:

```
Wed Dec 31 18:00:00 CST 1969
Default format: December 31, 1969 6:00:00 PM CST
Taking UTC into account: January 1, 1970 12:00:00 AM UTC
```

The Unix epoch, which is represented by passing 0 to the Date(long) constructor, is defined as January 1, 1970 00:00:00 UTC, but the first output line doesn't indicate this fact. Instead, it shows the epoch in our CST time zone, which is six hours away from GMT/UTC. To show the epoch correctly, we need to obtain the UTC time zone, which we accomplish by passing "UTC" to TimeZone's getTimeZone(String) class method and install this time zone instance into the date formatter with the help of DateFormat's void setTimeZone(TimeZone zone) method.

Note When you need to create customized date formatters, you'll find yourself working with DateFormat's java.text.SimpleDateFormat subclass and this subclass's java.text.DateFormatSymbols companion class. The "Customizing Formats" section (<http://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html>) and the "Changing Date Format Symbols" section (<http://docs.oracle.com/javase/tutorial/i18n/format/dateFormatSymbols.html>) in *The Java Tutorials* introduce you to these classes.

Parsing

The Format class has a dual personality in that it also declares a pair of parseObject() methods for parsing strings back into objects. Furthermore, it associates with a java.text.ParseException class whose instances are thrown when errors occur during parsing and a java.text.ParsePosition class that keeps track of the current parsing position and error position indexes.

Note ParsePosition declares int getIndex() and void setIndex(int index) methods for getting and setting the current parsing position, and it declares int getErrorIndex() and void setErrorIndex(int index) methods for getting and setting the current error position.

Format declares the following parseObject() methods:

- Object parseObject(String source) parses source from the beginning and returns a corresponding object. Not all of source's text may be parsed. This method throws ParseException when the beginning of this text cannot be parsed.
- Object parseObject(String source, ParsePosition pos) parses source starting at the current parsing position index stored in pos and returns a corresponding object. When parsing succeeds, pos's current parsing position index is updated to the index after the last character used (parsing doesn't necessarily use all characters up to the end of the string), and the parsed object is returned. The updated pos can be used to indicate the starting point for the next call to this method. When an error occurs, pos's current parsing position index isn't changed. However, its error position index is set to the index of the character where the error occurred and null is returned. This method throws NullPointerException when null is passed to pos.

parseObject(String) invokes the abstract parseObject(String, ParsePosition) method as if by calling parseObject(source, new ParsePosition(0)).

Format subclasses such as DateFormat override parseObject(String, ParsePosition) to invoke one of their own parse() methods. For example, MessageFormat overrides parseObject(String, ParsePosition) and calls this method when necessary.

Although you should refrain from using specialty subclasses so that your application can adapt to the widest possible audience, you might find occasions to use such subclasses. For example, you might want to work directly with SimpleDateFormat to parse legacy date/time strings that were stored in a database according to a specific format. Listing 17-4 demonstrates how you might accomplish this task with help from SimpleDateFormat.

Note `SimpleDateFormat` is not thread-safe. You cannot share a single instance of this class among different threads.

Listing 17-4. Parsing a Date/Time Argument That Must Be Formatted According to Specific Date Format

```
import java.text.ParseException;
import java.text.SimpleDateFormat;

import java.util.Date;

public class ParseDemo {
    public static void main(String[] args) throws ParseException {
        if (args.length != 1) {
            System.err.println("usage: java ParseDemo yyyy-MM-dd HH:mm:ss z");
            return;
        }
        SimpleDateFormat sdf;
        sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss z");
        System.out.println(sdf.parse(args[0]));
    }
}
```

Listing 17-4 first verifies that a single command-line argument was passed and then instantiates `SimpleDateFormat` with a pattern string that identifies the pattern to follow for parsing. (The complete details on pattern string syntax are available in `SimpleDateFormat`'s Java documentation.)

Compile Listing 17-4 as follows:

```
javac ParseDemo.java
```

Run this application as follows:

```
java ParseDemo "2014-01-04 11:21:00 CST"
```

On our platform, we observe the following output:

```
Sat Jan 04 11:21:00 CST 2014
```

The New Date and Time API

Starting with Android API level 26 (Android 8.0), a couple of new date-/time-related interfaces and classes are available. You can continue using the old API described in the previous section, but the new API contains some improvements we are going to outline in this section.

The new interfaces and classes live in package `java.time`. For the rest of this section, we usually omit the corresponding imports or use the `*` wildcard.

Local Dates and Times

Local dates and times get described from the context of the observer and basically use the following classes from the `java.time` package:

- `LocalDate`

This class corresponds to a date representation of the format `yyyy-MM-dd` (e.g., `2018-11-27`) and disregards the time of day.

- `LocalTime`

This class corresponds to a time representation of the format `HH:mm:ss` (e.g., `21:27:55`) and disregards the date.

- `LocalDateTime`

A combination of `LocalDate` and `LocalTime`, possibly represented by `yyyy-MM-ddTHH:mm:ss` (the “T” is a literal), where the format designators “`yyyy`”, “`HH`”, and so on are described in the API documentation of `java.time.DateTimeFormatter`.

All three of them include factory methods to generate object instances. This includes taking the current date and time:

```
import java.time.*;

// current day in the default time zone
LocalDate ld1 = LocalDate.now();

// "Now" corresponds to different days in different
// time zones. The following allows to specify a
```

```
// different time zone
ZoneId z2 = ZoneId.of("UTC+01");
LocalDate ld2 = LocalDate.now(z2);

LocalDate ld3 = LocalDate.of(2018, Month.MARCH, 27);
LocalDate ld4 = LocalDate.of(2018, 3, 27); // the same

LocalTime lt1 = LocalTime.now();
LocalTime lt2 = LocalTime.now(z2); // different time zone
LocalTime lt3 = LocalTime.of(23, 27, 55); // 23:27:55

LocalDateTime ldt1 = LocalDateTime.now();
LocalDateTime ldt2 = LocalDateTime.now(z2);
LocalDateTime ldt3 = LocalDateTime.of(
    2018, Month.APRIL, 23,
    23, 44, 12);
// <- 2018-04-23T23:44:12
```

Note that despite the possibility to add a time zone specification to further specify to which time “now” corresponds, this information is by no means somehow stored in the date/time object. Local dates and times are by definition time zone agnostic.

We can parse strings to obtain instances of `LocalDate`, `LocalTime`, and `LocalDateTime`:

```
import java.time.*;
import java.time.format.*;

// Parse ISO-8601
LocalDate ld1 = LocalDate.parse("2019-02-13");

// Parse other formats. For the format specification,
// see API documentation of class DateTimeFormatter.
DateTimeFormatter formatter1 =
    DateTimeFormatter.ofPattern("yyyy MM dd");
LocalDate ld2 = LocalDate.parse("2019 02 13", formatter1);

LocalTime lt1 = LocalTime.parse("21:17:23");
LocalTime lt2 = LocalTime.parse("21:17:23.3734");
```

```

DateTimeFormatter formatter2 =
    DateTimeFormatter.ofPattern("HH|mm|ss");
LocalTime lt3 = LocalTime.parse("21|17|23", formatter2);

val ldt1 = LocalDateTime.parse("2019-02-13T21:17:23")
val ldt2 = LocalDateTime.parse(
    "2019-02-13T21:17:23.3734")

DateTimeFormatter formatter3 = DateTimeFormatter.ofPattern(
    "yyyy.MM.dd.HH.mm.ss");
LocalTime ldt3 = LocalTime.parse("2019.04.23.17.45.23",
    formatter3);

```

We can tailor our own string representations of `LocalDate`, `LocalTime`, and `LocalDateTime` instances:

```

import java.time.*;
import java.time.format.*;

String s1 = LocalDate.now().format(
    DateTimeFormatter.ofPattern("yyyy|MM|dd"));
System.out.println("s1 = " + s1); // -> 2019|01|14

String s2 = LocalDate.now().format(
    DateTimeFormatter.ISO_LOCAL_DATE);
System.out.println("s2 = " + s2); // -> 2019-01-14

String s3 = LocalTime.now().format(
    DateTimeFormatter.ofPattern("HH mm ss"));
System.out.println("s3 = " + s3); // -> 14 46 20

String s4 = LocalTime.now().format(
    DateTimeFormatter.ISO_LOCAL_TIME);
System.out.println("s4 = " + s4); // 14:46:20.503

String s5 = LocalDateTime.now().format(
    DateTimeFormatter.ofPattern(
        "yyyy MM dd - HH mm ss"));
System.out.println("s5 = " + s5); // -> 2019 01 14 - 14 46 20

```

```
String s6 = LocalDateTime.now().format(
    DateTimeFormatter.ISO_LOCAL_DATE_TIME);
System.out.println("s6 = " + s6); // -> 2019-01-14T14:46:20.505
```

You can perform time arithmetic with `LocalDate`, `LocalTime`, and `LocalDateTime` instances:

```
import java.time.*;
import java.time.temporal.*;

LocalDate ld = LocalDate.now();
LocalTime lt = LocalTime.now();
LocalDateTime ldt = LocalDateTime.now();

LocalDate ld2 = ld.minusDays(7L);
LocalDate ld3 = ld.plusWeeks(2L);
LocalDate ld4 = ld.with(ChronoField.MONTH_OF_YEAR, 11L);

LocalTime lt2 = lt.plus(Duration.of(2L, ChronoUnit.SECONDS));
LocalTime lt3 = lt.plusSeconds(2L); // same

LocalDateTime ldt2 = ldt.plusWeeks(2L).minusHours(2L);
```

From `LocalDateTime` we can calculate the number of seconds which have passed since 1970-01-01:00:00:00 UTC, similar to the `System.currentTimeMillis()` function from the old API:

```
import java.time.*;

LocalDateTime ldt = ...
long secs = ldt.toEpochSecond(ZoneOffset.of("+01:00"));
```

Note that to get the epoch seconds a better solution is to take a `ZonedDateTime`. We'll talk about zoned dates and times in the following texts.

Instants

An instant is an instantaneous point on the timeline. Use this for cases where you need unique absolute timestamps, for example, to register events in databases and alike. The precise definition is somewhat involved—for an introduction read the API documentation of `java.time.Instant`.

You can get an `Instant` by, for example, querying the system clock, or by specifying the elapsed time since 1970-01-01T00:00:00Z, or by parsing a time string, or from other date and time objects:

```
import java.time.*;

Instant inz1 = Instant.now(); // default time zone

// Specify time zone
Instant inz2 = Instant.now(Clock.system(
    ZoneId.of("America/Buenos_Aires")));

Long secondsSince1970 = 1_000_000_000L;
Long nanoAdjustment = 300_000_000; // 300ms
Instant inz3 = Instant.ofEpochSecond(
    secondsSince1970, nanoAdjustment);

// "Z" is UTC ("Zulu" time)
Instant inz4 = Instant.parse("2018-01-23T23:33:14.513Z");

// Uniform converter, for the ZonedDateTime class
// See the following
Instant inz5 = Instant.from(
    ZonedDateTime.parse(
        "2019-02-13T21:17:23+01:00[Europe/Paris]"));
```

Offset Dates and Times

Offset dates and times are like `Instants` with an additional time offset from UTC/Greenwich. For such offset date and times, we have two classes, `OffsetTime` and `OffsetDateTime`, for which you can get instances as follows:

```
import java.time.*;
import java.time.format.DateTimeFormatter;

// Get now -----
// System clock, default time zone
OffsetTime ot1 = OffsetTime.now();
OffsetDateTime odt1 = OffsetDateTime.now();
```

CHAPTER 17 DATE AND TIME

```
// Use a different clock.  
Clock clock = ...  
OffsetTime ot2 = OffsetTime.now(clock);  
OffsetDateTime odt2 = OffsetDateTime.now(clock);  
  
// Use a different time zone  
OffsetTime ot3 = OffsetTime.now(  
    ZoneId.of("America/Buenos_Aires"));  
OffsetDateTime odt3 = OffsetDateTime.now(  
    ZoneId.of("America/Buenos_Aires"));  
  
// From time details -----  
  
OffsetTime ot4 = OffsetTime.of(23, 17, 3, 500_000_000,  
    ZoneOffset.of("-02:00"));  
OffsetDateTime odt4 = OffsetDateTime.of(  
    1985, 4, 23, // 1985-04-23  
    23, 17, 3, 500_000_000, //23:17:03.5  
    ZoneOffset.of("+02:00"));  
  
// Parsed -----  
  
OffsetTime ot5 = OffsetTime.parse("16:15:30+01:00");  
OffsetDateTime odt5 = OffsetDateTime.parse(  
    "2007-12-03T17:15:30-08:00");  
OffsetTime ot6 = OffsetTime.parse("16 15 +00:00",  
    DateTimeFormatter.ofPattern("HH mm XXX"));  
OffsetDateTime odt6 = OffsetDateTime.parse(  
    "20181115 - 231644 +02:00",  
    DateTimeFormatter.ofPattern(  
        "yyyyMMdd - HHmmss XXX"));  
  
// From other objects -----  
  
LocalTime lt = LocalTime.parse("16:14:27.235");  
LocalDate ld = LocalDate.parse("2018-05-24");  
Instant inz = Instant.parse("2018-01-23T23:33:14.513Z");  
OffsetTime ot7 = OffsetTime.of(lt,  
    ZoneOffset.of("+02:00"));
```

```

OffsetDateTime odt7 = OffsetDateTime.of(ld, lt,
    ZoneOffset.of("+02:00"));
OffsetTime ot8 = OffsetTime.ofInstant(inz,
    ZoneId.of("America/Buenos_Aires"));
OffsetDateTime odt8 = OffsetDateTime.ofInstant(inz,
    ZoneId.of("America/Buenos_Aires"));

ZonedDateTime zdt = ZonedDateTime.of( // see below
    2018, 2, 27,           // 2018-02-27
    23, 27, 33, 0,         // 23:27:33.0
    ZoneId.of("Pacific/Tahiti")));
OffsetDateTime odt9 = zdt.toOffsetDateTime();

// Uniform converter
OffsetTime ot10 = OffsetTime.from(zdt);
OffsetDateTime odt10 = OffsetDateTime.from(zdt);

```

With offset date and times, you can do arithmetic and formatting basically the same way as was possible for local dates and times. In addition, we have

```

import java.time.*;

OffsetTime ot = OffsetTime.parse("16:15:30+01:00");
LocalTime lt = ot.toLocalTime();

OffsetDateTime odt = OffsetDateTime.parse(
    "2007-12-03T17:15:30-08:00");
LocalDateTime ldt = odt.toLocalDateTime();
LocalTime lt2 = odt.toLocalTime();
LocalDate ld2 = odt.toLocalDate();
OffsetTime ot2 = odt.toOffsetTime();

ZonedDateTime zdt = odt.toZonedDateTime();
// See the following for class ZonedDateTime

```

for conversion operations.

Zoned Dates and Times

Local dates and times are great if we don't have to care about user location. If we have different entities, users, computers, or whatever all over the world entering dates and times, we need to add the time zone information. This is what the class `ZonedDateTime` is for.

Note that this is not the same as a date and time with a fixed time offset information, as is the case for `OffsetDateTime`. A time zone includes things like daylight saving time, which need to be taken into account.

Similar to `LocalDateTime`, the `ZonedDateTime` has factory methods for getting "now":

```
import java.time.*;

// Get "now" using the system clock and the default
// time zone from your operating system.
ZonedDateTime zdt1 = ZonedDateTime.now();

// Get "now" using a time zone. To list all available
// predefined zone IDs, try
//     for(ZoneId zi : ZoneId.getAvailableZoneIds())
//         System.out.println( zi );
ZoneId z2 = ZoneId.of("UTC+01");
ZonedDateTime zdt2 = ZonedDateTime.now(z2);

// Get "now" using an instance of Clock.
Clock clock3 = Clock.systemUTC();
ZonedDateTime zdt3 = ZonedDateTime.now(clock3);
```

And we can get a `ZonedDateTime` using detailed time information, and parse a string representation of a timestamp to get a `ZonedDateTime`:

```
import java.time.*;

val z4 = ZoneId.of("Pacific/Tahiti");
val zdt4 = ZonedDateTime.of(
    2018, 2, 27,           // 2018-02-27
    23, 27, 33, 0,         // 23:27:33.0
    z4);
// The 7th par is nanoseconds, so for
// 23:27:33.5 you have to enter
// 500_000_000 here
```

```

LocalDate localDate = LocalDate.parse("2018-02-27");
LocalTime localTime = LocalTime.parse("23:44:55");
ZonedDateTime zdt5 = ZonedDateTime.of(localDate, localTime,
    ZoneId.of("America/Buenos_Aires")));

LocalDateTime ldt = LocalDateTime.parse("2018-02-27T23:44:55.3");
ZonedDateTime zdt6 = ZonedDateTime.of(ldt,
    ZoneId.of("America/Buenos_Aires")));

Instant inz = Instant.parse("2018-01-23T23:33:14.513Z");
ZonedDateTime zdt7 = ZonedDateTime.ofInstant(inz,
    ZoneId.of("America/Buenos_Aires")));

ZonedDateTime zdt8 = ZonedDateTime.parse(
    "2018-01-23T23:33:14Z[America/Buenos_Aires]");

```

A `ZonedDateTime` allows for operations like `plusWeeks(long weeks)` and `minusDays(long days)` to build a new instance with the time given added or subtracted. This works for any of years, months, weeks, days, hours, minutes, seconds, and nanos.

There are various getter functions for the different time fractions: `getYear()`, `getMonth()`, `getMonthValue()`, `getDayOfMonth()`, `getHour()`, `getMinute()`, `getSecond()`, `getNano()`, plus a few others. To get the time zone, write `getZone()`.

To parse a date/time string and to convert a `ZonedDateTime` to a string, write:

```

import java.time.*;
import java.time.format.DateTimeFormatter;

ZonedDateTime zdt1 = ZonedDateTime.parse(
    "2007-12-03T10:15:30+01:00[Europe/Paris]");

DateTimeFormatter formatter = DateTimeFormatter.ofPattern(
    "HH:mm:ss.SSS");
// See DateTimeFormatter API docs for more options.
String str = zdt1.format(formatter);

```

The connection between a `ZonedDateTime` and a `LocalDateTime` happens via

```
import java.time.*;

LocalDateTime ldt = LocalDateTime.parse("2018-02-27T23:44:55.3");
ZonedDateTime zdt = ZonedDateTime.of(ldt,
        ZoneId.of("America/Buenos_Aires"));

LocalDateTime ldt2 = zdt.toLocalTime();
```

Duration and Periods

A duration is the physical time span between two instances. A period is similar, but only handles years, months, and days and takes the calendar system into account. For duration and periods, there are the special `Duration` and `Period` classes for handling them:

```
import java.time.*;
import java.time.temporal.ChronoUnit;

LocalDateTime ldt1 = LocalDateTime.parse("2018-01-23T17:23:00");
LocalDateTime ldt2 = LocalDateTime.parse("2018-01-24T16:13:10");
LocalDateTime ldt3 = LocalDateTime.parse("2020-01-24T16:13:10");

// Getting a duration: -----
Duration d1 = Duration.between(ldt1, ldt2);
// Note: this works also for Instant and ZonedDateTime
// objects.

Duration d2 = Duration.of(27L, ChronoUnit.HOURS); // 27 hours

Duration d3 = Duration.ZERO.
        plusDays(3L).
        plusHours(4L).
        minusMinutes(78L);

Duration d4 = Duration.parse("P2DT3H4M");
// <- 2 days, 3 hrs, 4 minutes
// For more specifiers, see the API documentation
```

```
// of Duration.parse()

// Getting a period: ----

LocalDate ld1 = LocalDate.parse("2018-04-23");
LocalDate ld2 = LocalDate.parse("2018-08-16");

Period p1 = Period.between(ld1, ld2);
// Note, end date not inclusive

Period p2 = Period.of(2, 3, -1);
// <- 2 years + 3 months - 1 day

Period p3 = Period.parse("P1Y2M-3D");
// <- 1 year + 2 months - 3 days
// For more specifiers, see the API documentation
// of Period.parse()
```

You can perform arithmetic on instances of the Duration or Period class:

```
import java.time.*;

// Duration operations: ----

Duration d = Duration.parse("P2DT3H4M");
// <- 2 days, 3 hrs, 4 minutes

Duration d2 = d.plusDays(3L);
// also: .minusDays(33L)
// or    .plusHours(2L)   or .minusHours(1L)
// or    .plusMinutes(77L) or .minusMinutes(7L)
// or    .plusSeconds(23L) or .minusSeconds(5L)
// or    .plusMillis(11L)  or .minusMillis(55L)
// or    .plusNanos(1000L) or .minusNanos(5_000_000L)

Duration d3 = d.abs();           // make positive
Duration d4 = d.negated();      // swap sign
Duration d5 = d.multipliedBy(3L); // three times a long
Duration d6 = d.dividedBy(2L);   // half as long
```

```
// Period operations: -----
Period p = Period.of(2, 3, -1);
// <- 2 years + 3 months - 1 day

Period p2 = p.normalized();
// <- possibly adjusts the year to make the month lie
// inside [-11;+11]

Period p3 = p.negated();

Period p4 = p.minusYears(11L);
// also: .plusYears(3L)
// or     .minusMonths(4L)  or  .plusMonths(2L)
// or     .minusDays(40L)   or  .plusDays(5L)

Period p5 = p.multipliedBy(5); // 5 times as long
```

You can use duration and periods to add or subtract time amounts to and from LocalDate, LocalTime, LocalDateTime, ZonedDateTime, and Instant objects.

```
import java.time.*;

Duration d = Duration.parse("P2DT3H4M");

Period p = Period.of(2, 3, -1);
// <- 2 years + 3 months - 1 day

LocalDate ld = LocalDate.parse("2018-04-23");
LocalTime lt = LocalTime.parse("17:13:12");
LocalDateTime ldt = LocalDateTime.of(ld, lt);
ZonedDateTime zdt = ZonedDateTime.parse(
    "2007-12-03T10:15:30+01:00[Europe/Paris]");
Instant inz = Instant.parse("2018-01-23T23:33:14.513Z");

// ---- Using a LocalDate
LocalDate ld2 = ld.plus(p); // or .minus(p)
// LocalDate ld3 = ld.plus(d); // -> exception
// LocalDate ld4 = ld.minus(d); // -> exception
```

```

// ---- Using a LocalTime
LocalTime lt2 = lt.plus(d); // or .minus(d)
// LocalTime lt3 = lt.minus(p); // -> exception
// LocalTime lt4 = lt.plus(p); // -> exception

// ---- Using a LocalDateTime
LocalDateTime ldt2 = ldt.plus(d); // or .minus(d)
LocalDateTime ldt3 = ldt.plus(p); // or .minus(p)

// ---- Using a ZonedDateTime
ZonedDateTime zdt2 = zdt.plus(d); // or .minus(d)
ZonedDateTime zdt3 = zdt.plus(p); // or .minus(p)

// ---- Using an Instant
Instant inz2 = inz.plus(d); // or .minus(d)
// Instant inz3 = inz.minus(p); // -> exception
// Instant inz4 = inz.plus(p); // -> exception

```

Note that some of the operations are not allowed and lead to an exception. Those are commented out in the previous listing. The reasons for the exceptions are possible precision losses or mismatches in the time concepts. See the API documentation for details.

Clock

A Clock sits in the depths of the date and time API. For many, if not most applications, you can work well with local dates and times, offset and zoned dates and times, and instants. Only for testing and special needs it might be necessary to tweak the clock usage for getting “now”:

```

import java.time.*;

Clock clock = ...
LocalDateTime ldt = LocalDateTime.now(clock);
ZonedDateTime zdt = ZonedDateTime.now(clock);
Instant inz = Instant.now(clock);

```

Apart from overwriting the abstract `Clock` class, `Clock` itself provides a couple of functions to tweak clocking usage. These two are particularly interesting:

- `Clock.fixed(Instant fixedInstant, ZoneId zone)`: A clock which always returns the same instant
- `Clock.offset(Clock baseClock, Duration offsetDuration)`: Returns a new clock derived from the base clock with the specified duration added

If you however overwrite the clock, you must of course implement at least the abstract functions from the `Clock` base class. As an example of a clock which always returns the same instant and doesn't care about zones:

```
import java.time.*;

Clock myClock = new Clock() {
    public Clock withZone(ZoneId zone) {
        // Supposed to return a copy of this clock
        // with a different time zone
        return this;
    }

    public ZoneId getZone() {
        // Supposed to return the zone ID
        return ZoneId.of("Z");
    }

    public Instant instant() {
        // This is the engine of the clock. It must
        // provide an Instant
        return Instant.parse("2018-01-23T23:33:14Z");
    }
};

... use myClock
```

EXERCISES

The following exercises are designed to test your understanding of Chapter 17's content:

1. Define date, time zone, and calendar.
2. How does the Date class represent a date?
3. How do you obtain a calendar for the default locale that uses a specific time zone?
4. True or false: Calendar declares a Date getDate() method that returns a calendar's time representation as a Date instance.
5. How would you obtain a date formatter to format the time portion of a date in a particular style for the default locale?
6. LocalDate contains detailed time of day information (hour, minute, second). Right or wrong?
7. A LocalDateTime object contains information of the system's time zone. Right or wrong?
8. Write the current system time to a string like in “2010-04-01T23:57:12.835”. Use the old date and time API.
9. Do the same as described in the previous exercise using the new date and time API.
10. Use Clock to determine the number of milliseconds since 1970-01-01 00:00:00 UTC.
11. Parse the string “2020-01-10 22:16:45” to create a ZonedDateTime object using the current system's zone.
12. Build a LocalDateTime object from the previous exercise's result.
13. Parse the string “2020-01-10 22:16:45” to create a ZonedDateTime object using UTC.
14. Calculate the duration between “2020-01-10 22:16:45” and “2020-01-12 13:10:45” in seconds.

15. Create a clock `ClockTwiceAsFast` inheriting from `Clock` with a constructor fetching the time from the UTC system clock. After that, the clock should run twice as fast. Disregard zone information. Use

```
import java.time.*;
Clock myClock = new ClockTwiceAsFast();
System.out.println(
    LocalDateTime.now(myClock).format(
        DateTimeFormatter.ISO_LOCAL_DATE_TIME));
Thread.sleep(1000L);
System.out.println(
    LocalDateTime.now(myClock).format(
        DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

to prove that it is running the intended way.

Summary

Applications must properly handle dates, time zones, and calendars. A date is a recorded temporal moment, a time zone is a set of geographical regions that share a common number of hours relative to Greenwich Mean Time (GMT), and a calendar is a system of organizing the passage of time.

The old date and time API dates back to Java 1.0 and 1.1. It focuses around the `java.util.Date` and `java.util.Calendar` classes for providing time and date objects and calendar operations. For date and time formatting and parsing, mostly `java.text.SimpleDateFormat` gets used.

While the old API is not deprecated and still can be used and is usable, the new API available since Java 8 (and Android API 26) introduces several new concepts: a localized date and time representation, a representation with time zone included, and a representation with offset included. Furthermore, an `Instant` represents a point in time, and a `Duration` and a `Period` class stand for time ranges. The `Clock` class encapsulates a time-advancing mechanism and a subclass of it may be used to tweak time propagation for testing purposes.

APPENDIX A

Solutions to Exercises

Each of Chapters 1 through 17 closes with an “Exercises” section that tests your understanding of the chapter’s material. Solutions to these exercises are presented in this appendix.

Chapter 1: Getting Started with Java

1. Java is a language and a platform. The language is partly patterned after the C and C++ languages to shorten the learning curve for C/C++ developers. The platform consists of a virtual machine and an associated execution environment.
2. A virtual machine is a software-based processor that presents its own instruction set.
3. The purpose of the Java compiler is to translate source code into instructions (and associated data) that are executed by the virtual machine.
4. The answer is true: a class file’s instructions are commonly referred to as bytecode.
5. When the JVM’s interpreter learns that a sequence of bytecode instructions is being executed repeatedly, it informs the JVM’s just-in-time (JIT) compiler to compile these instructions into native code.
6. The Java platform promotes portability by providing an abstraction over the underlying platform. As a result, the same bytecode runs unchanged on Windows-based, Linux-based, Mac OS X-based, and other platforms.

APPENDIX A SOLUTIONS TO EXERCISES

7. The Java platform promotes security by doing its best to provide a secure environment in which code executes. It accomplishes this task in part by using a bytecode verifier to make sure that the class file's bytecode is valid.
8. The answer is false: Java SE is the platform for developing applications and applets.
9. The JDK's `javac` tool is used to compile Java source code.
10. The JDK's `java` tool is used to run Java applications.
11. Standard I/O is a mechanism consisting of standard input, standard output, and standard error that makes it possible to read text from different sources (keyboard or file), write nonerror text to different destinations (screen or file), and write error text to different destinations (screen or file).
12. You specify the `main()` method's header as `public static void main(String[] args)`.
13. An IDE is a development framework consisting of a project manager for managing a project's files, a text editor for entering and editing source code, a debugger for locating bugs, and other features. The IDE that Google supports for developing Android apps is Android Studio.
14. Android is Google's software stack for mobile devices. This stack consists of apps (such as Browser and Contacts), a virtual machine in which apps run, middleware (software that sits on top of the operating system and provides various services to the virtual machine and its apps), and a Linux-based operating system.

Chapter 2: Learning Language Fundamentals

1. Unicode is a computing industry standard for consistently encoding, representing, and handling text that's expressed in most of the world's writing systems.
2. A comment is a language feature for embedding documentation in source code.
3. The three kinds of comments that Java supports are single-line, multiline, and Javadoc.
4. An identifier is a language feature that consists of letters (A-Z, a-z), digits (0-9), and the underscore (others are possible, but not used commonly or impractical). This name must begin with a letter or an underscore and cannot contain spaces.
5. The answer is false: Java is a case-sensitive language.
6. A type is a language feature that identifies a set of values (and their representation in memory) and a set of operations that transform these values into other values of that set.
7. A primitive type is a type that's defined by the language and whose values are not objects.
8. Java supports the Boolean, character, byte integer, short integer, integer, long integer, floating-point, and double precision floating-point primitive types.
9. An object type is a type that's defined by the developer using a class, an interface, an enum, or an annotation type and whose values are objects.
10. An array type is a special reference type that signifies an array, a region of memory that stores values in equal-size and contiguous slots, which are commonly referred to as elements.
11. A variable is a named memory location that stores some type of value.

APPENDIX A SOLUTIONS TO EXERCISES

12. An expression is a combination of literals, variable names, method calls, and operators. At runtime, it evaluates to a value whose type is referred to as the expression's type.
13. The two expression categories are simple expression and compound expression.
14. A literal is a value specified verbatim.
15. String literal "The quick brown fox \jumps\ over the lazy dog." is illegal because, unlike \" , \'j and \\ (a backslash followed by a space character) are not valid escape sequences. To make this string literal legal, you must escape these backslashes, as in "The quick brown fox \\jumps\\ over the lazy dog.".
16. An operator is a sequence of instructions symbolically represented in source code.
17. The difference between a prefix operator and a postfix operator is that a prefix operator precedes its operand and a postfix operator trails its operand.
18. The purpose of the cast operator is to convert from one type to another type. For example, you can use this operator to convert from floating-point type to 32-bit integer type.
19. Precedence refers to an operator's level of importance.
20. The answer is true: most of Java's operators are left-to-right associative.
21. A statement is a language feature that assigns a value to a variable, controls a program's flow by making a decision and/or repeatedly executing another statement, or performs another task.
22. The difference between the while and do-while statements is that the while statement evaluates its Boolean expression at the top of the loop, whereas the do-while statement evaluates its Boolean expression at the bottom of the loop. As a result, while executes zero or more times, whereas do-while executes one or more times.

23. The difference between the break and continue statements is that break transfers execution to the first statement following a switch statement or a loop, whereas continue skips the remainder of the current loop iteration, reevaluates the loop's Boolean expression, and performs another iteration (when true) or terminates the loop (when false).
24. Listing [A-1](#) presents the Compass application that was called for in Chapter [2](#).

Listing A-1. Finding a Direction in Which to Travel

```
public class Compass {
    public static void main(String[] args){
        int direction = 1;
        switch (direction) {
            case 0: System.out.println("You are travelling north."); break;
            case 1: System.out.println("You are travelling east."); break;
            case 2: System.out.println("You are travelling south."); break;
            case 3: System.out.println("You are travelling west."); break;
            default: System.out.println("You are lost.");
        }
    }
}
```

25. Listing [A-2](#) presents the Triangle application that was called for in Chapter [2](#).

Listing A-2. Printing a Triangle of Asterisks

```
public class Triangle {
    public static void main(String[] args){
        for (int row = 1; row < 20; row += 2){
            for (int col = 0; col < 19 - row / 2; col++)
                System.out.print(" ");
            for (int col = 0; col < row; col++)
                System.out.print("*");
        }
    }
}
```

```
        System.out.print('\n');
    }
}
}
```

Chapter 3: Discovering Classes and Objects

1. A class is a container for housing an application and is also a template for manufacturing objects.
2. You declare a class by minimally specifying reserved word `class` followed by a name that identifies the class (so that it can be referred to from elsewhere in the source code), followed by a body. The body starts with an open brace character (`{`) and ends with a close brace (`}`). Sandwiched between these delimiters are various kinds of member declarations.
3. The answer is false: you can declare only one public class in a source file.
4. An object is an instance of a class.
5. You obtain an object by using the `new` operator to allocate memory to store a class instance and a constructor to initialize this instance.
6. A constructor is a block of code for constructing an object by initializing it in some manner.
7. The answer is true: Java creates a default noargument constructor when a class declares no constructors.
8. A parameter list is a round bracket-delimited and comma-separated list of zero or more parameter declarations. A parameter is a constructor or method variable that receives an expression value passed to the constructor or method when it's called.
9. An argument list is a round bracket-delimited and comma-separated list of zero or more expressions. An argument is one of these expressions whose value is passed to the corresponding parameter when a constructor or method is called.

10. The answer is false: you invoke another constructor by specifying this followed by an argument list.
11. Arity is the number of arguments passed to a constructor or method or the number of operator operands.
12. A local variable is a variable that's declared in a constructor or method and isn't a member of the constructor or method parameter list.
13. Lifetime is a property of a variable that determines how long the variable exists. For example, parameters come into existence when a constructor or method is called and are destroyed when the constructor or method finishes. Similarly, an instance field comes into existence when an object is created and is destroyed when the object is garbage collected.
14. Scope is a property of a variable that determines how accessible the variable is to code. For example, a parameter can be accessed only by the code within the constructor or method in which the parameter is declared.
15. Encapsulation refers to the merging of state and behaviors into a single source code entity. Instead of separating state and behaviors, which is done in structured programs, state and behaviors are combined into classes and objects, which are the focus of object-based programs. For example, where a structured program makes you think in terms of separate balance state and deposit/withdraw behaviors, an object-based program makes you think in terms of bank accounts, which unite balance state with deposit/withdraw behaviors through encapsulation.
16. A field is a variable declared within a class body.
17. The difference between an instance field and a class field is that an instance field describes some attribute of the real-world entity that an object is modeling and is unique to each object, and a class field identifies some data item that's shared by all objects.

APPENDIX A SOLUTIONS TO EXERCISES

18. A blank final is a read-only instance field. It differs from a true constant in that there are multiple copies of blank finals (one per object) and only one true constant (one per class).
19. You prevent a field from being shadowed by changing the name of a same-named local variable or parameter, or by qualifying the local variable's name or parameter's name with `this` or the class name followed by the member access operator.
20. A method is a named block of code declared within a class body.
21. The difference between an instance method and a class method is that an instance method describes some behavior of the real-world entity that an object is modeling and can access a specific object's state, whereas a class method identifies some behavior that's common to all objects and cannot access a specific object's state.
22. Recursion is the act of a method invoking itself.
23. You overload a method by introducing a method with the same name as an existing method but with a different parameter list into the same class.
24. A class initializer is a `static`-prefixed block that's introduced into a class body. An instance initializer is a block that's introduced into a class body as opposed to being introduced as the body of a method or a constructor.
25. A garbage collector is code that runs in the background and occasionally checks for unreferenced objects.
26. The answer is false: `String[] letters = new String[2] { "A", "B" };` is incorrect syntax. Remove the 2 from between the square brackets to make it correct.
27. A ragged array is a two-dimensional array in which each row can have a different number of columns.
28. Listing A-3 presents the Image application that was called for in Chapter 3.

Listing A-3. Testing the Image Class

```
public class Image {  
    public Image() {  
        System.out.println("Image() called");  
    }  
  
    public Image(String filename) {  
        this(filename, null);  
        System.out.println("Image(String filename) called");  
    }  
  
    public Image(String filename, String imageType) {  
        System.out.println("Image(String filename, String imageType)  
                           called");  
        if (filename != null){  
            System.out.println("reading " + filename);  
            if (imageType != null)  
                System.out.println("interpreting " + filename + " as  
                                   storing a " + imageType + " image");  
        }  
        // Perform other initialization here.  
    }  
  
    public static void main(String[] args){  
        Image image = new Image();  
        System.out.println();  
        image = new Image("image.png");  
        System.out.println();  
        image = new Image("image.png", "PNG");  
    }  
}
```

29. Listing [A-4](#) presents the *Conversions* application that was called for in Chapter [3](#).

Listing A-4. Converting Between Degrees Fahrenheit and Degrees Celsius

```

public class Conversions {
    public static double c2f(double degrees) {
        return degrees * 9.0 / 5.0 + 32;
    }

    public static double f2c(double degrees) {
        return (degrees - 32) * 5.0 / 9.0;
    }

    public static void main(String[] args) {
        System.out.println("Fahrenheit equivalent of 100 degrees Celsius
is " + Conversions.c2f(100));
        System.out.println("Celsius equivalent of 98.6 degrees Fahrenheit
is " + Conversions.f2c(98.6));
        System.out.println("Celsius equivalent of 32 degrees Fahrenheit
is " + f2c(32));
    }
}

```

30. Listing [A-5](#) presents the Utilities application that was called for in Chapter [3](#).

Listing A-5. Calculating Factorials and Summing a Variable Number of Double Precision Floating-Point Values

```

public class Utilities {
    public static int factorial1(int n) {
        int product = 1;
        for (int i = 2; i <= n; i++)
            product *= i;
        return product;
    }
}

```

```

public static int factorial2(int n) {
    if (n == 0 || n == 1)
        return 1; // base problem
    else
        return n * factorial2(n - 1);
}

public static double sum(double... values) {
    int total = 0;
    for (int i = 0; i < values.length; i++)
        total += values[i];
    return total;
}

public static void main(String[] args) {
    System.out.println(factorial1(4));
    System.out.println(factorial2(4));
    System.out.println(factorial2(0));
    System.out.println(factorial2(1));
    System.out.println(sum(10.0, 20.0));
    System.out.println(sum(30.0, 40.0, 50.0));
}
}

```

31. Listing A-6 presents the GCD application that was called for in Chapter 3.

Listing A-6. Recursively Calculating the Greatest Common Divisor

```

public class GCD {
    public static int gcd(int a, int b) {
        // The greatest common divisor is the largest positive integer that
        // divides evenly into two positive integers a and b. For example,
        // GCD(12, 18) is 6.

        if (b == 0) // Base problem
            return a;
    }
}

```

APPENDIX A SOLUTIONS TO EXERCISES

```
    else
        return gcd(b, a % b);
}
}

public static void main(String[] args) {
    System.out.println(gcd(12, 18));
}
}
```

32. Listing [A-7](#) presents the Book application that was called for in Chapter [3](#).

Listing A-7. Building a Library of Books

```
public class Book {
    private String name;
    private String author;
    private String isbn;

    public Book(String name, String author, String isbn) {
        this.name = name;
        this.author = author;
        this.isbn = isbn;
    }

    public String getName(){
        return name;
    }

    public String getAuthor(){
        return author;
    }

    public String getISBN(){
        return isbn;
    }
}
```

```

public static void main(String[] args) {
    Book[] books = new Book[] {
        new Book("Jane Eyre",
                 "Charlotte Brontë",
                 "0895772000"),
        new Book("A Kick in the Seat of the Pants",
                 "Roger von Oech",
                 "0060155280"),
        new Book("The Prince and the Pilgrim",
                 "Mary Stewart",
                 "0340649925")
    };
    for (int i = 0; i < books.length; i++)
        System.out.println(books[i].getName() + " - " +
                           books[i].getAuthor() + " - " +
                           books[i].getISBN());
}
}

```

Chapter 4: Discovering Inheritance, Polymorphism, and Interfaces

1. Implementation inheritance is inheritance through class extension.
2. Java supports implementation inheritance by providing reserved word `extends`.
3. A subclass can have only one superclass because Java doesn't support multiple implementation inheritance.
4. You prevent a class from being subclassed by declaring the class `final`.
5. The answer is false: the `super()` call can only appear in a constructor.

APPENDIX A SOLUTIONS TO EXERCISES

6. If a superclass declares a constructor with one or more parameters, and if a subclass constructor doesn't use `super()` to call that constructor, the compiler reports an error because the subclass constructor attempts to call a nonexistent noargument constructor in the superclass. (When a class doesn't declare any constructors, the compiler creates a constructor with no parameters, a noargument constructor, for that class. Therefore, if the superclass didn't declare any constructors, a noargument constructor would be created for the superclass. Continuing, if the subclass constructor didn't use `super()` to call the superclass constructor, the compiler would insert the call and there would be no error.)
7. An immutable class is a class whose instances cannot be modified.
8. The answer is false: a class cannot inherit constructors.
9. Overriding a method means replacing an inherited method with another method that provides the same signature and the same return type but provides a new implementation.
10. To call a superclass method from its overriding subclass method, prefix the superclass method name with reserved word `super` and the member access operator in the method call.
11. You prevent a method from being overridden by declaring the method `final`.
12. You cannot make an overriding subclass method less accessible than the superclass method it is overriding because subtype polymorphism would not work properly if subclass methods could be made less accessible. Suppose you upcast a subclass instance to superclass type by assigning the instance's reference to a variable of superclass type. Now suppose you specify a superclass method call on the variable. If this method is overridden by the subclass, the subclass version of the method is called. However, if access to the subclass's overriding method's access could be made private, calling this method would break encapsulation; private methods cannot be called directly from outside of their class.

13. You tell the compiler that a method overrides another method by prefixing the overriding method's header with the `@Override` annotation.
14. Java doesn't support multiple implementation inheritance because this form of inheritance can lead to ambiguities.
15. The name of Java's ultimate superclass is `Object`. This class is located in the `java.lang` package.
16. The purpose of the `clone()` method is to duplicate an object without calling a constructor.
17. `Object`'s `clone()` method throws `CloneNotSupportedException` when the class whose instance is to be shallowly cloned doesn't implement the `Cloneable` interface.
18. The `==` operator cannot be used to determine if two objects are logically equivalent because this operator only compares object references and not the contents of these objects.
19. `Object`'s `equals()` method compares the current object's `this` reference to the reference passed as an argument to this method. (When we refer to `Object`'s `equals()` method, we are referring to the `equals()` method in the `Object` class.)
20. You can optimize a time-consuming `equals()` method by first using `==` to determine if this method's reference argument identifies the current object (which is represented in source code via reserved word `this`).
21. The purpose of the `finalize()` method is to provide a safety net for calling an object's cleanup method in case that method isn't called.
22. You shouldn't rely on `finalize()` for closing open files because file descriptors are a limited resource and an application might not be able to open additional files until `finalize()` is called, and this method might be called infrequently (or perhaps not at all).
23. A hash code is a number value that results from applying a mathematical function to a potentially large amount of data.

APPENDIX A SOLUTIONS TO EXERCISES

24. The answer is true: you should override the `hashCode()` method whenever you override the `equals()` method.
25. `Object`'s `toString()` method returns a string representation of the current object that consists of the object's class name, followed by the @ symbol, followed by a hexadecimal representation of the object's hash code. (When we refer to `Object`'s `toString()` method, we're referring to the `toString()` method in the `Object` class.)
26. You should override `toString()` to provide a concise but meaningful description of the object to facilitate debugging via `System.out.println()` method calls. It's more informative for `toString()` to reveal object state than to reveal a class name, followed by the @ symbol, followed by a hexadecimal representation of the object's hash code.
27. Composition is a way to reuse code by composing classes out of other classes based on a “has-a” relationship between them.
28. The answer is false: composition is used to describe “has-a” relationships and implementation inheritance is used to describe “is-a” relationships.
29. Subtype polymorphism is a kind of polymorphism where a subtype instance appears in a supertype context, and executing a supertype operation on the subtype instance results in the subtype's version of that operation executing.
30. Subtype polymorphism is accomplished by upcasting the subtype instance to its supertype, by assigning the instance's reference to a variable of that type, and, via this variable, calling a superclass method that's been overridden in the subclass.
31. You would use abstract classes and abstract methods to describe generic concepts (such as shape, animal, or vehicle) and generic operations (such as drawing a generic shape). Abstract classes cannot be instantiated and abstract methods cannot be called because they have no code bodies.

32. An abstract class can contain concrete methods.
33. The purpose of downcasting is to access subtype features. For example, you would downcast a `Point` variable that contains a `Circle` instance reference to the `Circle` type so that you can call `Circle's getRadius()` method on the instance.
34. Two forms of RTTI are the virtual machine verifying that a cast is legal and using the `instanceof` operator to determine whether or not an instance is a member of a type.
35. A covariant return type is a method return type that, in the superclass's method declaration, is the supertype of the return type in the subclass's overriding method declaration.
36. You formally declare an interface by specifying at least reserved word `interface`, followed by a name, followed by a brace-delimited body of method headers (constants are also allowed, but they are discouraged).
37. A marker interface is an interface that declares no members.
38. Interface inheritance is inheritance through interface implementation or interface extension.
39. You implement an interface by appending an `implements` clause, consisting of reserved word `implements` followed by the interface's name, to a class header and by overriding the interface's method headers in the class.
40. You form a hierarchy of interfaces by appending reserved word `extends` followed by an interface name to an interface header.
41. Java's interfaces feature is so important because it gives developers the utmost flexibility in designing their applications.
42. Interfaces and abstract classes describe abstract types.
43. Interfaces and abstract classes differ in that interfaces can only declare abstract methods and constants and can be implemented by any class in any class hierarchy. In contrast, abstract classes

APPENDIX A SOLUTIONS TO EXERCISES

can declare constants and nonconstant fields, can declare abstract and concrete methods, and can only appear in the upper levels of class hierarchies, where they're used to describe abstract concepts and behaviors.

44. Listings [A-8](#) through [A-14](#) declare the `Animal`, `Bird`, `Fish`, `AmericanRobin`, `DomesticCanary`, `RainbowTrout`, and `SockeyeSalmon` classes that were called for in Chapter [4](#).

Listing A-8. The `Animal` Class Abstracting over Birds and Fish (and Other Organisms)

```
public abstract class Animal {  
    private String kind;  
    private String appearance;  
  
    public Animal(String kind, String appearance) {  
        this.kind = kind;  
        this.appearance = appearance;  
    }  
  
    public abstract void eat();  
  
    public abstract void move();  
  
    @Override  
    public String toString() {  
        return kind + " -- " + appearance;  
    }  
}
```

Listing A-9. The `Bird` Class Abstracting over American Robins, Domestic Canaries, and Other Kinds of Birds

```
public abstract class Bird extends Animal {  
    public Bird(String kind, String appearance){  
        super(kind, appearance);  
    }  
}
```

```

@Override
public void eat() {
    System.out.println("eats seeds and insects");
}

@Override
public void move() {
    System.out.println("flies through the air");
}

}

```

Listing A-10. The Fish Class Abstracting over Rainbow Trout, Sockeye Salmon, and Other Kinds of Fish

```

public abstract class Fish extends Animal {
    public Fish(String kind, String appearance) {
        super(kind, appearance);
    }

    @Override
    public void eat() {
        System.out.println("eats krill, algae, and insects");
    }

    @Override
    public void move() {
        System.out.println("swims through the water");
    }
}

```

Listing A-11. The AmericanRobin Class Denoting a Bird with a Red Breast

```

public class AmericanRobin extends Bird {
    public AmericanRobin() {
        super("americanrobin", "red breast");
    }
}

```

Listing A-12. The DomesticCanary Class Denoting a Bird of Various Colors

```
public class DomesticCanary extends Bird {
    public DomesticCanary() {
        super("domestic canary", "yellow, orange, black, brown, white, red");
    }
}
```

Listing A-13. The RainbowTrout Class Denoting a Rainbow-Colored Fish

```
public class RainbowTrout extends Fish {
    public RainbowTrout() {
        super("rainbowtrout", "bands of brilliant speckled multicolored " +
              "stripes running nearly the whole length of its body");
    }
}
```

Listing A-14. The SockeyeSalmon Class Denoting a Red-and-Green Fish

```
public class SockeyeSalmon extends Fish {
    public SockeyeSalmon() {
        super("sockeyesalmon", "bright red with a green head");
    }
}
```

45. Listing [A-15](#) declares the `Animals` class that was called for in Chapter [4](#).

Listing A-15. The Animals Class Letting Animals Eat and Move

```
public class Animals {
    public static void main(String[] args) {
        Animal[] animals = { new AmericanRobin(), new RainbowTrout(),
                            new DomesticCanary(), new SockeyeSalmon() };
        for (int i = 0; i < animals.length; i++) {
            System.out.println(animals[i]);
            animals[i].eat();
            animals[i].move();
        }
    }
}
```

```

        System.out.println();
    }
}
}
}
```

46. Listings [A-16](#) through [A-18](#) declare the Countable interface, the modified Animal class, and the modified Animals class that were called for in Chapter [4](#).

Listing A-16. The Countable Interface for Use in Taking a Census of Animals

```

public interface Countable {
    String getID();
}
```

Listing A-17. The Refactored Animal Class for Help in Census Taking

```

public abstract class Animal implements Countable {
    private String kind;
    private String appearance;

    public Animal(String kind, String appearance) {
        this.kind = kind;
        this.appearance = appearance;
    }

    public abstract void eat();

    public abstract void move();

    @Override
    public String toString() {
        return kind + " -- " + appearance;
    }

    @Override
    public String getID() {
        return kind;
    }
}
```

Listing A-18. The Modified Animals Class for Carrying Out the Census

```
public class Animals {  
    public static void main(String[] args) {  
        Animal[] animals = { new AmericanRobin(), new RainbowTrout(),  
                            new DomesticCanary(), new SockeyeSalmon(),  
                            new RainbowTrout(), new AmericanRobin() };  
        for (int i = 0; i < animals.length; i++) {  
            System.out.println(animals[i]);  
            animals[i].eat();  
            animals[i].move();  
            System.out.println();  
        }  
  
        Census census = new Census();  
        Countable[] countables = (Countable[]) animals;  
        for (int i = 0; i < countables.length; i++)  
            census.update(countables[i].getID());  
  
        for (int i = 0; i < Census.SIZE; i++)  
            System.out.println(census.get(i));  
    }  
}
```

Chapter 5: Mastering Advanced Language Features, Part 1

1. A nested class is a class that's declared as a member of another class or scope.
2. The four kinds of nested classes are static member classes, nonstatic member classes, anonymous classes, and local classes.
3. Nonstatic member classes, anonymous classes, and local classes are also known as inner classes.

4. The answer is false: a static member class doesn't have an enclosing instance.
5. You instantiate a nonstatic member class from beyond its enclosing class by first instantiating the enclosing class and then prefixing the new operator with the enclosing class instance as you instantiate the enclosed class. For example, new EnclosingClass().new EnclosedClass().
6. It's necessary to declare local variables and parameters final when they are being accessed by an instance of an anonymous class or a local class.
7. The answer is true: an interface can be declared within a class or within another interface.
8. A package is a unique namespace that can contain a combination of top-level classes, other top-level types, and subpackages.
9. You ensure that package names are unique by specifying your reversed Internet domain name as the top-level package name.
10. A package statement is a statement that identifies the package in which a source file's types are located.
11. The answer is false: you cannot specify multiple package statements in a source file.
12. An import statement is a statement that imports types from a package by telling the compiler where to look for unqualified type names during compilation.
13. You indicate that you want to import multiple types via a single import statement by specifying the wildcard character (*).
14. During a runtime search, the virtual machine reports a "no class definition found" error when it cannot find a class file.
15. You specify the user classpath to the virtual machine via the -classpath (or -cp) option used to start the virtual machine or, when not present, the CLASSPATH environment variable.

APPENDIX A SOLUTIONS TO EXERCISES

16. A static import statement is a version of the import statement that lets you import a class's static members so that you don't have to qualify them with their class names.
17. You specify a static import statement as `import`, followed by `static`, followed by a member access operator-separated list of package and subpackage names, followed by the member access operator, followed by a class's name, followed by the member access operator, followed by a single static member name or the asterisk wildcard, for example, `import static java.lang.Math.cos;` (import the `cos()` static method from the `Math` class).
18. An exception is a divergence from an application's normal behavior.
19. Objects are superior to error codes for representing exceptions because error code Boolean or integer values are less meaningful than object names and because objects can contain information about what led to the exception. These details can be helpful to a suitable workaround. Furthermore, error codes are easy to ignore.
20. A throwable is an instance of `Throwable` or one of its subclasses.
21. `Exception` describes exceptions that result from external factors (such as not being able to open a file) and from flawed code (such as passing an illegal argument to a method). `Error` describes virtual machine-oriented exceptions such as running out of memory or being unable to load a class file.
22. A checked exception is an exception that represents a problem with the possibility of recovery and for which the developer must provide a workaround.
23. A runtime exception is an exception that represents a coding mistake.
24. You would introduce your own exception class when no existing exception class in the standard class library meets your needs.

25. The answer is false: you use a throws clause to identify exceptions that are thrown from a method by appending this clause to a method's header.
26. The purpose of a try statement is to provide a scope (via its brace-delimited body) in which to present code that can throw exceptions. The purpose of a catch block is to receive a thrown exception and provide code (via its brace-delimited body) that handles that exception by providing a workaround.
27. The purpose of a finally block is to provide cleanup code that's executed whether an exception is thrown or not.
28. Listing [A-19](#) presents the G2D class that was called for in Chapter [5](#).

Listing A-19. The G2D Class with Its Matrix Nonstatic Member Class

```
public class G2D {
    private Matrix xform;

    public G2D(){
        xform = new Matrix();
        xform.a = 1.0;
        xform.e = 1.0;
        xform.i = 1.0;
    }

    private class Matrix {
        double a, b, c;
        double d, e, f;
        double g, h, i;
    }
}
```

29. To extend the logging package (presented in Chapter [5](#)'s discussion of packages) to support a null device in which messages are thrown away, first introduce Listing [A-20](#)'s NullDevice package-private class.

Listing A-20. Implementing the Proverbial “Bit Bucket” Class

```
package logging;

public class NullDevice implements Logger {
    private String dstName;

    public NullDevice(String dstName) {
    }

    public boolean connect() {
        return true;
    }

    public boolean disconnect(){
        return true;
    }

    public boolean log(String msg){
        return true;
    }
}
```

Continue by introducing, into the LoggerFactory class, a NULLDEVICE constant and code that instantiates NullDevice with a null argument—a destination name isn’t required—when newLogger()’s dstType parameter contains this constant’s value. Check out Listing A-21.

Listing A-21. A Refactored LoggerFactory Class

```
package logging;

public abstract class LoggerFactory {
    public final static int CONSOLE = 0;
    public final static int FILE = 1;
    public final static int NULLDEVICE = 2;

    public static Logger newLogger(int dstType, String...dstName){
        switch (dstType) {
            case CONSOLE : return new Console(dstName.length == 0 ?
                null : dstName[0]);
        }
    }
}
```

```

        case FILE      : return new File(dstName.length == 0 ?
            null : dstName[0]);
        case NULLDEVICE: return new NullDevice(null);
        default       : return null;
    }
}
}

```

30. Modifying the logging package (presented in Chapter 5's discussion of packages) so that Logger's connect() method throws a CannotConnectException instance when it cannot connect to its logging destination, and the other two methods each throw a NotConnectedException instance when connect() was not called or when it threw a CannotConnectException instance, results in Listing A-22's Logger interface.

Listing A-22. A Logger Interface Whose Methods Throw Exceptions

```

package logging;

public interface Logger {
    void connect() throws CannotConnectException;
    void disconnect() throws NotConnectedException;
    void log(String msg) throws NotConnectedException;
}

```

Listing A-23 presents the CannotConnectException class.

Listing A-23. An Uncomplicated CannotConnectException Class

```

package logging;

public class CannotConnectException extends Exception {
}

```

The NotConnectedException class has the same structure but with a different name. Listing A-24 presents the Console class.

Listing A-24. The Console Class Satisfying Logger's Contract Without Throwing Exceptions

```
package logging;

public class Console implements Logger {
    private String dstName;

    public Console(String dstName) {
        this.dstName = dstName;
    }

    public void connect() throws CannotConnectException {
    }

    public void disconnect() throws NotConnectedException {
    }

    public void log(String msg) throws NotConnectedException {
        System.out.println(msg);
    }
}
```

[Listing A-25](#) presents the File class.

Listing A-25. The File Class Satisfying Logger's Contract by Throwing Exceptions As Necessary

```
package logging;

public class File implements Logger {
    private String dstName;

    public File(String dstName) {
        this.dstName = dstName;
    }

    public void connect() throws CannotConnectException {
        if (dstName == null)
            throw new CannotConnectException();
    }
}
```

```

public void disconnect() throws NotConnectedException {
    if (dstName == null)
        throw new NotConnectedException();
}

public void log(String msg) throws NotConnectedException {
    if (dstName == null)
        throw new NotConnectedException();
    System.out.println("writing " + msg + " to file " + dstName);
}
}

```

31. When you modify `TestLogger` to respond appropriately to thrown `CannotConnectException` and `NotConnectedException` objects, you end up with something similar to Listing A-26.

Listing A-26. A `TestLogger` Class That Handles Thrown Exceptions

```

import logging.*;

public class TestLogger {
    public static void main(String[] args) {
        try {
            Logger logger = LoggerFactory.newLogger(LoggerFactory.CONSOLE);
            logger.connect();
            logger.log("test message #1");
            logger.disconnect();
        } catch (CannotConnectException cce) {
            System.err.println("cannot connect to console-based logger");
        } catch (NotConnectedException nce) {
            System.err.println("not connected to console-based logger");
        }
        try {
            Logger logger = LoggerFactory.newLogger(LoggerFactory.FILE,
                "x.txt");
            logger.connect();

```

```
    logger.log("test message #2");
    logger.disconnect();
} catch (CannotConnectException cce) {
    System.err.println("cannot connect to file-based logger");
} catch (NotConnectedException nce) {
    System.err.println("not connected to file-based logger");
}

try {
    Logger logger = LoggerFactory.newLogger(LoggerFactory.FILE);
    logger.connect();
    logger.log("test message #3");
    logger.disconnect();
} catch (CannotConnectException cce) {
    System.err.println("cannot connect to file-based logger");
} catch (NotConnectedException nce) {
    System.err.println("not connected to file-based logger");
}
}
```

Chapter 6: Mastering Advanced Language Features, Part 2

1. An annotation is an instance of an annotation type and associates metadata with an application element. It's expressed in source code by prefixing the type name with the @ symbol.
2. Constructors, fields, local variables, methods, packages, parameters, and types (annotation, class, enum, and interface) can be annotated.
3. The three compiler-supported annotation types are `Override`, `Deprecated`, and `SuppressWarnings`.

4. You declare an annotation type by specifying the @ symbol, immediately followed by reserved word `interface`, followed by the type's name, followed by a body.
5. A marker annotation is an instance of an annotation type that supplies no data apart from its name; the type's body is empty.
6. An element is a method header that appears in the annotation type's body. It cannot have parameters or a throws clause. Its return type must be primitive (such as `int`), `String`, `Class`, an enum type, an annotation type, or an array of the preceding types. It can have a default value.
7. You assign a default value to an element by specifying `default` followed by the value, whose type must match the element's return type. For example, `String developer() default "unassigned";`.
8. A meta-annotation is an annotation that annotates an annotation type.
9. Java's four meta-annotation types are `Target`, `Retention`, `Documented`, and `Inherited`.
10. Generics can be defined as a suite of language features for declaring and using type-agnostic classes and interfaces.
11. You would use generics to ensure that your code is typesafe by avoiding thrown `ClassCastException`s.
12. The difference between a generic type and a parameterized type is that a generic type is a class or interface that introduces a family of parameterized types by declaring a formal type parameter list, and a parameterized type is an instance of a generic type.
13. Anonymous classes cannot be generic because they have no names.
14. The five kinds of actual type arguments are concrete types, concrete parameterized types, array types, type parameters, and wildcards.

APPENDIX A SOLUTIONS TO EXERCISES

15. The answer is true: you cannot specify a primitive-type name (such as `double` or `int`) as an actual type argument.
16. A raw type is a generic type without its type parameters.
17. The compiler reports an unchecked warning message when it detects an explicit cast that involves a type parameter. The compiler is concerned that downcasting to whatever type is passed to the type parameter might result in a violation of type safety.
18. You suppress an unchecked warning message by prefixing the constructor or method that contains the unchecked code with the `@SuppressWarnings("unchecked")` annotation.
19. The answer is true: `List<E>`'s `E` type parameter is unbounded.
20. You specify a single upper bound via reserved word `extends` followed by a type name.
21. A recursive type bound is a type parameter bound that includes the type parameter.
22. Wildcard type arguments are necessary because by accepting any actual type argument they provide a typesafe workaround to the problem of polymorphic behavior not applying to multiple parameterized types that differ only in regard to one type parameter being a subtype of another type parameter. For example, because `List<String>` isn't a kind of `List<Object>`, you cannot pass an object whose type is `List<String>` to a method parameter whose type is `List<Object>`. However, you can pass a `List<String>` object to `List<?>`, provided that you're not going to add the `List<String>` object to the `List<?>`.
23. A generic method is a class or instance method with a type-generalized implementation.
24. Although you might think otherwise, Listing 6-28's `methodCaller()` generic method calls `someOverloadedMethod(0bject o)`. This method, instead of `someOverloadedMethod(Date d)`, is called because overload resolution happens at compile time, when the generic method is translated to its unique bytecode representation,

and erasure (which takes care of that mapping) causes type parameters to be replaced by their leftmost bound or `Object` (when there is no bound). After erasure, you are left with Listing A-27's nongeneric `methodCaller()` method.

Listing A-27. The Nongeneric `methodCaller()` Method That Results from Erasure

```
public static void methodCaller(Object t) {  
    someOverloadedMethod(t);  
}
```

25. Reification is representing the abstract as if it was concrete.
26. The answer is false: type parameters are not reified.
27. Erasure is the throwing away of type parameters following compilation so that they are not available at runtime. Erasure also involves replacing uses of other type variables by the upper bound of the type variable (such as `Object`) and inserting casts to the appropriate type when the resulting code isn't type correct.
28. An enumerated type is a type that specifies a named sequence of related constants as its legal values.
29. Three problems that can arise when you use enumerated types whose constants are `int`-based are lack of compile-time type safety, brittle applications, and the inability to translate `int` constants into meaningful string-based descriptions.
30. An enum is an enumerated type that's expressed via reserved word `enum`.
31. You use a switch statement with an enum by specifying an enum constant as the statement's selector expression and constant names as case values.
32. You can enhance an enum by adding fields, constructors, and methods; you can even have the enum implement interfaces. Also, you can override `toString()` to provide a more useful description of a constant's value and subclass constants to assign different behaviors.

APPENDIX A SOLUTIONS TO EXERCISES

33. The purpose of the abstract `Enum` class is to serve as the common base class of all Java language-based enumeration types.
34. The difference between `Enum`'s `name()` and `toString()` methods is that `name()` always returns a constant's name, but `toString()` can be overridden to return a more meaningful description instead of the constant's name.
35. The answer is true: `Enum`'s generic type is `Enum<E extends Enum<E>>`.
36. Listing A-28 presents a `ToDo` marker annotation type that annotates only type elements and that also uses the default retention policy.

Listing A-28. The `ToDo` Annotation Type for Marking Types That Need to Be Completed

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
public @interface ToDo { }
```

37. Listing A-29 presents a rewritten `StubFinder` application that works with Listing 6-4's `Stub` annotation type (with appropriate `@Target` and `@Retention` annotations) and Listing 6-5's `Deck` class.

Listing A-29. Reporting a Stub's ID, Due Date, and Developer via a New Version of `StubFinder`

```
import java.lang.reflect.Method;

public class StubFinder {
    public static void main(String[] args) throws Exception {
        if (args.length != 1){
            System.err.println("usage: java StubFinder classfile");
            return;
    }}
```

```

    }
    Method[] methods = Class.forName(args[0]).getMethods();
    for (int i = 0; i < methods.length; i++)
        if (methods[i].isAnnotationPresent(Stub.class)){
            Stub stub = methods[i].getAnnotation(Stub.class);
            System.out.println("Stub ID = " + stub.id());
            System.out.println("Stub Date = " + stub.dueDate());
            System.out.println("Stub Developer = " + stub.developer());
            System.out.println();
        }
    }
}

```

38. Listing A-30 presents the generic Stack class and the StackEmptyException and StackFullException helper classes that were called for in Chapter 6.

Listing A-30. Stack and Its StackEmptyException and StackFullException Helper Classes Proving That Not All Helper Classes Need to Be Nested

```

public class Stack<E> {
    private E[] elements;
    private int top;

    @SuppressWarnings("unchecked")
    public Stack(int size) {
        if (size < 2)
            throw new IllegalArgumentException("" + size);
        elements = (E[]) new Object[size];
        top = -1;
    }

    public void push(E element) throws StackFullException {
        if (top == elements.length - 1)
            throw new StackFullException();
        elements[++top] = element;
    }
}

```

APPENDIX A SOLUTIONS TO EXERCISES

```
public E pop() throws StackEmptyException {
    if (isEmpty())
        throw new StackEmptyException();
    return elements[top--];
}

public boolean isEmpty() {
    return top == -1;
}

public static void main(String[] args)
    throws StackFullException, StackEmptyException
{
    Stack<String> stack = new Stack<String>(5);
    assert stack.isEmpty();
    stack.push("A");
    stack.push("B");
    stack.push("C");
    stack.push("D");
    stack.push("E");
    // Uncomment the following line to generate a StackFullException.
    //stack.push("F");
    while (!stack.isEmpty())
        System.out.println(stack.pop());
    // Uncomment the following line to generate a StackEmptyException.
    //stack.pop();
    assert stack.isEmpty();
}

class StackEmptyException extends Exception {
}

class StackFullException extends Exception {
}
```

39. Listing [A-31](#) presents the `Compass` enum that was called for in Chapter [6](#).

Listing A-31. A Compass Enum with Four Direction Constants

```
enum Compass {
    NORTH, SOUTH, EAST, WEST
}
```

Listing A-32 presents the UseCompass class that was called for in Chapter 6.

Listing A-32. Using the Compass Enum to Keep from Getting Lost

```
public class UseCompass {
    public static void main(String[] args) {
        int i = (int) (Math.random() * 4);
        Compass[] dir = { Compass.NORTH, Compass.EAST, Compass.SOUTH,
                          Compass.WEST };
        switch(dir[i]) {
            case NORTH: System.out.println("heading north"); break;
            case EAST : System.out.println("heading east"); break;
            case SOUTH: System.out.println("heading south"); break;
            case WEST : System.out.println("heading west"); break;
            default   : assert false; // Should never be reached.
        }
    }
}
```

Chapter 7: Exploring the Basic APIs, Part 1

1. Math declares double constants E and PI that represent, respectively, the natural logarithm base value (2.71828...) and the ratio of a circle's circumference to its diameter (3.14159...). E is initialized to 2.718281828459045 and PI is initialized to 3.141592653589793.
2. Math.abs(Integer.MIN_VALUE) equals Integer.MIN_VALUE because there doesn't exist a positive 32-bit integer equivalent of MIN_VALUE. (Integer.MIN_VALUE equals -2147483648 and Integer.MAX_VALUE equals 2147483647.)

APPENDIX A SOLUTIONS TO EXERCISES

3. Math's `random()` method returns a pseudorandom number between 0.0 (inclusive) and 1.0 (exclusive).
4. The five special values that can arise during floating-point calculations are `+infinity`, `-infinity`, `NaN`, `+0.0`, and `-0.0`.
5. `BigDecimal` is an immutable class that represents a signed decimal number (such as 23.653) of arbitrary precision with an associated scale. You might use this class to store floating-point values accurately, which represent monetary values and properly round the result of each monetary calculation.
6. The `RoundingMode` constant that describes the form of rounding commonly taught at school is `HALF_UP`.
7. `BigInteger` is an immutable class that represents a signed integer of arbitrary precision. It stores its value in two's complement format.
8. A primitive type wrapper class is a class whose instances wrap themselves around primitive-type values.
9. Java's primitive type wrapper classes are `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short`.
10. Java provides primitive type wrapper classes to facilitate storing primitive-type values in collections and as a convenient place to associate useful constants and class methods with the primitive types.
11. The answer is false: `Boolean` is the smallest of the primitive type wrapper classes.
12. You should use `Character` class methods instead of expressions such as `ch >= '0' && ch <= '9'` to determine whether or not a character is a digit, a letter, and so on because it's too easy to introduce a bug into the expression, expressions are not very descriptive of what they're testing, and the expressions are biased toward Latin digits (0–9) and letters (A–Z and a–z).

13. You determine whether or not double variable `d` contains `+infinity` or `-infinity` by passing this variable as an argument to `Double`'s `boolean isInfinite(double d)` class method, which returns true when this argument is `+infinity` or `-infinity`.
14. `Number` is the superclass of `Byte`, `Character`, and the other primitive type wrapper classes.
15. The answer is true: a string literal is a `String` object.
16. The purpose of `String`'s `intern()` method is to store a unique copy of a `String` object in an internal table of `String` objects. `intern()` makes it possible to compare strings via their references and `==` or `!=`. These operators are the fastest way to compare strings, which is especially valuable when sorting a huge number of strings.
17. `String` and `StringBuffer` differ in that `String` objects contain immutable sequences of characters, whereas `StringBuffer` objects contain mutable sequences of characters.
18. `StringBuffer` and `StringBuilder` differ in that `StringBuffer` methods are synchronized, whereas `StringBuilder`'s equivalent methods are not synchronized. As a result, you would use the thread-safe but slower `StringBuffer` class in multithreaded situations and the nonthread-safe but faster `StringBuilder` class in single-threaded situations.
19. You invoke the `System.arraycopy()` method to copy an array to another array.
20. You invoke the `System.currentTimeMillis()` method to obtain the current time in milliseconds.
21. A thread is an independent path of execution through an application's code.
22. The purpose of the `Runnable` interface is to identify those objects that supply code for threads to execute via this interface's solitary `void run()` method.

APPENDIX A SOLUTIONS TO EXERCISES

23. The purpose of the `Thread` class is to provide a consistent interface to the underlying operating system's threading architecture. It provides methods that make it possible to associate code with threads as well as to start and manage those threads.
24. The answer is false: a `Thread` object associates with a single thread.
25. A race condition is a scenario in which multiple threads are accessing shared data, and the final result of these accesses is dependent on the timing of how the threads are scheduled. Race conditions can lead to bugs that are hard to find and results that are unpredictable.
26. Synchronization is the act of allowing only one thread at a time to execute code within a method or a block.
27. Synchronization is implemented in terms of monitors and locks.
28. Synchronization works by requiring that a thread that wants to enter a monitor-controlled critical section first acquires a lock. The lock is released automatically when the thread exits the critical section.
29. The answer is true: variables of type `long` or `double` are not atomic on 32-bit virtual machines.
30. The purpose of reserved word `volatile` is to let threads running on multiprocessor or multicore machines access the main memory copy of an instance field or class field. Without `volatile`, each thread might access its cached copy of the field and won't see modifications made by other threads to their copies.
31. The answer is false: `Object`'s `wait()` methods cannot be called from outside of a synchronized method or block.
32. Deadlock is a situation where locks are acquired by multiple threads, neither thread holds its own lock but holds the lock needed by some other thread, and neither thread can enter and later exit its critical section to release its held lock because some other thread holds the lock to that critical section.

33. The purpose of the `ThreadLocal` class is to associate per-thread data (such as a user ID) with a thread.
34. Listing [A-33](#) presents the `PrimeNumberTest` application that was called for in Chapter [7](#).

Listing A-33. Checking a Positive Integer Argument to Discover If It's Prime

```
public class PrimeNumberTest {
    public static void main(String[] args) {
        if (args.length != 1){
            System.err.println("usage: java PrimeNumberTest integer");
            System.err.println("integer must be 2 or higher");
            return;
        }
        try {
            int n = Integer.parseInt(args[0]);
            if (n < 2) {
                System.err.println(n + " is invalid because it is less than 2");
                return;
            }
            for (int i = 2; i <= Math.sqrt(n); i++)
                if (n % i == 0){
                    System.out.println (n + " is not prime");
                    return;
                }
            System.out.println(n + " is prime");
        } catch (NumberFormatException nfe) {
            System.err.println("unable to parse " + args[0] + " into an int");
        }
    }
}
```

35. Listing [A-34](#) presents the `MultiPrint` application that was called for in Chapter [7](#).

Listing A-34. Printing a Line of Text Multiple Times

```
public class MultiPrint {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java MultiPrint text count");
            return;
        }
        String text = args[0];
        int count = Integer.parseInt(args[1]);
        for (int i = 0; i < count; i++)
            System.out.println(text);
    }
}
```

36. The following loop uses `StringBuffer` to minimize object creation:

```
String[] imageNames = new String[NUM_IMAGES];
StringBuffer sb = new StringBuffer();
for (int i = 0; i < imageNames.length; i++) {
    sb.append("image");
    sb.append(i);
    sb.append(".png");
    imageNames[i] = sb.toString();
    sb.setLength(0); // Erase previous StringBuffer contents.
}
```

37. Listing [A-35](#) presents the `DigitsToWords` application that was called for in Chapter [7](#).

Listing A-35. Converting an Integer Value to Its Textual Representation

```
public class DigitsToWords {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage: java DigitsToWords integer");
```

```
        return;
    }
    System.out.println(convertDigitsToWords(Integer.parseInt(args[0])));
}

public static String convertDigitsToWords(int integer) {
    if (integer < 0 || integer > 9999)
        throw new IllegalArgumentException("Out of range: " + integer);
    if (integer == 0)
        return "zero";
    String[] group1 = {
        "one",
        "two",
        "three",
        "four",
        "five",
        "six",
        "seven",
        "eight",
        "nine"
    };
    String[] group2 = {
        "ten",
        "eleven",
        "twelve",
        "thirteen",
        "fourteen",
        "fifteen",
        "sixteen",
        "seventeen",
        "eighteen",
        "nineteen"
    };
    String[] group3 = {
        "twenty",
        "thirty",
```

APPENDIX A SOLUTIONS TO EXERCISES

```
"forty",
"fifty",
"sixty",
"seventy",
"eighty",
"ninety"
};

StringBuffer result = new StringBuffer();
if (integer >= 1000) {
    int tmp = integer / 1000;
    result.append(group1[tmp - 1] + " thousand");
    integer -= tmp * 1000;
    if (integer == 0)
        return result.toString();
    result.append(" ");
}
if (integer >= 100) {
    int tmp = integer / 100;
    result.append(group1[tmp - 1] + " hundred");
    integer -= tmp * 100;
    if (integer == 0)
        return result.toString();
    result.append(" and ");
}
if (integer >= 10 && integer <= 19){
    result.append(group2[integer - 10]);
    return result.toString();
}
if (integer >= 20){
    int tmp = integer / 10;
    result.append(group3[tmp - 2]);
    integer -= tmp * 10;
    if (integer == 0)
        return result.toString();
    result.append("- ");
}
```

```

        result.append(group1[integer - 1]);
        return result.toString();
    }
}

```

38. Listing A-36 presents the EVDump application that was called for in Chapter 7.

Listing A-36. Dumping All Environment Variables to Standard Output

```

public class EVDump {
    public static void main(String[] args) {
        System.out.println(System.getenv()); // System.out.println() calls
                                                // toString()
                                                // on its object argument and
                                                // outputs this
                                                // string
    }
}

```

39. Listing A-37 presents the revised CountingThreads application that was called for in Chapter 7.

Listing A-37. Counting via Daemon Threads

```

public class CountingThreads {
    public static void main(String[] args) {
        Runnable r = new Runnable(){
            @Override
            public void run(){
                String name = Thread.currentThread().getName();
                int count = 0;
                while (true)
                    System.out.println(name + ":" + count++);
            }
        };
    }
}

```

APPENDIX A SOLUTIONS TO EXERCISES

```
    Thread thdA = new Thread(r);
    thdA.setDaemon(true);
    Thread thdB = new Thread(r);
    thdB.setDaemon(true);
    thdA.start();
    thdB.start();
}
}
```

When you run this application, the two daemon threads start executing, and you will probably see some output. However, the application will end as soon as the default main thread leaves the `main()` method and dies.

40. Listing A-38 presents the `StopCountingThreads` application that was called for in Chapter 7.

Listing A-38. Stopping the Counting Threads When Return/Enter Is Pressed

```
import java.io.IOException;

public class StopCountingThreads {
    private static volatile boolean stopped = false;

    public static void main(String[] args) {
        Runnable r = new Runnable(){
            @Override
            public void run(){
                String name = Thread.currentThread().getName();
                int count = 0;
                while (!stopped)
                    System.out.println(name + ": " + count++);
            }
        };
        Thread thdA = new Thread(r);
        Thread thdB = new Thread(r);
        thdA.start();
        thdB.start();
    }
}
```

```
try { System.in.read(); } catch (IOException ioe) {}  
stopped = true;  
}  
}
```

Chapter 8: Exploring the Basic APIs, Part 2

1. Instances of the `Random` class generate sequences of random numbers by starting with a special 48-bit value that's known as a seed. This value is subsequently modified by a mathematical algorithm, which is known as a linear congruential generator.
2. Some of the capabilities offered by the Reflection API are letting applications dynamically load and learn about loaded classes and other reference types and letting applications instantiate classes, call methods, access fields, and perform other tasks reflectively.
3. Reflection shouldn't be used indiscriminately for several reasons. Application performance suffers because it takes longer to perform operations with reflection than without reflection. Also, reflection-oriented code can be harder to read, and the absence of compile-time type checking can result in runtime failures.
4. The class that's the entry point into the Reflection API is `java.lang.Class`.
5. The answer is false: not all of the Reflection API is contained in the `java.lang.reflect` package. For example, `Class` is located in the `java.lang` package.
6. The three ways to obtain a `Class` object are to invoke `Class`'s `forName()` method, to invoke `Object`'s `getClass()` method, and to use a class literal.
7. The answer is true: you can use class literals with primitive types.
8. You instantiate a dynamically loaded class by invoking `Class`'s `newInstance()` method.

APPENDIX A SOLUTIONS TO EXERCISES

9. You invoke Constructor's `Class[] <?> getParameterTypes()` method to obtain a constructor's parameter types.
10. Class's `Field getField(String name)` method throws `NoSuchFieldException` when it cannot locate the named field.
11. You determine if a method is declared to receive a variable number of arguments by invoking Method's `isVarArgs()` method on the Method object that represents the method.
12. The answer is true: you can reflectively make a private method accessible. You do this by invoking the `setAccessible()` method that each of Constructor, Field, and Method inherits from its `AccessibleObject` superclass.
13. The purpose of Package's `isSealed()` method is to indicate whether or not a package is sealed (all classes that are part of the package are archived in the same JAR file). This method returns true when the package is sealed.
14. The answer is true: `getPackage()` requires at least one class file to be loaded from the package before it returns a Package object describing that package.
15. You reflectively create and access a Java array by invoking the class methods declared in the `java.lang.reflect.Array` class.
16. The purpose of the StringTokenizer class is to provide access to a string's individual components.
17. The `java.util` package's Timer and TimerTask classes are the standard class library's convenient and simpler alternative to the Threads API for scheduling task execution.
execution time of the previous execution. When an execution is delayed for any reason (such as garbage collection), subsequent executions are also delayed.
18. Listing A-39 presents the Guess application that was called for in Chapter 8.

Listing A-39. Guessing Game

```
import java.util.Random;

public class Guess {
    public static void main(String[] args) throws java.io.IOException{
        Random r = new Random();
        int hiddenValue = 'a' + r.nextInt(26);

        while (true){
            int guess = 0;
            while (guess < 'a' || guess > 'z'){
                System.out.print("Guess between a and z inclusive: ");
                guess = System.in.read();
                System.out.println();

                // Flush carriage return or carriage return/newline combination
                // so that each character isn't automatically read during the
                // next System.in.read() method call.

                int x = 0;
                while (x != '\n')
                    x = System.in.read();
            }
            if (guess < hiddenValue)
                System.out.println("too low");
            else
                if (guess > hiddenValue)
                    System.out.println("too high");
                else {
                    System.out.println("you got it");
                    break;
                }
        }
    }
}
```

19. Listing A-40 presents the `Classify` application that was called for in Chapter 8.

Listing A-40. Classifying a Command-Line Argument as an Annotation Type, Enum, Interface, or Class

```
public class Classify {
    public static void main(String[] args) {
        if (args.length != 1){
            System.err.println("usage: java Classify pkgAndTypeName");
            return;
        }
        try{
            Class<?> clazz = Class.forName(args[0]);
            if (clazz.isAnnotation())
                System.out.println("Annotation");
            else if (clazz.isEnum())
                System.out.println("Enum");
            else if (clazz.isInterface())
                System.out.println("Interface");
            else
                System.out.println("Class");
        } catch (ClassNotFoundException cnfe) {
            System.err.println("could not locate " + args[0]);
        }
    }
}
```

Specify `java Classify java.lang.Override`, and you'll see `Annotation` as the output. Also, `java Classify java.math.RoundingMode` outputs `Enum`, `java Classify java.lang.Runnable` outputs `Interface`, and `java Classify java.lang.Class` outputs `Class`.

20. Listing A-41 presents the `Tokenize` application that was called for in Chapter 8.

Listing A-41. Extracting the Month, Day, Year, Hour, Minute, and Second Tokens from a Date String

```
import java.util.StringTokenizer;

public class Tokenize {
    public static void main(String[] args) {
        String date = "03-12-2014 03:05:20";
        StringTokenizer st = new StringTokenizer(date, "- :");
        System.out.println("Month = " + st.nextToken());
        System.out.println("Day = " + st.nextToken());
        System.out.println("Year = " + st.nextToken());
        System.out.println("Hour = " + st.nextToken());
        System.out.println("Minute = " + st.nextToken());
        System.out.println("Second = " + st.nextToken());
    }
}
```

21. Listing [A-42](#) presents the BackAndForth application that was called for in Chapter [8](#).

Listing A-42. Repeatedly Moving an Asterisk Back and Forth via a Timer

```
import java.util.Timer;
import java.util.TimerTask;

public class BackAndForth {
    public static enum Direction { FORWARDS, BACKWARDS }

    public static void main(String[] args) {

        TimerTask task = new TimerTask() {
            final static int MAXSTEPS = 20;
            Direction direction = Direction.FORWARDS;
            int steps = 0;
            @Override
            public void run() {
                switch (direction){
                    case FORWARDS :
```

```
        System.out.print("\b ");
        System.out.print("*");
        break;

    case BACKWARDS:
        System.out.print("\b ");
        System.out.print("\b\b*");
    }

    if (++steps == MAXSTEPS) {
        direction = (direction == Direction.FORWARDS)
            ? Direction.BACKWARDS
            : Direction.FORWARDS;
        steps = 0;
    }
}

};

Timer timer = new Timer();
timer.schedule(task, 0, 100);
}
}
```

Chapter 9: Exploring the Collections Framework

1. A collection is a group of objects that are stored in an instance of a class designed for this purpose.
2. The Collections Framework is a group of types that offers a standard architecture for representing and manipulating collections.
3. The Collections Framework largely consists of core interfaces, implementation classes, and utility classes.
4. A comparable is an object whose class implements the Comparable interface.

5. You would have a class implement the Comparable interface when you want objects to be compared according to their natural ordering.
6. A comparator is an object whose class implements the Comparator interface. Its purpose is to allow objects to be compared according to an order that's different from their natural ordering.
7. The answer is false: a collection uses a comparable (an object whose class implements the Comparable interface) to define the natural ordering of its elements.
8. The Iterable interface describes any object that can return its contained objects in some sequence.
9. The Collection interface represents a collection of objects that are known as elements.
10. A situation where Collection's add() method would throw an instance of the UnsupportedOperationException class is an attempt to add an element to an unmodifiable collection.
11. The purpose of the enhanced for loop statement is to simplify collection or array iteration.
12. The enhanced for loop statement is expressed as `for (type id: collection)` or `for (type id: array)` and reads “for each *type* object in *collection*, assign this object to *id* at the start of the loop iteration” or “for each *type* object in *array*, assign this object to *id* at the start of the loop iteration.”
13. The answer is true: the enhanced for loop works with arrays. For example, `int[] x = { 1, 2, 3 }; for (int i: x) System.out.println(i);` declares array x and outputs all of its int-based elements.
14. Autoboxing is the act of wrapping a primitive-type value in an object of a primitive type wrapper class whenever a primitive type is specified but a reference is required. This feature saves the developer from having to instantiate a wrapper class explicitly when storing the primitive-type value in a collection.

APPENDIX A SOLUTIONS TO EXERCISES

15. Unboxing is the act of unwrapping a primitive-type value from its wrapper object whenever a reference is specified but a primitive type is required. This feature saves the developer from having to call a method explicitly on the object (such as `intValue()`) to retrieve the wrapped value.
16. A list is an ordered collection, which is also known as a sequence. Elements can be stored in and accessed from specific locations via integer indexes.
17. A `ListIterator` instance uses a cursor to navigate through a list.
18. A view is a list that's backed by another list. Changes that are made to the view are reflected in this backing list.
19. You would use the `subList()` method to perform range-view operations over a collection in a compact manner. For example, `list.subList(fromIndex, toIndex).clear();` removes a range of elements from `list`, where the first element is located at `fromIndex` and the last element is located at `toIndex - 1`.
20. The `ArrayList` class provides a list implementation that's based on an internal array.
21. The `LinkedList` class provides a list implementation that's based on linked nodes.
22. The answer is false: `ArrayList` provides slower element insertions and deletions than `LinkedList`.
23. A set is a collection that contains no duplicate elements.
24. The `TreeSet` class provides a set implementation that's based on a tree data structure. As a result, elements are stored in sorted order.
25. The `HashSet` class provides a set implementation that's backed by a hashtable data structure.
26. The answer is true: to avoid duplicate elements in a hashset, your own classes must correctly override `equals()` and `hashCode()`.

27. The difference between `HashSet` and `LinkedHashSet` is that `LinkedHashSet` uses a linked list to store its elements, resulting in its iterator returning elements in the order in which they were inserted.
28. The `EnumSet` class provides a `Set` implementation that's based on a bitset.
29. A sorted set is a set that maintains its elements in ascending order, sorted according to their natural ordering or according to a comparator that's supplied when the sorted set is created. Furthermore, the set's implementation class must implement the `SortedSet` interface.
30. A navigable set is a sorted set that can be iterated over in descending order as well as ascending order and which can report closest matches for given search targets.
31. The answer is false: `HashSet` isn't an example of a sorted set. However, `TreeSet` is an example of a sorted set.
32. A sorted set's `add()` method would throw `ClassCastException` when you attempt to add an element to the sorted set because the element's class doesn't implement `Comparable`.
33. A queue is a collection in which elements are stored and retrieved in a specific order. Most queues are categorized as "first-in, first-out," "last-in, first-out," or priority.
34. The answer is true: `Queue`'s `element()` method throws `NoSuchElementException` when it's called on an empty queue.
35. The `PriorityQueue` class provides an implementation of a priority queue, which is a queue that orders its elements according to their natural ordering or by a comparator provided when the queue is instantiated.
36. A map is a group of key/value pairs (also known as entries).
37. The `TreeMap` class provides a map implementation that's based on a red-black tree. As a result, entries are stored in sorted order of their keys.

APPENDIX A SOLUTIONS TO EXERCISES

38. The `HashMap` class provides a map implementation that's based on a hashtable data structure.
39. A hashtable uses a hash function to map keys to integer values.
40. Continuing from the previous exercise, the resulting integer values are known as hash codes. They identify hashtable array elements, which are known as buckets or slots.
41. A hashtable's capacity refers to the number of buckets.
42. A hashtable's load factor refers to the ratio of the number of stored entries divided by the number of buckets.
43. The difference between `HashMap` and `LinkedHashMap` is that `LinkedHashMap` uses a linked list to store its entries, resulting in its iterator returning entries in the order in which they were inserted.
44. The `IdentityHashMap` class provides a Map implementation that uses reference equality (`==`) instead of object equality (`equals()`) when comparing keys and values.
45. The `EnumMap` class provides a Map implementation whose keys are the members of the same enum.
46. A sorted map is a map that maintains its entries in ascending order, sorted according to the keys' natural ordering or according to a comparator that's supplied when the sorted map is created. Furthermore, the map's implementation class must implement the `SortedMap` interface.
47. A navigable map is a sorted map that can be iterated over in descending order as well as ascending order and which can report closest matches for given search targets.
48. The answer is true: `TreeMap` is an example of a sorted map.
49. The purpose of the `Arrays` class's static `<T> List<T> asList(T... array)` method is to return a fixed-size list backed by the specified array. (Changes to the returned list "write through" to the array.)
50. The answer is false: binary search is faster than linear search.

51. You would use Collections' static <T> Set<T> synchronizedSet(Set<T> s) method to return a synchronized variation of a hashset.
52. The seven legacy collections-oriented types are Vector, Enumeration, Stack, Dictionary, Hashtable, Properties, and BitSet.
53. Listing A-43 presents the JavaQuiz application that was called for in Chapter 9.

Listing A-43. How Much Do You Know About Java? Take the Quiz and Find Out!

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class JavaQuiz {
    private static class QuizEntry {
        private String question;
        private String[] choices;
        private char answer;

        QuizEntry(String question, String[] choices, char answer){
            this.question = question;
            this.choices = choices;
            this.answer = answer;
        }

        String[] getChoices(){
            // Demonstrate returning a copy of the choices array
            // to prevent clients from directly manipulating
            // (and possibly screwing up) the internal
            // choices array.
            String[] temp = new String[choices.length];
            System.arraycopy(choices, 0, temp, 0, choices.length);
            return temp;
        }
    }
}
```

APPENDIX A SOLUTIONS TO EXERCISES

```
String getQuestion() {
    return question;
}

char getAnswer() {
    return answer;
}

}

static QuizEntry[] quizEntries = {
    new QuizEntry("What was Java's original name?",
        new String[] { "Oak", "Duke", "J", "None of the above" },
        'A'),
    new QuizEntry("Which of the following reserved words is also a
literal?", 
        new String[] { "for", "long", "true", "enum" },
        'C'),
    new QuizEntry("The conditional operator (?:) resembles which
statement?", 
        new String[] { "switch", "if-else", "if", "while" },
        'B')
};

public static void main(String[] args) {
    // Populate the quiz list.
    List<QuizEntry> quiz = new ArrayList<QuizEntry>();
    for (QuizEntry entry: quizEntries)
        quiz.add(entry);
    // Perform the quiz.
    System.out.println("Java Quiz");
    System.out.println("-----\n");
    Iterator<QuizEntry> iter = quiz.iterator();
    while (iter.hasNext()) {
        QuizEntry qe = iter.next();
        System.out.println(qe.getQuestion());
        String[] choices = qe.getChoices();
        for (int i = 0; i < choices.length; i++)
```

```
System.out.println(" " + (char) ('A' + i) + ": " + choices[i]);
int choice = -1;
while (choice < 'A' || choice > 'A' + choices.length) {
    System.out.print("Enter choice letter: ");
    Try {
        choice = System.in.read();
        // Remove trailing characters up to and including the newline
        // to avoid having these characters automatically returned in
        // subsequent System.in.read() method calls.
        while (System.in.read() != '\n');
        choice = Character.toUpperCase((char) choice);
    } catch (java.io.IOException ioe){
    }
}
if (choice == qe.getAnswer())
    System.out.println("You are correct!\n");
else
    System.out.println("You are not correct!\n");
}
}
```

54. `(int) (f ^ (f >>> 32))` is used instead of `(int) (f ^ (f >> 32))` in the hash code generation algorithm because `>>>` always shifts a 0 to the right, which doesn't affect the hash code, whereas `>>` shifts a 0 or a 1 to the right (whatever value is in the sign bit), which affects the hash code when a 1 is shifted.
55. Listing A-44 presents the `FrequencyDemo` application that was called for in Chapter 9.

Listing A-44. Reporting the Frequency of Last Command-Line Argument Occurrences in the Previous Command-Line Arguments

```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
```

```

public class FrequencyDemo {
    public static void main(String[] args) {
        List<String> listOfArgs = new LinkedList<String>();
        String lastArg = (args.length == 0) ? null : args[args.length - 1];
        for (int i = 0; i < args.length - 1; i++)
            listOfArgs.add(args[i]);
        System.out.println("Number of occurrences of " + lastArg + " = " +
                           Collections.frequency(listOfArgs, lastArg));
    }
}

```

Chapter 10: Functional Programming

1. Right. This is one of the main characteristics of functional programming inside an object-oriented environment.
2. Wrong. Although using streams without functional constructs doesn't make much sense, a stream per se is not a functional subconcept.
3. Wrong. The other way round is true.
4. A consumer is a function with one input and no output.
5. As supplier is a function with no input and one output.
6. The solution is shown in Listing A-45.

Listing A-45. Adding 1.0 to Any Double

```

d -> d + 1.0          // This is the shortest form.
(d) -> d + 1.0        // ...and...
(Double d) -> d + 1.0 // ...and...
d -> { return d + 1.0; } // ...are possible as well.

```

7. The solution is shown in Listing A-46.

Listing A-46. A Currency Value Formatter

```
Function<BigDecimal, String> f1 = bd ->
    NumberFormat.getCurrencyInstance(Locale.US).format(bd);
```

8. Wrong. The int array gets seen as a single stream object.
9. Stream.iterate(100L, l -> l + 1) creates a stream of Long objects. The second possible answer is LongStream.iterate(100, l -> l + 1), which creates a LongStream.
10. IntStream.iterate(0, i -> i + 1).limit(100) is a correct answer, but more elegant is IntStream.rangeClosed(0, 99). If you really need a stream of integer objects, you can also use Stream.iterate(0, i -> i + 1).limit(100).
11. The solution is IntStream.iterate(1, i->i+1).map(i -> i*i).
12. The solution is new HashSet<>(Arrays.asList(1,6,3,7)).stream().
13. The solution is shown in Listing A-47.

Listing A-47. Building an IntStream

```
new HashSet<>(Arrays.asList(1,6,3,7))
    .stream()
    .mapToInt(i -> i)
```

14. The solution is shown in Listing A-48. The boxed() function converts a stream of native data type to an object stream.

Listing A-48. Back to Integer Objects

```
new HashSet<>(Arrays.asList(1,6,3,7))
    .stream()
    .mapToInt(i -> i)
    .boxed()
```

APPENDIX A SOLUTIONS TO EXERCISES

15. The stream's `filter()` function allows us to extract certain members of a stream. So we can write `stringStream.filter(s -> s.length() >= 3);`.
16. Just add `.collect(Collectors.toList());`.
17. The solution is shown in Listing A-49.

Listing A-49. Limiting and Outputting

```
IntStream.  
    iterate(1, i->i+1).  
    map(i -> i*i).  
    limit(10).  
    forEach(System.out::println);
```

18. $1 * 6 * 7 * 3$ can be calculated via

```
Stream.of(1,6,7,3).reduce(1, (i,j) -> i * j).
```

19. The solution is shown in Listing A-50.

Listing A-50. Building a Deque from a Stream

```
Stream.of(1,6,7,3).collect(  
    ArrayDeque::new,  
    (bas,inj) -> { bas.add(inj); },  
    (bas1,bas2) -> { bas1.addAll(bas2); } );
```

20. The solution is shown in Listing A-51.

Listing A-51. Hash Codes from a Stream

```
Arrays.asList("Hello", "World", "A").  
    stream().  
    map(Object::hashCode).  
    collect(Collectors.toList());
```

21. The solution is shown in Listing A-52.

Listing A-52. An Infinite Stream

```
DoubleStream dstr = IntStream.iterate(1, i -> i + 2).
    mapToDouble( i -> (((i+1)/2)%2==1) ? 1.0/i : -1.0/i );

22. Write double pi1 = dstr.limit(1_000_000).sum() * 4;
    System.out.println("pi ~ " + pi1);

23. The solution is shown in Listing A-53.
```

Listing A-53. Metering a Pi Calculation

```
DoubleStream dstr = IntStream.iterate(1, i -> i + 2).
    mapToDouble( i -> (((i+1)/2)%2==1) ? 1.0/i : -1.0/i );
long t1 = System.currentTimeMillis();
double pi1 = dstr.limit(1_000_000).sum() * 4;
long t2 = System.currentTimeMillis();
System.out.println("pi ~ " + pi1);
System.out.println("Took " + (t2-t1) + "ms");
// On our laptop, this took 36ms.
```

Chapter 11: Exploring the Concurrency Utilities

- Concurrency Utilities is a framework of classes and interfaces that overcome problems with the Threads API. Specifically, low-level concurrency primitives such as synchronized and wait()/notify() are often hard to use correctly; too much reliance on the synchronized primitive can lead to performance issues, which affect an application's scalability; and higher-level constructs such as thread pools and semaphores aren't included with Java's low-level threading capabilities.
- The packages in which Concurrency Utilities types are stored are java.util.concurrent, java.util.concurrent.atomic, and java.util.concurrent.locks.

APPENDIX A SOLUTIONS TO EXERCISES

3. A task is an object whose class implements the `Runnable` interface (a runnable task) or the `Callable` interface (a callable task).
4. An executor is an object whose class directly or indirectly implements the `Executor` interface, which decouples task submission from task-execution mechanics.
5. The `Executor` interface focuses exclusively on `Runnable`, which means that there's no convenient way for a runnable task to return a value to its caller (because `Runnable`'s `run()` method doesn't return a value); `Executor` doesn't provide a way to track the progress of executing runnable tasks, cancel an executing runnable task, or determine when the runnable task finishes execution; `Executor` cannot execute a collection of runnable tasks; and `Executor` doesn't provide a way for an application to shut down an executor (much less to shut down an executor properly).
6. The differences existing between `Runnable`'s `run()` method and `Callable`'s `call()` method are as follows: `run()` cannot return a value, whereas `call()` can return a value; and `run()` cannot throw checked exceptions, whereas `call()` can throw checked exceptions.
7. A future is an object whose class implements the `Future` interface. It represents an asynchronous computation and provides methods for canceling a task, for returning a task's value, and for determining whether or not the task has finished.
8. The `Executors` class's `newFixedThreadPool()` method creates a thread pool that reuses a fixed number of threads operating off of a shared unbounded queue. At most, `nThreads` threads are actively processing tasks. If additional tasks are submitted when all threads are active, they wait in the queue for an available thread. If any thread terminates because of a failure during execution before the executor shuts down, a new thread will take its place when needed to execute subsequent tasks. The threads in the pool will exist until the executor is explicitly shut down.

9. A synchronizer is a class that facilitates a common form of synchronization.
10. The concurrency-oriented extensions to the Collections Framework provided by the Concurrency Utilities are `ArrayBlockingQueue`, `BlockingDeque`, `BlockingQueue`, `ConcurrentHashMap`, `ConcurrentMap`, `ConcurrentNavigableMap`, `ConcurrentLinkedQueue`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `DelayQueue`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, and `SynchronousQueue`.
11. A lock is an instance of a class that implements the `Lock` interface, which provides more extensive locking operations than can be achieved via the synchronized reserved word. `Lock` also supports a wait/notification mechanism through associated `Condition` objects.
12. The biggest advantage that `Lock` objects hold over the implicit locks that are obtained when threads enter critical sections (controlled via the synchronized reserved word) is their ability to back out of an attempt to acquire a lock.
13. You obtain a `Condition` instance for use with a particular `Lock` instance by invoking `Lock`'s `Condition newCondition()` method.
14. An atomic variable is an instance of a class that encapsulates a single variable and supports lock-free, thread-safe operations on that variable, for example, `AtomicInteger`.
15. The `AtomicIntegerArray` class describes an `int` array whose elements may be updated atomically.
16. The atomic variable equivalent of `int total = ++counter;` is as follows:

```
AtomicInteger counter = new AtomicInteger(0);
int total = counter.incrementAndGet();
```

The atomic variable equivalent of `int total = counter--;` is as follows:

```
AtomicInteger counter = new AtomicInteger(0);  
int total = counter.getAndDecrement();
```

Chapter 12: Performing Classic I/O

1. The purpose of the `File` class is to offer access to the underlying platform's available filesystem(s).
2. Instances of the `File` class contain the pathnames of files and directories that may or may not exist in their filesystems.
3. `File`'s `listRoots()` method returns an array of `File` objects denoting the root directories (roots) of available filesystems.
4. A path is a hierarchy of directories that must be traversed to locate a file or a directory. A pathname is a string representation of a path; a platform-dependent separator character (such as the Windows backslash [\] character) appears between consecutive names.
5. The difference between an absolute pathname and a relative pathname is as follows: an absolute pathname is a pathname that starts with the root directory symbol, whereas a relative pathname is a pathname that doesn't start with the root directory symbol (it's interpreted via information taken from some other pathname).
6. You obtain the current user (also known as working) directory by specifying `System.getProperty("user.dir")`.
7. A parent pathname is a string that consists of all pathname components except for the last name.
8. Normalize means to replace separator characters with the default name-separator character so that the pathname is compliant with the underlying filesystem.

9. You obtain the default name-separator character by accessing `File`'s `separator` and `separatorChar` class fields. The first field stores the character as a `char` and the second field stores it as a `String`.
10. A canonical pathname is a pathname that's absolute and unique, and it is formatted the same way every time.
11. The difference between `File`'s `getParent()` and `getName()` methods is that `getParent()` returns the parent pathname and `getName()` returns the last name in the pathname's name sequence.
12. The answer is false: `File`'s `exists()` method determines whether or not a file or directory exists.
13. A normal file is a file that's not a directory and that satisfies other platform-dependent criteria: it's not a symbolic link or named pipe, for example. Any nondirectory file created by a Java application is guaranteed to be a normal file.
14. `File`'s `lastModified()` method returns the time that the file denoted by this `File` object's pathname was last modified or 0 when the file doesn't exist or an I/O error occurred during this method call. The returned value is measured in milliseconds since the Unix epoch (00:00:00 GMT, January 1, 1970).
15. The answer is true: `File`'s `list()` method returns an array of `Strings` where each entry is a file name rather than a complete path.
16. The difference between the `FilenameFilter` and `FileFilter` interfaces is as follows: `FilenameFilter` declares a single boolean `accept(File dir, String name)` method, whereas `FileFilter` declares a single boolean `accept(String pathname)` method. Either method accomplishes the same task of accepting (by returning true) or rejecting (by returning false) the inclusion of the file or directory identified by the argument(s) in a directory listing.

APPENDIX A SOLUTIONS TO EXERCISES

17. The answer is false: `File`'s `createNewFile()` method checks for file existence and creates the file when it doesn't exist in a single operation that's atomic with respect to all other filesystem activities that might affect the file.
18. The default temporary directory where `File`'s `createTempFile(String, String)` method creates temporary files can be located by reading the `java.io.tmpdir` system property.
19. You ensure that a temporary file is removed when the virtual machine ends normally (it doesn't crash and the power isn't lost) by registering the temporary file for deletion through a call to `File`'s `deleteOnExit()` method.
20. You would accurately compare two `File` objects by first calling `File`'s `getCanonicalFile()` method on each `File` object and then comparing the returned `File` objects.
21. The purpose of the `RandomAccessFile` class is to create and/or open files for random access in which a mixture of write and read operations can occur until the file is closed.
22. The purpose of the "rwd" and "rws" mode arguments is to ensure that any writes to a file located on a local storage device are written to the device, which guarantees that critical data isn't lost when the system crashes. No guarantee is made when the file doesn't reside on a local device.
23. A file pointer is a cursor that identifies the location of the next byte to write or read. When an existing file is opened, the file pointer is set to its first byte at offset 0. The file pointer is also set to 0 when the file is created.
24. The answer is false: when you call `RandomAccessFile`'s `seek(long)` method to set the file pointer's value, and when this value is greater than the length of the file, the file's length doesn't change. The file length will only change by writing after the offset has been set beyond the end of the file.

25. A stream is an ordered sequence of bytes of arbitrary length. Bytes flow over an output stream from an application to a destination, and flow over an input stream from a source to an application (do not mix up with streams from the Streams API).
26. The purpose of `OutputStream`'s `flush()` method is to write any buffered output bytes to the destination. If the intended destination of this output stream is an abstraction provided by the underlying platform (such as a file), flushing the stream only guarantees that bytes previously written to the stream are passed to the underlying platform for writing; it doesn't guarantee that they're actually written to a physical device such as a disk drive.
27. The answer is true: `OutputStream`'s `close()` method automatically flushes the output stream. If an application ends before `close()` is called, the output stream is automatically closed and its data is flushed.
28. The purpose of `InputStream`'s `mark(int)` and `reset()` methods is to reread a portion of a stream. `mark(int)` marks the current position in this input stream. A subsequent call to `reset()` repositions this stream to the last marked position so that subsequent read operations reread the same bytes. Don't forget to call `markSupported()` to find out if the subclass supports `mark()` and `reset()`.
29. You would access a copy of a `ByteArrayOutputStream` instance's internal byte array by calling `ByteArrayOutputStream`'s `toByteArray()` method.
30. The answer is false: `FileOutputStream` and `FileInputStream` don't provide internal buffers to improve the performance of write and read operations.
31. You would use `PipedOutputStream` and `PipedInputStream` to communicate data between a pair of executing threads.
32. A filter stream is a stream that buffers, compresses/uncompresses, encrypts/decrypts, or otherwise manipulates an input stream's byte sequence before it reaches its destination.

APPENDIX A SOLUTIONS TO EXERCISES

33. Two streams are chained together when a stream instance is passed to another stream class's constructor.
34. You improve the performance of a file output stream by chaining a `BufferedOutputStream` instance to a `FileOutputStream` instance and calling the `BufferedOutputStream` instance's `write()` methods so that data is buffered before flowing to the file output stream. You improve the performance of a file input stream by chaining a `BufferedInputStream` instance to a `FileInputStream` instance so that data flowing from a file input stream is buffered before being returned from the `BufferedInputStream` instance by calling this instance's `read()` methods.
35. `DataOutputStream` and `DataInputStream` support `FileOutputStream` and `FileInputStream` by providing methods to write and read primitive-type values and strings in a platform-independent way. In contrast, `FileOutputStream` and `FileInputStream` provide methods for writing/reading bytes and arrays of bytes only.
36. Object serialization is a virtual machine mechanism for serializing object state into a stream of bytes. Its deserialization counterpart is a virtual machine mechanism for deserializing this state from a byte stream.
37. The purpose of the `Serializable` interface is to tell the virtual machine that it's okay to serialize objects of the implementing class.
38. When the serialization mechanism encounters an object whose class doesn't implement `Serializable`, it throws an instance of the `NotSerializableException` class.
39. The three stated reasons for Java not supporting unlimited serialization are as follows: security, performance, and objects not amenable to serialization.
40. You initiate serialization by creating an `ObjectOutputStream` instance and calling its `writeObject()` method. You initiate deserialization by creating an `ObjectInputStream` instance and calling its `readObject()` method.

41. The answer is false: class fields are not automatically serialized.
42. The purpose of the `transient` reserved word is to mark instance fields that don't participate in default serialization and default deserialization.
43. The deserialization mechanism causes `readObject()` to throw an instance of the `InvalidClassException` class when it attempts to deserialize an object whose class has changed.
44. The deserialization mechanism detects that a serialized object's class has changed as follows: every serialized object has an identifier, and the deserialization mechanism compares the identifier of the object being serialized with the serialized identifier of its class (all serializable classes are automatically given unique identifiers unless they explicitly specify their own identifiers) and causes `InvalidClassException` to be thrown when it detects a mismatch.
45. You can add an instance field to a class and avoid trouble when deserializing an object that was serialized before the instance field was added by introducing a `long serialVersionUID = long integer value;` declaration into the class. The *long integer value* must be unique, and it is known as a stream unique identifier (SUID).
46. You customize the default serialization and deserialization mechanisms without using externalization by declaring `private void writeObject(ObjectOutputStream)` and `void readObject(ObjectInputStream)` methods in the class.
47. You tell the serialization and deserialization mechanisms to serialize or deserialize the object's normal state before serializing or deserializing additional data items by first calling `ObjectOutputStream's defaultWriteObject()` method in `writeObject(ObjectOutputStream)` and by first calling `ObjectInputStream's defaultReadObject()` method in `readObject(ObjectInputStream)`.

APPENDIX A SOLUTIONS TO EXERCISES

48. The difference between `PrintStream`'s `print()` and `println()` methods is that the `print()` methods don't append a line terminator to their output, whereas the `println()` methods append a line terminator.
49. `PrintStream`'s noargument `void println()` method outputs the `line.separator` system property's value to ensure that lines are terminated in a portable manner (such as a carriage return followed by a newline/line feed on Windows, or only a newline/line feed on Unix/Linux).
50. Java's stream classes are not good at streaming characters because bytes and characters are two different things: a byte represents an 8-bit data item and a character represents a 16-bit data item. Also, byte streams have no knowledge of character sets and their character encodings.
51. Java provides writer and reader classes as the preferred alternative to stream classes when it comes to character I/O.
52. The purpose of the `OutputStreamWriter` class is to serve as a bridge between an incoming sequence of characters and an outgoing stream of bytes. Characters written to this writer are encoded into bytes according to the default or specified character encoding. The purpose of the `InputStreamReader` class is to serve as a bridge between an incoming stream of bytes and an outgoing sequence of characters. Characters read from this reader are decoded from bytes according to the default or specified character encoding.
53. You identify the default character encoding by reading the value of the `file.encoding` system property.
54. The purpose of the `FileWriter` class is to connect conveniently to the underlying file output stream using the default character encoding. The purpose of the `FileReader` class is to connect conveniently to the underlying file input stream using the default character encoding.

55. Listing [A-54](#) presents the Touch application that was called for in Chapter [12](#).

Listing A-54. Setting a File or Directory's Timestamp to the Current Time

```
import java.io.File;
import java.util.Date;

public class Touch {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage: java Touch pathname");
            return;
        }
        new File(args[0]).setLastModified(new Date().getTime());
    }
}
```

56. Listing [A-55](#) presents the Copy application that was called for in Chapter [12](#).

Listing A-55. Copying a Source File to a Destination File with Buffered I/O

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Copy {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: java Copy srcfile dstfile");
            return;
        }
    }
}
```

APPENDIX A SOLUTIONS TO EXERCISES

```
BufferedInputStream bis = null;
BufferedOutputStream bos = null;
try {
    FileInputStream fis = new FileInputStream(args[0]);
    bis = new BufferedInputStream(fis);
    FileOutputStream fos = new FileOutputStream(args[1]);
    bos = new BufferedOutputStream(fos);
    int b; // We chose b instead of byte because byte is a reserved word.
    while ((b = bis.read()) != -1)
        bos.write(b);
} catch (FileNotFoundException fnfe) {
    System.err.println(args[0] + " could not be opened for input, or " +
                       args[1] + " could not be created for output");
} catch (IOException ioe){
    System.err.println("I/O error: " + ioe.getMessage());
} finally {
    if (bis != null)
        try {
            bis.close();
        } catch (IOException ioe){
        }

    if (bos != null)
        try {
            bos.close();
        } catch (IOException ioe){
        }
    }
}
```

57. Listing A-56 presents the Split application that was called for in Chapter 12.

Listing A-56. Splitting a Large File into Numerous Smaller Part Files

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class Split {
    private static final int FILESIZE = 1400000;
    private static byte[] buffer = new byte[FILESIZE];

    public static void main(String[] args){
        if (args.length != 1) {
            System.err.println("usage: java Split pathname");
            return;
        }
        File file = new File(args[0]);
        long length = file.length();
        int nWholeParts = (int) (length / FILESIZE);
        int remainder = (int) (length % FILESIZE);
        System.out.printf("Splitting %s into %d parts%n", args[0],
                           (remainder == 0) ? nWholeParts : nWholeParts + 1);
        BufferedInputStream bis = null;
        BufferedOutputStream bos = null;
        try {
            FileInputStream fis = new FileInputStream(args[0]);
            bis = new BufferedInputStream(fis);
            for (int i = 0; i < nWholeParts; i++){
                bis.read(buffer);
                System.out.println("Writing part " + i);
                FileOutputStream fos =
                    new FileOutputStream("part" + i);
                bos = new BufferedOutputStream(fos);
                bos.write(buffer);
                bos.close();
            }
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            if (bis != null)
                bis.close();
            if (bos != null)
                bos.close();
        }
    }
}
```

APPENDIX A SOLUTIONS TO EXERCISES

```
        bos = null;
    }
    if (remainder != 0) {
        int br = bis.read(buffer);
        if (br != remainder) {
            System.err.println("Last part mismatch: expected " +
                remainder + " bytes");
            System.exit(0);
        }
        System.out.println("Writing part " + nWholeParts);
        FileOutputStream fos = new FileOutputStream("part" +
            nWholeParts);
        bos = new BufferedOutputStream(fos);
        bos.write(buffer, 0, remainder);
    }
} catch (IOException ioe) {
    ioe.printStackTrace();
} finally {
    if (bis != null)
        try {
            bis.close();
        } catch (IOException ioe) {
        }
    if (bos != null)
        try {
            bos.close();
        } catch (IOException ioe) {
        }
    }
}
```

Chapter 13: Accessing Networks

1. A network is a group of interconnected nodes that can be shared among the network's users.
2. An intranet is a network located within an organization, and an internet is a network connecting organizations to each other.
3. Intranets and internets often use TCP/IP to communicate between nodes. Transmission Control Protocol (TCP) is a connection-oriented protocol, User Datagram Protocol (UDP) is a connectionless protocol, and Internet Protocol (IP) is the basic protocol over which TCP and UDP perform their tasks.
4. A host is a computer-based TCP/IP node.
5. A socket is an endpoint in a communications link between two processes.
6. A socket is identified by an IP address that identifies the host and by a port number that identifies the process running on that host.
7. An IP address is a 32-bit or 128-bit unsigned integer that uniquely identifies a network host or some other network node.
8. A packet is an addressable message chunk. Packets are often referred to as IP datagrams.
9. A socket address is comprised of an IP address and a port number.
10. The `InetAddress` subclasses that are used to represent IPv4 and IPv6 addresses are `Inet4Address` and `Inet6Address`.
11. The loopback interface is a software-based network interface where outgoing data loops back as incoming data.
12. The local host is represented by hostname `localhost` or by an IP address that's commonly expressed as `127.0.0.1` (IPv4) or `::1` (IPv6).
13. Sockets based on the `Socket` class are commonly referred to as stream sockets because `Socket` is associated with the `InputStream` and `OutputStream` classes.

APPENDIX A SOLUTIONS TO EXERCISES

14. In the context of a `Socket` instance, binding makes a client socket address available to a server socket so that a server process can communicate with the client process via the server socket.
15. A proxy is a host that sits between an intranet and the Internet for security purposes. Java represents proxy settings via instances of the `java.net.Proxy` class.
16. The answer is false: the `ServerSocket()` constructor creates an unbound server socket.
17. The difference between the `DatagramSocket` and `MulticastSocket` classes is as follows: `DatagramSocket` lets you perform UDP-based communications between a pair of hosts, whereas `MulticastSocket` lets you perform UDP-based communications between many hosts.
18. The difference between unicasting and multicasting is as follows: unicasting is the act of a server sending a message to a single client, whereas multicasting is the act of a server sending a message to multiple clients.
19. A URL is a character string that specifies where a resource (such as a web page) is located on a TCP-/IP-based network (such as the Internet). Also, it provides the means to retrieve that resource.
20. A URN is a character string that names a resource and doesn't provide a way to access that resource (the resource might not be available).
21. The answer is true: URLs and URNs are also URIs.
22. The equivalent of `openStream()` is to execute `openConnection().getInputStream()`.
23. The answer is false: you don't need to invoke `URLConnection`'s `void setDoInput(boolean doInput)` method with `true` as the argument before you can input content from a web resource. The default setting is `true`.
24. When it encounters a space character, `URLEncoder` converts it to a plus sign.

25. The purpose of the `URI` class is to represent names (URNs) and resources (URLs). Also, it provides normalization, resolution, and relativization operations; the resulting `URI` can be converted into a URL as long as it represents a resource.
26. The `NetworkInterface` class represents a network interface as a name and a list of IP addresses assigned to this interface. Furthermore, it's used to identify the local interface on which a multicast group is joined.
27. A MAC address is an array of bytes containing a network interface's hardware address.
28. MTU stands for maximum transmission unit. This size represents the maximum length of a message that can fit into an IP datagram without needing to fragment the message into multiple IP datagrams.
29. The answer is false: `NetworkInterface`'s `getName()` method returns a network interface's name (such as `eth0` or `lo`), not a human-readable display name.
30. `InterfaceAddress`'s `getNetworkPrefixLength()` method returns the subnet mask under IPv4.
31. HTTP cookie (cookie for short) is a state object.
32. It's preferable to store cookies on the client rather than on the server because of the potential for millions of cookies (depending on a website's popularity).
33. The four `java.net` types that are used to work with cookies are `CookieHandler`, `CookieManager`, `CookiePolicy`, and `CookieStore`.
34. Listing A-57 presents the enhanced `EchoClient` application that was called for in Chapter 13.

Listing A-57. Echoing Data to and Receiving It Back from a Server and Explicitly Closing the Socket

```
import java.io.BufferedReader;
import java.io.InputStream;
```

APPENDIX A SOLUTIONS TO EXERCISES

```
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

import java.net.Socket;
import java.net.UnknownHostException;

public class EchoClient {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("usage : java EchoClient message");
            System.err.println("example: java EchoClient \"This is a
test.\"");
            return;
        }
        Socket socket = null;
        try {
            socket = new Socket("localhost", 9999);
            OutputStream os = socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(os);
            PrintWriter pw = new PrintWriter(osw);
            pw.println(args[0]);
            pw.flush();
            InputStream is = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            System.out.println(br.readLine());
        } catch (UnknownHostException uhe) {
            System.err.println("unknown host: " + uhe.getMessage());
        } catch (IOException ioe) {
            System.err.println("I/O error: " + ioe.getMessage());
        }
    }
}
```

```
        finally {
            if (socket != null)
                try {
                    socket.close();
                } catch (IOException ioe){
                }
        }
    }
}
```

35. Listing [A-58](#) presents the enhanced EchoServer application that was called for in Chapter [13](#).

Listing A-58. Receiving Data from and Echoing It Back to a Client and Explicitly Closing the Socket After a kill File Appears

```
import java.io.BufferedReader;
import java.io.File;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer {
    public static void main(String[] args) {
        System.out.println("Starting echo server...");
        ServerSocket ss = null;
        try {
            ss = new ServerSocket(9999);
            File file = new File("kill");
            while (!file.exists()) {
                Socket s = ss.accept(); // waiting for client request
```

APPENDIX A SOLUTIONS TO EXERCISES

```
try{
    InputStream is = s.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    String msg = br.readLine();
    System.out.println(msg);
    OutputStream os = s.getOutputStream();
    OutputStreamWriter osw = new OutputStreamWriter(os);
    PrintWriter pw = new PrintWriter(osw);
    pw.println(msg);
    pw.flush();
}catch (IOException ioe){
    System.err.println("I/O error: " + ioe.getMessage());
}finally{
    try {
        s.close();
    } catch (IOException ioe) {
    }
}
}catch (IOException ioe){
    System.err.println("I/O error: " + ioe.getMessage());
} finally {
    if (ss != null)
        try {
            ss.close();
        }catch (IOException ioe){
        }
}
}
```

Chapter 14: Migrating to New I/O

1. New I/O is a more powerful I/O architecture that supports memory-mapped file I/O, readiness selection, file locking, and more. This architecture largely consists of buffers, channels, selectors, regular expressions, and charsets, but it also could be considered to include a `printf`-style formatting facility.
2. A buffer is an object that stores a fixed amount of data to be sent to or received from an I/O service (a means for performing input/output). It sits between an application and a channel that writes the buffered data to the service or reads the data from the service and deposits it into the buffer.
3. A buffer's four properties are capacity, limit, position, and mark.
4. When you invoke Buffer's `array()` method on a buffer backed by a read-only array, this method throws `ReadOnlyBufferException`.
5. When you invoke Buffer's `flip()` method on a buffer, the limit is set to the current position and then the position is set to zero. When the mark is defined, it is discarded. The buffer is now ready to be drained.
6. When you invoke Buffer's `reset()` method on a buffer where a mark has not been set, this method throws `InvalidMarkException`.
7. The answer is false: buffers are not thread-safe.
8. You create a byte buffer by invoking one of its `allocate()`, `allocateDirect()`, or `wrap()` class methods.
9. A view buffer is a buffer that manages another buffer's data.
10. A view buffer is created by calling a Buffer subclass's `duplicate()` method.
11. You create a read-only view buffer by calling a Buffer subclass method such as `ByteBuffer asReadOnlyBuffer()` or `CharBuffer asReadOnlyBuffer()`.

APPENDIX A SOLUTIONS TO EXERCISES

12. ByteBuffer's methods for storing a single byte in a byte buffer are `ByteBuffer put(int index, byte b)` and `ByteBuffer put(byte b)`. ByteBuffer's methods for fetching a single byte from a byte buffer are `byte get(int index)` and `byte get()`.
13. Attempting to use the relative `put()` method or the relative `get()` method when the current position is greater than or equal to the limit causes `BufferOverflowException` or `BufferUnderflowException` to occur.
14. The answer is false: calling `flip()` twice doesn't return you to the original state. Instead, the buffer has a zero size.
15. The difference between Buffer's `clear()` and `reset()` methods is as follows: the `clear()` method marks a buffer as empty, whereas `reset()` changes the buffer's current position to the previously set mark or throws `InvalidMarkException` when there's no previously set mark.
16. ByteBuffer's `compact()` method compacts a buffer by copying all bytes between the current position and the limit to the beginning of the buffer. The byte at index `p = position()` is copied to index 0, the byte at index `p + 1` is copied to index 1, and so on until the byte at index `limit() - 1` is copied to index `n = limit() - 1 - p`. The buffer's current position is then set to `n + 1` and its limit is set to its capacity. The mark, when defined, is discarded.
17. The purpose of the `ByteOrder` class is to help you deal with byte-order issues when writing/reading multibyte values to/from a multibyte buffer.
18. A direct byte buffer is a byte buffer that interacts with channels and native code to perform I/O. The direct byte buffer attempts to store byte elements in a memory area that a channel uses to perform direct (raw) access via native code that tells the operating system to drain or fill the memory area directly.
19. You obtain a direct byte buffer by invoking ByteBuffer's `allocateDirect()` method.

20. A channel is an object that represents an open connection to a hardware device, a file, a network socket, an application component, or another entity that's capable of performing write, read, and other I/O operations. Channels efficiently transfer data between byte buffers and I/O service sources or destinations.
21. The capabilities that the Channel interface provides are closing a channel (via the close() method) and determining whether or not a channel is open (via the isOpen() method).
22. The answer is true: a channel that implements InterruptibleChannel is asynchronously closeable.
23. A file channel is a channel to an underlying file.
24. An exclusive lock is a lock that prevents other file locks from being used within the region governed by the exclusive lock. In contrast, a shared lock is a lock that may apply to a region governed by other shared locks.
25. The fundamental difference between FileChannel's lock() and tryLock() methods is that the lock() methods can block and the tryLock() methods never block.
26. The FileLock lock() method throws OverlappingFileLockException when either a lock is already held that overlaps this lock request or another thread is waiting to acquire a lock that will overlap with this request.
27. The pattern that you should adopt to ensure that an acquired file lock is always released follows:

```
FileLock lock = fileChannel.lock();
try {
    // interact with the file channel
} catch (IOException ioe) {
    // handle the exception
} finally {
    lock.release();
}
```

APPENDIX A SOLUTIONS TO EXERCISES

28. `FileChannel` provides the `MappedByteBuffer map(FileChannel, MapMode mode, long position, long size)` method for mapping a region of a file into memory.
29. The answer is true: socket channels are selectable and can function in nonblocking mode.
30. The answer is false: datagram channels are thread-safe.
31. Socket channels support nonblocking mode because the blocking nature of sockets created from Java's socket classes is a serious limitation to a network-oriented Java application's scalability.
32. You would obtain a socket channel's associated socket by invoking its `socket()` method.
33. You obtain a server socket channel by invoking `ServerSocketChannel's open()` class method.
34. A selector is an object created from a subclass of the abstract `Selector` class. It maintains a set of channels, which it examines to determine which of them are ready for reading, writing, completing a connection sequence, accepting another connection, or some combination of these tasks. The actual work is delegated to the operating system via a POSIX `select()` or similar system call.
35. A regular expression (also known as a regex or regexp) is a string-based pattern that represents the set of strings that match this pattern.
36. Instances of the `Pattern` class represent patterns via compiled regexes. Regexes are compiled for performance reasons; pattern matching via compiled regexes is much faster than if the regexes were not compiled.
37. `Pattern's compile()` methods throw `PatternSyntaxException` when they discover illegal syntax in their regular expression arguments.
38. Instances of the `Matcher` class attempt to match compiled regexes against input text.

39. The difference between `Matcher's matches()` and `lookingAt()` methods is that unlike `matches()`, `lookingAt()` doesn't require the entire region to be matched.
40. A character class is a set of characters appearing between [and].
41. There are six kinds of character classes: simple, negation, range, union, intersection, and subtraction.
42. A capturing group saves a match's characters for later recall during pattern matching.
43. A zero-length match is a match of zero length in which the start and end indexes are equal.
44. A quantifier is a numeric value implicitly or explicitly bound to a pattern. Quantifiers are categorized as greedy, reluctant, or possessive.
45. The difference between a greedy quantifier and a reluctant quantifier is that a greedy quantifier attempts to find the longest match, whereas a reluctant quantifier attempts to find the shortest match.
46. Possessive and greedy quantifiers differ in that a possessive quantifier only makes one attempt to find the longest match, whereas a greedy quantifier can make multiple attempts.
47. The two main classes that contribute to the NIO `printf`-style formatting facility are `Formatter` and `Scanner`.
48. The `%n` format specifier outputs a platform-specific line separator.
49. Listing A-59 presents the enhanced Copy application that was called for in Chapter 14.

Listing A-59. Copying a File via a Byte Buffer and a File Channel

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

APPENDIX A SOLUTIONS TO EXERCISES

```
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
public class Copy {
    public static void main(String[] args) {
        if (args.length != 2){
            System.err.println("usage: java Copy srcfile dstfile");
            return;
        }
        FileChannel fcSrc = null;
        FileChannel fcDest = null;
        try {
            FileInputStream fis = new FileInputStream(args[0]);
            fcSrc = fis.getChannel();
            FileOutputStream fos = new FileOutputStream(args[1]);
            fcDest = fos.getChannel();
            ByteBuffer buffer = ByteBuffer.allocateDirect(2048);
            while ((fcSrc.read(buffer)) != -1){
                buffer.flip();
                while (buffer.hasRemaining())
                    fcDest.write(buffer);
                buffer.clear();
            }
        }catch (FileNotFoundException fnfe){
            System.err.println(args[0] +
                " could not be opened for input, or " +
                args[1] + " could not be created for output");
        }catch (IOException ioe){
            System.err.println("I/O error: " + ioe.getMessage());
        } finally {
            if (fcSrc != null)
                try{
                    fcSrc.close();
                } catch (IOException ioe) {
                }
        }
    }
}
```

```

        if (fcDest != null)
            try {
                fcDest.close();
            } catch (IOException ioe) {
            }
        }
    }
}

```

50. Listing A-60 presents the ReplaceText application that was called for in Chapter 14.

Listing A-60. Replacing All Matches of the Pattern with Replacement Text

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

public class ReplaceText {
    public static void main(String[] args) {
        if (args.length != 3){
            System.err.println("usage: " +
                "java ReplaceText text oldText newText");
            return;
        }
        try {
            Pattern p = Pattern.compile(args[1]);
            Matcher m = p.matcher(args[0]);
            String result = m.replaceAll(args[2]);
            System.out.println(result);
        } catch (PatternSyntaxException pse) {
            System.err.println(pse);
        }
    }
}

```

Chapter 15: Accessing Databases

1. A database is an organized collection of data.
2. A relational database is a database that organizes data into tables that can be related to each other.
3. A database management system is a set of programs that enables you to store, modify, and extract information from a database. It also provides users with tools to add, delete, access, modify, and analyze data stored in one location.
4. The answer is false: Derby's embedded driver causes the database engine to run in the same virtual machine as the application.
5. JDBC is an API for communicating with RDBMSs in an RDBMS-independent manner.
6. A JDBC driver implements the `java.sql.Driver` interface.
7. The answer is false: there are four kinds of JDBC drivers.
8. A type 3 JDBC driver doesn't depend on native code and communicates with a middleware server via an RDBMS-independent protocol. The middleware server then communicates the client's requests to the data source.
9. JDBC provides the `java.sql.DriverManager` class and the `javax.sql.DataSource` interface for communicating with a data source.

You obtain a connection to an Apache Derby data source via the embedded driver by passing a URL of the form `jdbc:derby:databaseName;URLAttributes` to one of `DriverManager`'s `getConnection()` methods.

10. A SQL state error code is a five-character string consisting of a two-character class value followed by a three-character subclass value.

11. The difference between `SQLNonTransientException` and `SQLTransientException` is as follows: `SQLNonTransientException` describes failed operations that cannot be retried without changing application source code or some aspect of the data source, and `SQLTransientException` describes failed operations that can be retried immediately.
12. JDBC's three statement types are `Statement`, `PreparedStatement`, and `CallableStatement`.
13. The `Statement` method that you call to execute an SQL SELECT statement is `ResultSet executeQuery(String sql)`.
14. A result set's cursor provides access to a specific row of data.
15. The SQL `FLOAT` type maps to Java's `double` type.
16. A prepared statement represents a precompiled SQL statement.
17. The answer is true: `CallableStatement` extends `PreparedStatement`.
18. A stored procedure is a list of SQL statements that perform a specific task.
19. You call a stored procedure by first obtaining a `CallableStatement` implementation instance (via one of `Connection`'s `prepareCall()` methods) that's associated with an escape clause, by next executing `CallableStatement` methods such as `void setInt(String parameterName, int x)` to pass arguments to escape clause parameters, and by finally invoking the `boolean execute()` method that `CallableStatement` inherits from its `PreparedStatement` superinterface.
20. Metadata is data about data.
21. Metadata includes a list of catalogs, base tables, views, indexes, schemas, and additional information.

Chapter 16: Working with XML and JSON Documents

1. XML (Extensible Markup Language) is a metalanguage for defining vocabularies (custom markup languages), which is key to XML's importance and popularity.
2. The XML declaration is special markup that informs an XML parser that the document is XML.
3. The XML declaration's three attributes are `version`, `encoding`, and `standalone`. The `version` attribute is nonoptional.
4. The answer is false: an element can consist of the empty-element tag, which is a stand-alone tag whose name ends with a forward slash (/), such as `<break/>`.
5. Following the XML declaration, an XML document is anchored in a root element.
6. Mixed content is a combination of child elements and content.
7. A character reference is a code that represents the character. The two kinds of character references are numeric character references (such as `Σ`) and character entity references (such as `<`).
8. A CDATA section is a section of literal HTML or XML markup and content surrounded by the `<![CDATA[` prefix and the `]]>` suffix. You would use a CDATA section when you have a large amount of HTML/XML text and don't want to replace each literal `<` (start of tag) and `&` (start of entity) character with its `<` and `&` predefined character entity reference, which is a tedious and possibly error-prone undertaking—you might forget to replace one of these characters.
9. A namespace is a Uniform Resource Identifier-based container that helps differentiate XML vocabularies by providing a unique context for its contained identifiers.

10. A namespace prefix is an alias for the URI.
11. The answer is true: a tag's attributes don't need to be prefixed when those attributes belong to the element.
12. A comment is a character sequence beginning with `<!--` and ending with `-->`. A comment can appear anywhere in an XML document except before the XML declaration, except within tags, and except within another comment.
13. A processing instruction is an instruction that's made available to the application parsing the document. The instruction begins with `<?` and ends with `?>`.
14. The rules that an XML document must follow to be considered well formed are as follows: all elements must either have start and end tags or consist of empty-element tags, tags must be nested correctly, all attribute values must be quoted, empty elements must be properly formatted, and you must be careful with case. Furthermore, XML parsers that are aware of namespaces enforce two additional rules: all element and attribute names must not include more than one colon character; and no entity names, processing instruction targets, or notation names can contain colons.
15. For an XML document to be valid, the document must adhere to certain constraints. For example, one constraint might be that a specific element must always follow another specific element.
16. The two commonly used grammar languages are document type definition and XML Schema.
17. SAX is an event-based API for parsing an XML document sequentially from start to finish. As a SAX-oriented parser encounters an item from the document's infoset, it makes this item available to an application as an event by calling one of the methods in one of the application's handlers, which the application has previously registered with the parser. The application can then consume this event by processing the infoset item in some manner.

APPENDIX A SOLUTIONS TO EXERCISES

18. You obtain a SAX 2-based parser by calling one of the `org.xml.sax.helpers.XMLReaderFactory` class's `createXMLReader()` methods, which returns an `XMLReader` instance.
19. The purpose of the `XMLReader` interface is to describe a SAX parser. This interface makes available several methods for configuring the SAX parser and parsing an XML document's content.
20. You tell a SAX parser to perform validation by invoking `XMLReader`'s `setFeature(String name, boolean value)` method, passing "<http://xml.org/sax/features/validation>" to name and true to value.
21. Ignorable whitespace is whitespace located between tags where the DTD doesn't allow mixed content.
22. The purpose of the `org.xml.sax.helpers.DefaultHandler` class is to serve as a convenience base class for SAX 2 applications. It provides default implementations for all of the callbacks in the four core SAX 2 handler interfaces: `ContentHandler`, `DTDHandler`, `EntityResolver`, and `ErrorHandler`.
23. An entity is aliased data. An entity resolver is an object that uses the public identifier to choose a different system identifier. Upon encountering an external entity, the parser calls the custom entity resolver to obtain this identifier.
24. DOM is an API for parsing an XML document into an in-memory tree of nodes and for creating an XML document from a tree of nodes. After a DOM parser has created a document tree, an application uses the DOM API to navigate over and extract info set items from the tree's nodes.
25. You obtain a document builder by first instantiating `DocumentBuilderFactory` via one of its `newInstance()` methods and then invoking `newDocumentBuilder()` on the returned `DocumentBuilderFactory` instance to obtain a `DocumentBuilder` instance.

26. You use a document builder to parse an XML document by invoking one of `DocumentBuilder`'s `parse()` methods.
27. You use a document builder to create a new XML document by invoking `DocumentBuilder`'s `Document newDocument()` method and by invoking `Document`'s various "create" methods.
28. When creating a new XML document, you cannot use the DOM API to specify the XML declaration's `encoding` attribute.
29. XPath is a non-XML declarative query language (defined by the W3C) for selecting an XML document's infoset items as one or more nodes.
30. XSLT is a family of languages for transforming and formatting XML documents.
31. JSON stands for JavaScript Object Notation and is a format for data exchange between web pages and servers or between servers and other servers
32. The solution is shown in Listing A-61.

Listing A-61. The JSON Code

```
{  
  "title": "Grilled Cheese Sandwich",  
  "ingredients" : [  
    "bread slice",  
    "cheese slice",  
    "margarine pat"  
,  
  "instructions" :  
    "Place frying pan on element ..."  
}
```

33. The solution is shown in Listing A-62.

APPENDIX A SOLUTIONS TO EXERCISES

Listing A-62. Writing JSON from Java

```
Map<String, Object> properties = new HashMap<String, Object>() {{
    put(JsonGenerator.PRETTY_PRINTING, true);
}}
JsonGeneratorFactory jgf = Json.createGeneratorFactory(properties);
StringWriter sw = new StringWriter();
JsonGenerator jg = jgf.createGenerator(sw);
jg.writeStartObject()
    .write("title", "Grilled Cheese Sandwich")
    .writeStartArray("ingredients")
        .write("bread slice")
        .write("cheese slice")
        .write("margarine pat")
    .writeEnd()
    .write("instructions", "Place frying pan ...")
.jg.writeEnd()
.close();
String json = sw.toString();
```

34. The solution is shown in Listing A-63.

Listing A-63. Adjusted JSON Code

```
{
    "title": "Grilled Cheese Sandwich",
    "ingredients" : [
        { "name" : "bread slice", quantity: 2 },
        { "name" : "cheese slice" },
        { "name" : "margarine pat", quantity: 2 }
    ],
    "instructions" : "Place frying pan ..."
}
```

35. The solution is shown in Listing A-64.

Listing A-64. Converting from Java to JSON

```

public class Recipe {
    static public class Ingredient {
        public String name;
        public int quantity = 1;
        // constructors, getter, setter, ...
    }
    public String title;
    public List<Ingredient> ingredients;
    public String instructions;
    // constructors, getter, setter, ...
}

Recipe r = new Recipe() {{
    ingredients = new ArrayList<Ingredient>();
    ingredients.add(new Ingredient(){{
        name="bread slice"; quantity=2; }});
    ingredients.add(new Ingredient(){{
        name="cheese slice"; }});
    ingredients.add(new Ingredient(){{
        name="margarine pat"; quantity=2; }});
    instructions = "Place frying pan ...";
}};
String json = JsonbBuilder.create(
    new JsonbConfig().withFormatting(true)).toJson(r);
System.out.println(json);

```

36. The solution is shown in Listing A-65.

Listing A-65. Converting from JSON to Java

```

String json = "...";
try {
    Jsonb jsonb = JsonbBuilder.create();
    Recipe r = jsonb.fromJson(new StringReader(json), Recipe.class);
    System.out.print(r.title);
} catch(Exception e) {
    e.printStackTrace(System.err);
}

```

Chapter 17: Date and Time

1. A date is a recorded temporal moment, a time zone is a set of geographical regions that share a common number of hours relative to Greenwich Mean Time (GMT), and a calendar is a system of organizing the passage of time.
2. The Date class represents a date as a positive or negative milliseconds value that's relative to the Unix epoch (January 1, 1970 GMT).
3. You obtain a calendar for the default locale that uses a specific time zone by invoking the Calendar `getInstance(TimeZone zone)` factory method.
4. The answer is false: Calendar declares a Date `getTime()` method that returns a calendar's time representation as a Date instance.
5. You would obtain a date formatter to format the time portion of a date in a particular style for the default locale by invoking the DateFormat `getTimeInstance(int style)` factory method.
6. Wrong. LocalDate's granularity is day.
7. Wrong. If you want to have the time zone or a time offset included, you must use ZonedDateTime or OffsetDateTime.
8. The solution is shown in Listing A-66. The 'T' in the format string is enclosed in single quotation marks to mark it as a literal character.

Listing A-66. Formatting Date and Time

```
SimpleDateFormat sdf = new SimpleDateFormat(
    "yyyy-MM-dd'T'HH:mm:ss.SSS");
System.out.println(sdf.format(new Date()));
```

9. The solution is shown in Listing A-67.

Listing A-67. Formatting Date and Time Using the New API

```
DateTimeFormatter dtf = DateTimeFormatter.ofPattern(
    "yyyy-MM-dd'T'HH:mm:ss.SSS");
System.out.println(LocalDateTime.now().format(dtf));
```

10. Use `System.out.println(Clock.systemUTC().millis())`.

11. The solution is shown in Listing A-68. The zone ID must be added in this case; otherwise, we'd get a `ParseException`.

Listing A-68. Parsing a Date and Time String

```
DateTimeFormatter dtf2 = DateTimeFormatter.
    ofPattern("yyyy-MM-dd HH:mm:ss z");
ZonedDateTime zdt = ZonedDateTime.
    parse("2020-01-10 22:16:45"
        + " " + ZoneId.systemDefault().toString(), dtf2);
System.out.println(zdt);
```

12. Write `LocalDateTime ldt = zdt.toLocalDateTime()`.

13. The solution is shown in Listing A-69. The zone ID must be added in this case; otherwise, we'd get a `ParseException`.

Listing A-69. Parsing a Date and Time String

```
DateTimeFormatter dtf3 = DateTimeFormatter.ofPattern(
    "yyyy-MM-dd HH:mm:ss z");
ZonedDateTime zdt2 = ZonedDateTime.parse("2020-01-10 22:16:45"
    + " +00:00", // <- ParseException without that dtf3);
System.out.println(zdt2);
```

14. The solution is shown in Listing A-70.

Listing A-70. Calculating a Duration

```
DateTimeFormatter dtf4 = DateTimeFormatter.ofPattern(
    "yyyy-MM-dd HH:mm:ss");
LocalDateTime ldt4 = LocalDateTime.parse(
    "2020-01-10 22:16:45", dtf4);
LocalDateTime ldt5 = LocalDateTime.parse(
    "2020-01-12 13:10:45", dtf4);
System.out.println( Duration.between(ldt4, ldt5).getSeconds() );
```

15. The solution is shown in Listing A-71.

Listing A-71. A Twice-As-Fast Clock

```
import java.time.Clock;
import java.time.Duration;
import java.time.Instant;
import java.time.ZoneId;

public class ClockTwiceAsFast extends Clock {
    private Instant myStartInstant;
    public ClockTwiceAsFast() {
        myStartInstant = Clock.systemUTC().instant();
    }
    @Override
    public Clock withZone(ZoneId zone) {
        return this;
    }
    @Override
    public ZoneId getZone() {
        return ZoneId.of("Z");
    }
}
```

```
@Override
public Instant instant() {
    Duration dur2 = Duration.between(myStartInstant,
        Clock.systemUTC().instant()).
        multipliedBy(2L);
    return myStartInstant.plus(dur2);
}
```

Index

A

Abstract classes and methods
abstract reserved word, 181
graphics class, 180
instantiating shape, 181
shape class, 180

Advanced language features
annotations (*see* Annotations)
anonymous class, 210–212
enums (*see* Enumerated type)
exceptions (*see* Exceptions)
generics (*see* Generics)
import statement, 217, 218
local class, 212–214
mastering packages, 215
nested types, 203
nonstatic member classes, 208–210
packages (*see* Packages)
static imports, 226, 227
static member classes, 203–208
within classes, 214

allocate() class method, 558

American Standard Code for Information Interchange (ASCII), 505

Android
APIs, 21, 23, 24
application framework, 23, 24
C/C++ libraries, 24
Dalvik virtual machine, 25
description, 21

Google, 22
HelloWorld, 26
implementation, 27
Java code, 25
Java programming language, 23
layered architecture, 22
Linux kernel, 26
Linux process, 25
mobile devices, 21
permissions, 26
runtime environment, 25
sandbox, 26
security model, 26
software stacks, 22
user interface screen, 27
virtual machine, 21, 25

android.database.sqlite packages, 608

Annotations
declaration
element, 254, 255
stub annotation type, 253
stubbed-out method, 253, 254
stub instance's value() element, 255
stub instance's elements, 255
@deprecated annotations, 250
deprecated field, 251, 252
description, 249
@override annotations, 250
processing
 StubFinder application, 259, 260
throws exception, 260, 261

INDEX

- Annotations (*cont.*)
 retention policies, 256
 @Stub annotations, 258
 @suppresswarnings
 annotations, 252
- Application program interfaces (APIs)
 Android, 22
 arrays and collections utility
 binary search, 421, 422
 collections class, 422, 423
 linear search, 421
 methods, 420, 421
- Java Android, 8
- JDK 7 documentation, 8
- legacy collection
 BitSet, 424
 dictionary, 424
 hashtable, 424
 stack, 424
 vector, 423
- software library, 1
- arraycopy() method, 326
- ArrayDeque
 definition, 410
 stack, 411
- Array index operator ([]), 61
- ArrayList
 constructors, 396
 demonstration, 397
 description, 396
- Array type, 99
- Assignment operator (=), 63
- Atomic variables
 classes, 461
 getNextID() class method, 463
 synchronization, 462
- Autoboxing, 394
- B**
- Base problem, 132
- Basic APIs
 absolute values, byte and short integers, 301
- BigDecimal (*see* BigDecimal)
- BigInteger, 307–311
- circumstances, 300
- exploring number, 302
- overloaded methods, 301
- primitive type wrapper classes
 Boolean, 311, 312
 character, 312–314
 float and double, 314–318
 integer, long, short, and byte, 318, 319
- String objects
 constructors and methods, 319
 description, 319
 iteration, 321
- StringBuffer
 constructors and methods, 322
 demonstration, 322
- StringBuilder, 322–324
 fragment code, 324
 initialization code, 323
- synchronization
 checking account, 335–337
 deadlock, 346–349
 mutual exclusion, monitors, and locks, 337, 338
 object, 337
 output, 336
 race condition, 336, 337
 thread-local variables, 350–353
 visibility, 338–340
 waiting and notification, 340–346

system methods, 324–327
 threads
 description, 327, 328
 joining default thread with
 background thread, 333, 334
 methods, 329
 Runnable, Thread(Runnable
 Runnable), currentThread(),
 getName(), and start(), 329, 330
 thread sleep, 331–333
 void run() method, 328
 Windows 7 platform, 330, 331

BigDecimal
 based invoice calculations, 306
 class, 305
 constants, 305

double/float, 303
 floating-point-based invoice
 calculations, 303, 304
 floating-point calculations, 304
 InvoiceCalc, 304, 307
 RoundingMode constants, 305
 salesTaxPercent, 307

BigInteger
 description, 307
 factorial() methods, 308, 309
 output, 310

Bitwise operators, 63
boolean isInfinite() method, 315
boolean isNaN() method, 315
boolean isOpen() methods, 560
Boolean primitive type
 wrapper class, 311, 312

BufferedOutputStream and
 BufferedInputStream, 491, 492

ByteArrayOutputStream and
 ByteArrayInputStream, 481, 482

Byte-order mark (BOM), 590, 644

C

CallableStatement
 Connection's prepareCall()
 methods, 631
 cstmt.execute() method, 632
 definition, 630
 Derby syntax, 630

Camel casing, 102

Cast operator
 compound expressions
 application, 66, 67
 description, 65
 primitive-type conversions, 66
 16-bit unsigned character values, 67

Channels, NIO
 asynchronously closeable, 562
 bytes coping, standard input/output
 streams, 563, 564
 data transfer, 559
 definition, 559
 direct byte buffer, 565
 implementation methods, 560
 interruptible, 562
 I/O classes, 563
 java.nio.channels package, 562
 ReadableByteChannel, 561
 shut down, 562
 WritableByteChannel, 560

Character encoding, 506

Character, primitive type
 wrapper classes, 312–314

Character set, 506

Checked exception, 231

Classes
 application, 102, 103
 arrays
 creation syntax, 148
 element initialization, 147, 148

INDEX

Classes (*cont.*)
 initialization, 148
 multidimensional, 150
 object references, 149
 ragged array, 151
behaviors representation
 via methods, 121
class initializers, 143, 144
class methods
 dumpMatrix(), 123
 header, 122
 public static void main(String[] args), 122, 124
 syntax, 122
declaration, 102
garbage collector, 146, 147
hiding information
 client code, 140
 contract, 136
 equality determination, 141
 getters, 138
 helper methods, 136
 implementation, 136
 interface separation, 138
 Java support, 137
 private methods, 137, 140, 141
 protected, 137
 public, 137
 related language feature, 140
 revising implementation, 139, 140
 setters, 138
initialization order, 146
instance initializers, 144, 145
objects and
 application, 109–111
 argument, 104
 construction syntax, 103, 104
 default constructor, 105
 explicit constructors, 105–109
 noargument constructor, 104
 parameter, 104
 reference variables, 104
return statement
 chaining instance
 method calls, 128–130
 instance method, 127
 Java, 128
 setName() instance method, 127
 syntax, 126
state and behaviors encapsulation
 field-access rules, 120, 121
 fields, declaration and
 access, 111–114
 instance field, declaration and
 access (*see* Instance field)
 instance methods, 124–126
 method-invocation rules, 135
 overloading methods, 133, 134
 pass-by-value passes, 130, 131
 recursion, 131–133
 state via fields, 111
Class field initializer, 111
Classic I/O APIs
 File instance, 468–470
 filesystem (*see* Filesystem)
 java.io package, 516
 RandomAccessFile API, 478
 standard I/O, 503, 505
 streaming sequences, 516
 writers and readers (*see* Writers and
 readers)
Class initializers, 143, 144
ClassLoader, 219
Clock, 725, 726, 728
clone() method, 166
Collections framework

addAll() method, [391](#)
add method, [391](#)
comparable *vs.* **comparator**, [389](#)
core interfaces

- description**, [387](#)
- hierarchy**, [388](#)

definition, [387](#)
elements, [391](#)
implementation classes, [388](#), [389](#)
iterable and collection, [391–395](#)
methods, [391](#)
utility classes, [388](#)
Collector.of() function, [441](#)
comparator() method, [403](#)
CompareTo() method, [271](#), [389](#), [390](#)
Compile-time constant, [114](#)
Composition, [173](#)
Compound expression

- definition**, [51](#)
- operators**, [52–57](#), [59](#)

Concurrency utilities

- atomic variables**, [461](#)
- concurrent collections**
 - java.util.concurrent package**, [457](#)
 - properties**, [456](#)
 - thread-safe collections**, [456](#)
- Condition interface**, [459](#)
- definition**, [450](#)
- executors**, [451](#)
- lock**, [457](#)
- low-level threads API**, problems, [449](#)
- packages**, [450](#), [464](#)
- ReentrantLock class**, [459](#)
- synchronizers**, [454](#)

Conditional operators, [68](#)
Constructors method, [104](#)
CookieHandler getDefault()

- class method**, [547](#)

CookieManager class, [547](#)
Cookies

- CookieHandler class**, [547](#)
- CookieManager**, objects, [547](#)
- description**, [546](#)
- HTTP protocol handler**, [547](#)
- HTTP response**, [547](#)
- listing**, [548](#), [549](#)
- copy()** method, [565](#)
- copyAlt()** method, [565](#)
- Covariant return type**
 - definition**, [185](#)
 - demonstration**, [185](#), [186](#)
 - upcasting and downcasting**, [186](#)
- createStatement()** method, [621](#)
- createTempFile()** methods, [475](#)

D

Database management system (DBMS), [607](#)
Databases, accessing

- Java DB** (*see* Java DB)
- Java supports database**, [607](#)
- JDBC** (*see* Java Database Connectivity (JDBC))
 - SQLite**, [615](#), [617](#)

Data definition language (DDL), [611](#)
DatagramPacket class, [530](#)
DatagramSocket

- client context**, [533](#)
- DatagramPacket(byte[] buf, int length)**, [530](#)
- DatagramSocket() constructor**, [531](#)
- DatagramSocket(int port) constructor**, [531](#)
- definition**, [530](#)
- EchoClient**, [531](#), [532](#)

INDEX

DatagramSocket (*cont.*)
 main() method, 532
 void receive(DatagramPacket dgp), 531
 void send(DatagramPacket dgp), 531

DataOutputStream and
 DataInputStream, 492, 494, 495

Date constructors and methods, 700–702

Dates, Time Zones, and
 Calendars, 699–708, 713–722

Deadlock, 346–349

Default noargument constructor, 105

deleteOnExit() method, 475

@Deprecated annotations, 250

Deques
 array deque, 410, 411
 definition, 409
 methods, 409, 410
 queue/equivalent methods, 410

distinct() stream method, 438

Document object model (DOM)
 advantages, SAX, 672
 creating XML document
 DocumentBuilder, 678, 685, 686
 outputs, 687
 creating XML documents
 DocumentBuilder, 676, 678
 DOMDemo, 685
 exploring DOM API, 676
 levels, 672
 nodes
 attribute, 673
 comment, 674
 document type, 674
 element node, 674
 entity, 674
 entity reference, 675
 leaf node, 673
 notation, 675

parent node, 673
processing instruction node, 675
section, 673
text node, 675

parsing XML document
 DOMDemo, 682
 node methods, 678–681
 XML content, 684
validation, 677, 678

Document type definition (DTD), 657

Double's equals() method, 316

doubleToIntBits() method, 316

Downcasting
 array, 184
 DowncastDemo, 183
 trouble, 182

Drawable interfaces
 declaration, 187
 draw() method, 187
 implementation, 188, 189

DumpArgs
 command-line interface, 13
 javac, 13
 for loop, 12
 main(), 12
 source code, 12
 System.out.println(args[i]), 12

dumpMatrix() class method, 123

Duration and periods, 722–724

E

EchoClient, 527–529

EchoServer, 529

EchoText
 Boolean, 14
 command-line arguments, 14, 16
 javac, 15

main(), 14
 outputting text, 13
 standard input stream, 13
 System.in.read(), 15
 System.out.println(), 15
 while loop, 15
 Employee's main() method, 251
 Employees database creation, 612
 Encapsulation, 111
 EnclosingClass, 213
 encode() method, 592
 Enterprise Edition (EE), 6
 Enumerated type
 class, 290–292
 compile-time type safety, 284
 description, 282
 enhancement, 285–290
 extending class, 292, 293
 switch statement, 285
 traditional, 282–284
 EnumMap, 419
 Enums, `lrange`, 761
 EnumSet, 400, 401
 Equality operators, 70
 Equals() method
 hashCode() method, 171
 point objects, 169, 170
 relation, nonnull object references, 168
 equalTo() method, 141
 Exceptions
 cleanup, 241–245
 custom exception classes, 232–234
 description, 228
 error, 228
 handling, 236–241
 resource management, 244, 245
 rethrowing, 240
 source code representations
 error codes *vs.* objects, 229
 throwable class hierarchy, 230
 throwing, 233–236
 types, 238–240
 Executors
 class's static methods, tasks, 453
 definition, 451
 interface, 452
 “obtain word entries” task, 452, 453
 Expressions
 compound, 51–57, 59
 description, 44
 simple (*see* Simple expressions)
 Extending classes
 inheriting members, 159, 160
 multiple implementation
 inheritance, 165
 overriding a method, 162
 relating two classes, 159
 Extensible Markup Language (XML)
 ASCII character, 644
 character references
 CDATA section, 648
 description, 647
 embedded XML document, 649
 entity, 648
 numeric, 648
 SVG, 649
 comments, 654
 declaration, 643–645
 description, 641
 DOM, parsing and creating, 672–687
 DTD, 657
 elements and attributes
 child elements, 646
 mixed content, 646, 647
 root element, 646
 tree structure, 645

INDEX

- Extensible Markup Language (XML) (*cont.*)
 grammar document, 656
 HTML documents, 642, 643
 namespaces
 default, 653, 654
 description, 649
 Google Chrome, 652
 pair, 650
 prefix, 650
 rules, 655
 URI-based container, 650
 well-formed documents, 655, 656
 XHTML language, 651
non-ASCII character, 645
processing instruction, 654
SAX, parsing, 658
schema
 definition, 657
target, 654
valid document, 656, 657
vocabulary documents, 641
Externalization, 501
- F**
- factorial() method, 132, 308
File locking, 555
FileOutputStream and
 FileInputStream, 483–485
Filesystem
 construction, 468–470
 creation/manipulation, 475, 476
 description, 467, 516
 Dir(ectory) application, 473, 474
 disk space information, 472
 DumpRoots application, 467
 File object's abstract
 pathname, 473
java.io.File class, 467
pathname's file/directory, 471, 472
setting/getting permission, 476–478
specify java Dir c:\windows exe, 474
stored abstract pathname, 470, 471
filter() method, 436
Filter input stream, 490
Filter output stream, 489
Finalization
 definition, 171
 super.finalize(), 171
First-In, First-Out (FIFO) queue, 406
flatMap*() method, 435
Float's equals() method, 316
floatToIntBits() method, 316
flush() method, 598
forEach() method, 437
Formatters, 708, 709
Formatter's Appendable out()
 method, 596
Functional programming
 collection, 439–441
 definition, 431
 distinct, 438
 filtering, 436, 437
 functions, 432
 lambda calculus, 432, 433
 limiting, 438
 mapping, 435, 436
 methods as functions, 442
 operators, 432
 parallelization, 444
 performing actions, 437
 protoonpack, 444
 ranges, 438
 reduction, 438
 single-method interfaces, 443
 skipping, 438

- sorting, 438
- streams, 433–435, 437
- stream utility, 444

G

- Garbage collector
 - referenced object, 147
 - unreferenced object, 147
- Generics
 - and arrays, 279–282
 - generic type
 - actual type argument, 264
 - declaration and using, 265–268
 - description, 263
 - output, 268
 - parameter bounds, 268–272
 - parameterized types, 264
 - parameter scope, 272
 - raw type, 264, 265
 - wildcards, 273–275
 - methods
 - copyList generic method, 277, 278
 - formal_type_parameter_list, 276
 - generic constructors, 279
 - type inference algorithm, 278
 - type safety, 261–263
 - getBoolean() method, 312
 - getCookieStore(), 548
 - getDatabaseMajorVersion()
 - method, 635
 - getFreeSpace() method, 473
 - getInterfaceAddresses() method, 544
 - getLength() method, 149
 - getMethods() method, 260
 - getMTU() method, 544
 - getSharedChar() method, 342
 - getUsableSpace() method, 473

H

- Greedy quantifier, 584
- Greenwich Mean Time (GMT), 700, 728

I

- hashCode() method, 172
- Hash codes
 - hashing, 172
 - Java documentation, 172
- Hash map
 - collision, 415
 - demonstration, 417, 418
 - hash function, 414, 415
 - load factor, 416
- HashSet, description, 400
- HyperText Markup Language (HTML), 539
- HyperText Transfer Protocol (HTTP), 535

INDEX

- Instance field
 - blank final, 119
 - Car Class declaration, 115
 - constant, 119
 - direct access, 115
- Employee class, constant
 - declaration, 120
- initialization via constructors, 117–119
- initializer, 114, 144–146
- lifetime, 120
- nonzero default value, 116
- scope, 120
- syntax, 114
- instanceMethod1() method, 348
- instanceMethod2() method, 348
- Instants, 716, 717, 728
- Integrated development
 - environment (IDE)
 - command-line arguments, 20
 - DumpArgs, 18, 19
 - Eclipse user interface, 18
 - JDK's tools, 16
 - JRE System Library items, 19
 - Program arguments, 20
 - splash screen, 17
 - version 4.3.1, 17
 - welcome tab, 17, 18
 - workbench, 19
 - workspace, 17
- Interface inheritance, 190, 192
- Interfaces
 - agile software development, 194
 - declaration, 186–188
 - drawable interface, 194
 - extending, 192, 193
 - formalizing class, 186
 - implementation, 188, 189
 - profiling, 195
- International Electrotechnical Commission (IEC), 506
- International Organization for Standardization (ISO), 506
- Internet Assigned Names Authority (IANA), 590
- Internet Protocol (IP), 519
- Intersection character class, 581
- int getDatabaseMinorVersion()
 - method, 635
- InvalidMediaFormatException class, 233
- isAcceptable() method, 574
- iterator() method, 569
 - Iterator interface
 - autoboxing and unboxing, 394, 395
 - enhanced for loop statement, 392, 393
 - methods, 392
 - iterator() method, 396, 569
- J, K
- Java
 - APIs, 1
 - bytecode, 4
 - C and C++, 2
 - class file, 5
 - classloader, 4
 - description, 2
 - EE, 6
 - enums
 - 1range, 761
 - execution environment, 3
 - IDE (*see* Integrated development environment (IDE))
 - implementation sizes, 3
 - interpretation, 4
 - JDK (*see* Java SE Development Kit (JDK))

JIT compilation, 4
 JNI, 5
 JVM, 4
 loops and expressions, 3
 object code, 4
 operators, 2
 portability, 5
 SE, 6
 secure environment, 6
 single-line and multi-line, 2
 source code, 2
 standard class library, 3
 sun microsystems, 2

Java 6

- boolean setExecutable(boolean executable), 476
- boolean setExecutable(boolean executable, boolean ownerOnly), 476
- boolean setReadable(boolean readable), 477
- boolean setReadable(boolean readable, boolean ownerOnly), 476
- boolean setWritable(boolean writable), 477
- boolean setWritable(boolean writable, boolean ownerOnly), 477
- long getFreeSpace() returns, 472
- long getTotalSpace() returns, 473
- long getUsableSpace() returns, 473

Java DB

- command-line tools, 611–613
- demos, 611
- embedded database engine,
start up/shut down, 609
- embedded driver, 609
- embedded mode, Derby, 614, 615
- installation and configuration, 611
- multiple clients communication, 610
- starting, Apache Derby server, 613

java.lang.Math class, 299
 java.lang.Number class, 302
 java.lang.Object class, 340
 java.lang.reflect package, 776
 Java native interface (JNI), 5, 491
 java.nio.Buffer class, 557

JavaScript Object Notation (JSON)

- builder pattern, 689
- configuration property, 691, 692
- constructor, 691
- description, 688
- parsing, 693–695
- processing, 689
- vs. XML*, 688

Java SE Development Kit (JDK)

- Android (*see* Android)
- APIs, 8
- App.class, 8
- brace characters, 9
- Command-line interface, 7
- documentation, 8
- DumpArgs, 12, 13
- EchoText, 13, 15, 16
- HelloWorld.class, 11
- home directory, 7
- IDE, 20
- Javac, 11
- line feed, 10
- main(), 9
- public class, 9
- source code, 8
- static and void, 9
- String[] args, 10
- System.out.println, 10

java.text.NumberFormat class, 304

INDEX

java.util.concurrent.atomic package, 465
java.util.concurrent.locks package, 465
java.util.regex.Pattern class, 577
Java virtual machine (JVM), 3
Java Database Connectivity (JDBC), 607
 DataSource, 61, 619
 DriverManager, 618
 drivers types, 618
 Java 7, 619
 statements
 CallableStatement, 630–632
 definition, 621
 executeQuery() method, 624
 JDBCDemo application, 622–624
 methods, 621
 PreparedStatement, 626, 630
 ResultSet, 625
 SQL Type/Java type
 mappings, 625, 626
 URLAttributes, 620
 URL syntax, 620
 Xerial SQLite driver, 620
Join group operation, 534
JsonGenerator.PRETTY_PRINTING
 property, 690
Just-in-time (JIT), 4

L

Lambda calculus, 432, 433

Last-In, First-Out (LIFO) queue, 406

Learning language fundamentals

 application structure, 31–33

 comments

 description, 33

 Javadoc, 34, 35, 37

 multiline, 34

 single-line, 33

description, 31
expressions (*see* Expressions)
identifiers, 38, 39
operators (*see* Operators)
precedence and associativity
 32-bit integer, 79
 compound expression, 76
 interprets, 78
 open and close parentheses, 77
 public class
 CompoundExpressions, 77

statements (*see* Learning statements)
types
 array, 43
 data items, 39
 integers, 39
 user-defined, 42
 variables, 43, 44

Learning statements

 assignment, 80
 break and labeled break
 employee search
 application, 95
 infinite loop, 94

 compound, 80

 continue statement, 96

 decision, 81

 do-while statement, 92, 93

 empty, 93

 If-else (*see* If-else statement)

 if statement, 81, 82

 labeled continue, 96

 loop, 87

 for statement, 88, 89

 simple, 80

 switch, 86, 87

 while statement, 90–92

Leave group operation, 534

Lempel-Ziv-Welch (LZW), 491
 Lexicographic/dictionary, 389
 Linear congruential generator, 360
 LineNumberInputStream, 480
 List
 ArrayList, 396, 397
 description, 395, 396
 LinkedList, 397, 398
 methods, 396, 397
 listIterator() method, 396
 Local class, 212
 Local dates and times, 720, 721
 Locking, 338, 458
 Locking framework
 condition
 methods, 459, 460
 lock
 acquisition and release, 458
 methods, 457
 ReadWriteLock, 460
 ReentrantReadWriteLock
 constructors, 461
 methods, 461
 Logger's connect() method, 229
 LoggerFactory abstract class, 220
 Logging message, 511
 Logical operators, 70
 LongStream class, 438

M

main() method, 35
 map*() methods, 435
 Maps
 collection views, 412, 413
 EnumMap, 419
 entry methods, 413

hash map, 414–418
 IdentityHashMap, 418
 navigable maps (*see* Navigable maps)
 TreeMap, 414
 WeakHashMap, 419
 Math class's random() method, 359
 Member access operator (.), 73
 Memory-mapped file I/O, 555
 Meta-annotations, 256
 Metadata
 CATALOG and SCHEMA, 636
 employee Data Source, 633–635
 function escape clause, 635
 SQL, 635
 SYS schema stores, 636
 Method call operator(), 73
 Method class
 lrange, 776
 Method-invocation stack, 128
 Method overloading, 133
 Multicasting
 definition, 534
 group address, 534
 Multidimensional arrays, 150
 Multiplicative operators, 73
 Multipurpose Internet Mail Extensions
 [MIME] type, 539
 Mutual exclusion, 337

N

NamedNodeMap getAttributes()
 method, 680
 Native instructions, 4
 Natural ordering, 389
 Navigable maps
 definition, 420
 methods, 420

INDEX

Navigable sets
 description, 404
 tree set, 404–406

Negation character class, 580, 581

NetworkInterface
 description, 542
 enumeration, 543
 javac NetInfo.java, 544
 methods, 542

NetworkInterfaceAddress
 enumeration, 544, 545
 methods, 544
 NetInfo, 545

Network interface card (NIC), 520

New I/O
 buffer
 byte-oriented buffer
 logical layout, 556
 definition, 556
 demonstration, 558
 documentation, 557
 java.nio package, 557
 properties, 556
 channels (*see* Channels, NIO)
 charsets
 BOM, 590
 byte sequences, 591
 character-encoding scheme, 589
 character set, 588, 589
 coded character set, 589
 code point, 588
 definition, 589
 IANA, 590
 standards, 590
 string class, 593–595
 Unicode, 588
 definition, 555

formatter
 constructors, 596
 conversions, 597
 demonstration, 598, 599
 output, 599
 printf(), 601
 problem solving ways, 600
 syntaxes, 596
 void close() method, 598

regular expressions (*see* Regular expressions)

scanner
 constructors, 601
 delimiter pattern, 602

selectors
 attachment, 568
 code fragment, 568, 570, 571
 definition, 566
 interest set, 567
 key, 567
 nonblocking mode, 566, 568
 readiness selection, 566
 ready set, 567
 receiving time, server, 575, 576

SelectableChannel registration
 methods, 568
 selectable channels, 567, 571
 selected keys, 569
 server socket channel, 574
 serving time to client, 572, 573
 source code, 576
 streams, 566
 tasks, 569

Nonstatic member class
 declaration, 209
 description, 208
 instance method, 209, 210

NumberFormat's format() method, 305
 NumberFormat getCurrencyInstance()
 method, 304

O

Object-based language, 157
 Objects creation, *see* Classes
 Object serialization, 495
 Object type, 99
 Offset dates and times, 717–719
 Operators
 additive, 59–61
 array index, 61–63
 assignment, 63
 bitwise, 63–65
 cast (*see* Cast operator)
 conditional, 68–70
 equality, 70
 logical, 70–73
 member access, 73
 method call, 73
 multiplicative, 73, 74
 object creation, 74
 relational, 75
 shift, 75, 76
 unary minus/plus, 76
 outputList() method, 275
 OutputStreamWriter's write()
 methods, 509
 @Override annotations, 250

P

Packages
 description, 215
 and JAR files, 225, 226
 names, 216

playing, 220–225
 searching
 compile-time search, 218, 219
 runtime search, 219
 statement, 216, 217
 structure, 101
 Parallelism, 450
 Parallelization, 444
 parse() method, 678
 parseObject() methods, 710
 Parsing, 710–712
 Pass-by-value, 130
 Pathname, 469
 Performance, reflection, 775
 PipedOutputStream and
 PipedInputStream, 485–488
 Polymorphism
 coercion, 174
 description, 174
 overloading, 175
 parametric, 175
 subtype, 175, 177
 Portable Network Graphics (PNG),
 107, 482
 Possessive quantifier, 585, 586
 Predefined character class, 582
 Preincrement and postdecrement
 operators, 60
 PreparedStatement, 626
 Primitive types
 binary *vs.* decimal, 41
 definition, 40
 minimum and maximum values, 41
 twos-complement representation, 41
 Primitive type wrapper classes
 Boolean, 311, 312
 definition, 311
 printDetails() instance method, 124

INDEX

PrintStream

- description, 480, 481
- features, 502
- print() and println() methods, 502
- stream classes, 501
- void println() method, 502

PriorityQueue

- comparator, 408, 409
- description, 406, 407
- integers, 407
- ordering elements, 407

PrivateAccess class, 141

Protocol stack, 520

Protonpack, 444

- Pseudorandom numbers, 360
- public boolean isNaN() method, 315
- public static void main(String[] args) method, 98

Q

Queues

- First-In, First-Out (FIFO) queue, 406
- Last-In, First-Out (LIFO) queue, 406
- methods, 407, 408
- PriorityQueue (*see* PriorityQueue)

R

Race condition, threads, 768

Ragged array, 151

Random

- array of integers, shuffling, 362
- boolean nextBoolean(), 361
- double nextDouble() method, 361
- double nextGaussian(), 361
- float nextFloat(), 361
- int nextInt(), 361

- int nextInt(int n), 361

- linear congruential generator, 360
- long nextLong(), 362
- pseudorandom numbers, 360
- random(long seed), 360, 361
- random number generators, 359
- void nextBytes(byte[] bytes), 361

RandomAccessFile API

- description, 478
- stream (*see* Streams)

Range character class, 581

Readiness selection, 555

Rectangle class, 206

reduce() function, 439

Reflection

- array, 378
- class methods, 363
- class object
 - decompileClass() method, 367
 - decompiler tool, 366–369
 - forName() method, 363–366
 - getClass() method, 367, 368

dynamically loaded class

- newInstance() method, 368, 369
- VideoPlayer class, 369

field

- class, 369, 370
- main() method, 371

method

- class, 371
- main(), 372

package

- manifest.mf, 376, 377
- obtaining information, 373–376
- performance, 775

Regular expressions

- boundary matchers, 583
- capturing groups, 582, 583

- character classes, [580–582](#)
- definition, [577](#)
- matcher methods, [577](#)
- ox regex, [580](#)
- period metacharacter, [580](#)
- playing with, [578, 579](#)
- practical, [587, 588](#)
- quantifier, [584–587](#)
- zero-length matches, [584](#)
- Reification, [279](#)
- Relational DBMSs (RDBMSs), [607](#)
- Reluctant quantifier, [584, 586](#)
- ResultSet getCatalogs() method, [636](#)
- ResultSet getSchemas() method, [636](#)
- Root directory, [467](#)
- Runnable's run() method, [452](#)
- Runtime exception, [231](#)
- Runtime type identification (RTTI), [363](#)

S

- SavingsAccount class, [303](#)
- Scalable Vector Graphics (SVG), [649](#)
- Scrambling
 - output, fonts yield, [490](#)
 - stream of bytes, [490](#)
- select() method, [569](#)
- SelectableChannel class, [567](#)
- Selector open() class method, [568](#)
- ServerSocket
 - accept() method, [524, 525](#)
 - communication, [525, 526](#)
 - echoing data, [526, 527](#)
- setCookiePolicy(), [548, 549](#)
- setName() instance method, [127](#)
- Sets
 - EnumSet, [400, 401](#)
 - HashSet, [400](#)
- navigable sets (*see* Navigable sets)
- sorted sets (*see* Sorted sets)
- TreeSet (*see* TreeSet)
- setScale() method, [306](#)
- setSharedChar() method, [342](#)
- Shallow cloning, [166–168](#)
- Shuffle() methods, [254, 362](#)
- Simple API for XML (SAX)
 - Handler, [662](#)
 - ignorable whitespace, [671](#)
 - methods
 - ControlHandler, [661](#)
 - lexicalHandler, [661](#)
 - XMLReader object, [659](#)
 - SAXDemo, [661, 662](#)
 - svg-examples, [667](#)
 - validation, [659](#)
- Simple character class, [580](#)
- Simple expressions
 - application, [50](#)
 - array initializers, [47](#)
 - Boolean and character literal, [46](#)
 - counter variable, [48](#)
 - escape sequences, [45](#)
 - floating-point literal, [46](#)
 - integer literal, [46](#)
 - literals, [45](#)
 - null literal, [46](#)
 - one and two-dimensional array, [49](#)
 - primitive-type conversions via widening conversion rules, [49, 50](#)
 - string literal, [46](#)
 - void method, [45](#)
- Single-line comment, [33, 98](#)
- Single-method Interfaces, [443](#)

INDEX

Socket
 InputStream getInputStream()
 method, 524
 OutputStream getOutputStream()
 method, 524
Socket-suffixed classes, 523
Socket
 echoing data, 526, 527
 InetAddress dstAddress, int dstPort, 524
 network management software, 521
 options, 522, 523
 packets, 521
 processes, 519, 520
 String dstName, int dstPort, 524
 void close() method, 524
SocketAddress, 521, 522
Sorted maps
 definition, 419
 methods, 420
Sorted sets
 TreeSet, 401–404
SQLite, 615, 617
Standard Edition (SE), 6
Static member class
 class and instance methods, 205
 declaration, 204
 description, 203
 multiple implementations, 205, 206
 rectangle implementations, 207
Streams
 BufferedOutputStream and
 BufferedInputStream, 491, 492
 ByteArrayOutputStream and
 ByteArrayInputStream, 481, 482
 DataOutputStream and
 DataInputStream, 492, 494, 495
 FileOutputStream and
 FileInputStream, 483–485
 FilterOutputStream and
 FilterInputStream, 488–491
 java.io package, 479
 LineNumberInputStream and
 StringBufferInputStream, 480
 output and input, 479
 PipedOutputStream and
 PipedInputStream, 485–488
 PrintStream, 501, 502
 serialization and deserialization
 DataOutputStream and
 DataInputStream classes, 495
 default, 495–501
 virtual machine mechanism, 495
 Stream utility, 444
String, *See also* StringBuffer;
 StringBuilder
 constant pool, 320
 constructors and methods, 319
 description, 319
 iteration, 321
 literal, 320
 StringBufferInputStream, 480
 String representation, 173
 StringTokenizer
 description, 379
 loop context, 380, 381
 parsing or tokenizing, 379
 String str, 379
 String str, String delim, 379
 String str, String delim, boolean
 returnDelims, 380
 Structured Query Language (SQL), 607
 @Stub annotations, 258
Subtraction character class, 581, 582
sum() method, 131
 @SuppressWarnings
 annotations, 252, 256

Synchronizers

- cyclic barriers
 - countdown latch, 454
 - description, 454, 455
 - exchanger, 455
 - semaphore, 455
- exchangers
 - methods, 455

`System.out.println()` method, 105, 161, 178

T

- `TestLogger.class`, 226
- Thread's `currentThread()`
 - method, 330
- Thread's `start()` method, 330
- `ThreadLocal` class, 350
- Thread-local variables, 350–353
- Threads
 - description, 327, 328
 - joining default thread with
 - background, 333, 334
 - methods, 329
 - processors/cores, 331
 - race condition, 768
- Timer, description, 381–383
- `Timertask`
 - description, 381–383
- Time zone, 700
- Top-level classes, 203
- `toString()` method, 173, 288, 599
- Transmission Control
 - Protocol (TCP), 519
- `TreeMap`, defintion, 414
- `TreeSet`
 - constructors, 398
 - description, 398
 - natural ordering, 399

U

- Ultimate superclass, 165, 166
- Unboxing, 394
- Uniform Resource Identifiers (URIs), 650
 - absolute, 542
 - definition, 535
 - fragment, 542
 - relative, 542
 - scheme-specific-part, 542
 - syntax, 541
- Uniform Resource Locator (URL), 535
 - definition, 535
 - `dooutput` property, 538
 - `ListResource`'s source code, 537, 538
 - `openStream()`, 536, 538
 - protocol prefix, 536
 - `String getProtocol()`, 536
 - `String s`, 535
 - `URLConnection`, 538
- Uniform Resource Name (URN), 535
- Union character class, 581
- Universal Naming Convention (UNC), 469
- Unix epoch, 700
- Unlimited serialization, 496
- Upcasting
 - array, 179
 - description, 177
- `URLConnection`, `HttpURLConnection`, 538
- `URLDecoder`
 - decoding rules, 540
 - encoded string, 541
 - `String decode(String s, String enc)`, 540
- `URLEncoder`
 - encoded string, 541
 - encoding rules, 539
 - `String encode(String s, String enc)`, 539
- User Datagram Protocol (UDP), 519

INDEX

V

Varargs methods/constructors, 131
V call() method, 452
Vector's elements() method, 423
Virtual memory, 555
void close() method, 560
void execute(Runnable runnable)
 method, 451
void flush() method, 527
void run() method, 452

W

WeakHashMap, 419
World Wide Web Consortium (W3C), 643

Writers and readers

byte streams, 505
classes, 506, 507
FileWriter and FileReader, 510–512
InputStream, 508
Java's stream classes, 505
OutputStream, 508
stream characters, 505

X, Y, Z

XML (eXtensible Markup Language)
DOM, parsing and creating, 672
schema
 object-oriented approach, 657