Get the Java skills and know-how that you'll need to learn and write successful Android apps



Learn Java for Android Development

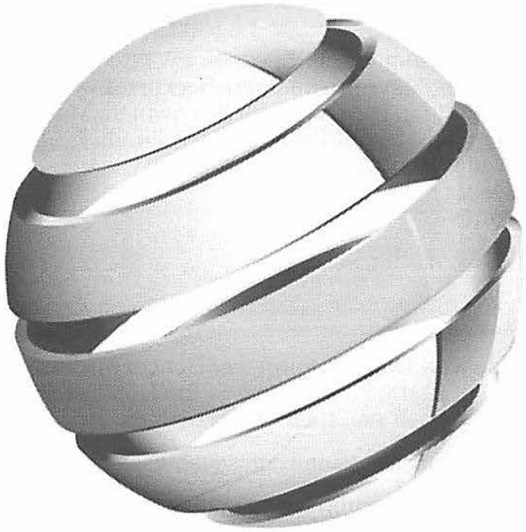
Jeff "JavaJeff" Friesen

Apress®

GOOGLE EXHIBIT 1028

001

Learn Java for Android Development



Jeff "JavaJeff" Friesen

Apress®

Learn Java for Android Development

Copyright © 2010 by Jeff "JavaJeff" Friesen

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN 978-1-4302-3156-1

ISBN 978-1-4302-3157-8 (eBook)

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.

President and Publisher: Paul Manning
Lead Editor: Steve Anglin
Development Editor: Tom Welsh
Technical Reviewer: Paul Connolly
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editor: Debra Kelly
Copy Editor: Bill McManus
Compositor: MacPS, LLC
Indexer: John Collin
Artist: April Milne
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books 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 Special Bulk Sales–eBook Licensing web page at www.apress.com/info/bulksales.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at www.apress.com/book/view/1430231564.

To my best friend

Contents at a Glance

Contents v
About the Author x
About the Technical Reviewer xi
Acknowledgments xii
Introduction xiii
Chapter 1: Getting Started with Java 1
Chapter 2: Learning Language Fundamentals 43
Chapter 3: Learning Object-Oriented Language Features 97
Chapter 4: Mastering Advanced Language Features Part 1 139
Chapter 5: Mastering Advanced Language Features Part 2 181
Chapter 6: Exploring the Basic APIs Part 1 227
Chapter 7: Exploring the Basic APIs Part 2 269
Chapter 8: Discovering the Collections Framework 315
Chapter 9: Discovering Additional Utility APIs 381
Chapter 10: Performing I/O 449
Appendix: Solutions to Exercises 533
Index 595

Contents

Contents at a Glance iv
About the Author x
About the Technical Reviewer xi
Acknowledgments xii
Introduction xiii

Chapter 1: Getting Started with Java 1
What Is Java? 1
Java Is a Language 2
Java Is a Platform 3
Java SE, Java EE, Java ME, and Android 5
Installing and Exploring the JDK 6
Installing and Exploring Two Popular IDEs 12
NetBeans IDE 13
Eclipse IDE 17
Four of a Kind 20
Understanding Four of a Kind 21
Modeling Four of a Kind in Pseudocode 21
Converting Pseudocode to Java Code 23
Compiling, Running, and Distributing FourOfAKind 37
Summary 41
Chapter 2: Learning Language Fundamentals 43
Classes 43
Declaring Classes 44
Introducing Fields 45
Introducing Methods 58
Introducing Constructors 75
Introducing Other Initializers 76
Interface Versus Implementation 82

Objects 85
Creating Objects and Arrays 85
Accessing Fields 87
Calling Methods 89
Garbage Collection 92
Summary 94
Chapter 3: Learning Object-Oriented Language Features 97
Inheritance 97
Extending Classes 98
The Ultimate Superclass 103
Composition 112
The Trouble with Implementation Inheritance 112
Polymorphism 116
Upcasting and Late Binding 117
Abstract Classes and Abstract Methods 120
Downcasting and Runtime Type Identification 121
Covariant Return Types 123
Interfaces 125
Declaring Interfaces 125
Implementing Interfaces 127
Extending Interfaces 130
Why Use Interfaces? 131
Summary 137
Chapter 4: Mastering Advanced Language Features Part 1 139
Nested Types 139
Static Member Classes 139
Nonstatic Member Classes 142
Anonymous Classes 146
Local Classes 148
Interfaces Within Classes 150
Packages 151
What Are Packages? 151
The Package Statement 152
The Import Statement 153
Searching for Packages and Types 154
Playing with Packages 155
Packages and JAR Files 159
Static Imports 160
Exceptions 161
What Are Exceptions? 162
Representing Exceptions in Source Code 162
Throwing Exceptions 166
Handling Exceptions 168
Performing Cleanup 172
Summary 178
Chapter 5: Mastering Advanced Language Features Part 2 181
Assertions 181

Declaring Assertions 182
Using Assertions 183
Avoiding Assertions 188
Enabling and Disabling Assertions 189
Annotations 190
Discovering Annotations 190
Declaring Annotation Types and Annotating Source Code 193
Processing Annotations 198
Generics 200
Collections and the Need for Type Safety 200
Generic Types 202
Generic Methods 212
Enums 214
The Trouble with Traditional Enumerated Types 214
The Enum Alternative 215
The Enum Class 220
Summary 225

Chapter 6: Exploring the Basic APIs Part 1 227

Math APIs 227
Math and StrictMath 227
BigDecimal 234
BigInteger 239
Package Information 243
Primitive Wrapper Classes 247
Boolean 248
Character 250
Float and Double 251
Integer, Long, Short, and Byte 255
Number 257
References API 257
Basic Terminology 257
Reference and ReferenceQueue 259
SoftReference 260
WeakReference 263
PhantomReference 263
Summary 268

Chapter 7: Exploring the Basic APIs Part 2 269

Reflection API 269
String Management 277
String 278
StringBuffer 281
System 284
Threading API 287
Runnable and Thread 288
Thread Synchronization 296
Summary 313

Chapter 8: Discovering the Collections Framework 315

Framework Overview 315
Comparable Versus Comparator 316
Iterable and Collection 318
Iterator and the Enhanced For Loop Statement 321
Autoboxing and Unboxing 323
List 325
ArrayList 329
LinkedList 330
Set 332
TreeSet 332
HashSet 333
EnumSet 337
SortedSet 339
Queue 346
PriorityQueue 348
Map 351
TreeMap 355
HashMap 356
IdentityHashMap 362
WeakHashMap 364
EnumMap 366
SortedMap 367
Utilities 369
Classic Collections Classes 372
Summary 379

Chapter 9: Discovering Additional Utility APIs 381

Concurrency Utilities 381
Executors 381
Synchronizers 390
Concurrent Collections 392
Locks 394
Atomic Variables 397
Internationalization APIs 397
Locales 398
Resource Bundles 400
Break Iterators 409
Collators 413
Dates, Time Zones, and Calendars 415
Formatters 421
Preferences API 428
Random Number Generation 432
Regular Expressions API 434
Summary 447

Chapter 10: Performing I/O 449

File 449
RandomAccessFile 462
Streams 473

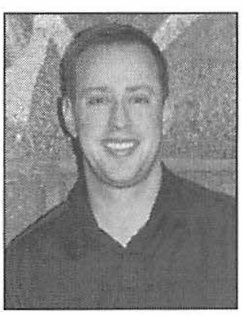
Stream Classes Overview 473
OutputStream and InputStream 475
ByteArrayOutputStream and ByteArrayInputStream 477
FileOutputStream and FileInputStream 479
PipedOutputStream and PipedInputStream 481
FilterOutputStream and FilterInputStream 485
BufferedOutputStream and BufferedInputStream 492
DataOutputStream and DataInputStream 493
Object Serialization and Deserialization 496
PrintStream 508
Writers and Readers 511
Writer and Reader Classes Overview 512
Writer and Reader 513
OutputStreamWriter and InputStreamReader 514
FileWriter and FileReader 518
Summary 530
The Road Goes Ever On 530
Appendix: Solutions to Exercises 533
Chapter 1: Getting Started with Java 533
Chapter 2: Learning Language Fundamentals 539
Chapter 3: Learning Object-Oriented Language Features 542
Chapter 4: Mastering Advanced Language Features Part 1 549
Chapter 5: Mastering Advanced Language Features Part 2 555
Chapter 6: Exploring the Basic APIs Part 1 560
Chapter 7: Exploring the Basic APIs Part 2 563
Chapter 8: Discovering the Collections Framework 569
Chapter 9: Discovering Additional Utility APIs 575
Chapter 10: Performing I/O 581
Index 595

About the Author



Jeff "JavaJeff" Friesen has been actively involved with Java since the late 1990s. Jeff has worked with Java in various companies, including a healthcare-oriented consulting firm, where he created his own Java/C++ software for working with smart cards. Jeff has written about Java in numerous articles for JavaWorld (`www.javaworld.com`), informIT (`www.informit.com`), and java.net (`http://java.net`), and has authored several books on Java, including *Beginning Java SE 6 Platform: From Novice to Professional* (Apress, 2007; ISBN: 159059830X), which focuses exclusively on Java version 6's new and improved features. Jeff has also taught Java in university and college continuing education classes. He has a Bachelor of Science degree in mathematics and computer science from Brandon University in Brandon, Manitoba, Canada, and currently freelances in Java and other software technologies.

About the Technical Reviewer



Paul Connolly is the Director of Engineering for Atypon Systems' RightSuite product line. RightSuite is an enterprise access-control and commerce solution used by many of the world's largest publishing and media companies. Paul enjoys designing and implementing high-performance, enterprise-class software systems. He is also an active contributor in the open-source community.

Prior to joining Atypon Systems, Paul worked as a senior software engineer at Standard & Poor's where he architected and developed key communications systems. Paul is a Sun Certified Java Programmer, Sun Certified Business Component Developer, and a Sun Certified Web Component Developer. Paul lives in New York City with his wife, Marina.

Acknowledgments

I thank Steve Anglin for contacting me to write this book, Debra Kelly for guiding me through the various aspects of this project, Tom Welsh for helping me with the development of my chapters, Paul Connolly for his diligence in catching various flaws that would otherwise have made it into this book, and Bill McManus and the production team for making the book's content look good.

It has been many years since I started writing about Java, and I also thank the following editors who have helped me share my knowledge with others: Chris Adamson, Bridget Collins, Richard Dal Porto, Sean Dixon, Victoria Elzey, Kevin Farnham, Todd Green, Jennifer Orr, Athen O'Shea, Esther Schindler, Daniel Steinberg, Jill Steinberg, Dustin Sullivan, and Atlanta Wilson.

Introduction

Smartphones and other touch-based mobile devices are all the rage these days. Their popularity is largely due to their ability to run *apps*. Although the iPhone and iPad with their growing collection of Objective-C-based apps are the leaders of the pack, Android-based smartphones with their growing collection of Java-based apps are proving to be a strong competitor.

Not only are many iPhone/iPad developers making money by selling their apps, many Android developers are also making money by selling similar apps. According to tech websites such as *The Register* (`www.theregister.co.uk/`), some Android developers are making lots of money (`www.theregister.co.uk/2010/03/02/android_app_profit/`).

In today's tough economic climate, perhaps you would like to try your hand at becoming an Android developer and make some money. 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.

Tip: A good reason to consider Android app development over iPhone/iPad app development is the lower startup costs that you will incur with Android. For example, you do not need to purchase a Mac on which to develop Android apps (a Mac is required for developing iPhone/iPad apps); your existing Windows, Linux, or Unix machine will do nicely.

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

I 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 many important APIs that are fundamental to Android apps and other Java applications. It will also introduce you to key development tools.

Learn Java for Android Development is organized into ten chapters and one appendix. Each chapter focuses on a collection of related topics and presents a set of exercises that you should complete to get the most benefit from the chapter's content. The appendix provides the solutions to each chapter's exercises.

Note: You can download this book's source code by pointing your web browser to `www.apress.com/book/view/1430231564` and clicking the Source Code link under Book Resources. Although most of this code is compilable with Java version 6, you will need Java version 7 to compile one of the applications.

Chapter 1 introduces you to Java by first focusing on Java's dual nature (language and platform). It then briefly introduces you to Sun's/Oracle's Java SE, Java EE, and Java ME editions of the Java development software, as well as Google's Android edition. You next learn how to download and install the Java SE Development Kit (JDK), and learn some Java basics by developing and playing with a pair of simple Java applications. After receiving a brief introduction to the NetBeans and Eclipse IDEs, you learn about application development in the context of *Four of a Kind*, a console-based card game.

Chapter 2 starts you on an in-depth journey of the Java language by focusing on language fundamentals (such as types, expressions, variables, and statements) in the contexts of classes and objects. Because applications are largely based on classes, it is important to learn how to architect classes correctly, and this chapter shows you how to do so.

Chapter 3 adds to Chapter 2's pool of object-based knowledge by introducing you to those 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 the real reason for their inclusion 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 4 introduces you to four categories of advanced language features: nested types, packages, static imports, and exceptions. While discussing nested types, I briefly introduce the concept of a closure and state that closures will appear in Java version 7 (which many expect to arrive later this year).

Note: I wrote this book several months before Java version 7's expected arrival in the fall of 2010. Although I have tried to present an accurate portrayal of version 7–specific language features, it is possible that feature syntax may differ somewhat from what is presented in this book. Also, I only discuss closures briefly because this feature was still in a state of flux while writing this book. For more information about closures and other functional programming concepts (such as lambdas) being considered for Java version 7, I recommend that you check out articles such as "Functional Programming Concepts in JDK 7" by Alex Collins (`http://java.dzone.com/articles/lambdas-closures-jdk-7`).

Chapter 5 continues to explore advanced language features by focusing on assertions, annotations, generics, and enums. Although the topic of generics has brought confusion to many developers, I believe that my discussion of this topic will clear up much of the murkiness. Among other items, you learn how to interpret type declarations such as `Enum<E extends Enum<E>>`.

Chapter 6 begins a trend that focuses more on APIs than language features. This chapter first introduces you to many of Java's math-oriented types (such as `Math`, `StrictMath`, `BigDecimal`, and `BigInteger`), and also introduces you to Java's `strictfp` reserved word. It then looks at the `Package` class, primitive wrapper classes, and the References API.

Chapter 7 continues to explore Java's basic APIs by focusing on reflection, string management, the `System` class, and threading.

Chapter 8 focuses exclusively on Java's collections framework, which provides you with a solution for organizing objects in lists, sets, queues, and maps.

Chapter 9 continues to explore Java's utility APIs by introducing you to the concurrency utilities, internationalization, preferences, random number generation, and regular expressions.

Chapter 10 is all about input/output (I/O). In this chapter, you explore Java's classic I/O support in terms of its `File` class, `RandomAccessFile` class, various stream classes, and various writer/reader classes. My discussion of stream I/O includes coverage of Java's object serialization and deserialization mechanisms.

Note: This book largely discusses APIs that are common to Java SE and Android. However, it diverges from this practice in Chapter 9, where I use the Swing toolkit to provide a graphical user interface for one of this chapter's internationalization examples. (Android does not support Swing.)

After you complete this book, I recommend that you obtain a copy of *Beginning Android 2* by Mark L Murphy (Apress, 2010; ISBN: 1430226293) and start learning how to develop Android apps. In that book, "you'll learn how to develop applications for Android 2.x mobile devices, using simple examples that are ready to run with your copy of the JDK."

Note: Over the next few months, I will make available at my `javajeff.mb.ca` website six additional PDF-based chapters. These chapters will introduce you to more Java APIs (such as networking and database APIs) that I could not discuss in this book because the book has greatly exceeded its initial 400-page estimate (and the good folks at Apress have been gracious enough to let me do so, but there are limits). I present more information about these PDF files at the end of Chapter 10's "Summary" section.

Thanks for purchasing my book. I hope you find it a helpful preparation for, and I wish you lots of success in achieving, a satisfying and lucrative career as an Android app developer.

Jeff "JavaJeff" Friesen, August 2010

Chapter 1

Getting Started with Java

Android is Google's software stack for mobile devices that includes an operating system and middleware. With help from Java, the OS runs specially designed Java applications known as *Android apps*. Because these apps are based on Java, it makes sense for you to learn about Java before you dive into the world of Android development.

NOTE: This book illustrates Java concepts via non-Android Java applications.

This chapter sets the stage for teaching you the essential Java concepts that you need to understand before you embark on your Android career. I first answer the "What is Java?" question. I next show you how to install the Java SE Development Kit, and introduce you to JDK tools for compiling and running Java applications.

After showing you how to install and use the open source NetBeans and Eclipse IDEs so that you can develop these applications faster, I present an application for playing a card game that I call *Four of a Kind*. This application gives you a significant taste of the Java language, and is the centerpiece of my discussion on developing applications.

What Is Java?

Java is a language and a platform originated by Sun Microsystems. This section briefly describes this language and reveals what it means for Java to be a platform. To meet various needs, Sun organized Java into three main editions: Java SE, Java EE, and Java ME. This section also briefly explores each of these editions, along with Android.

NOTE: Java has an interesting history that dates back to December 1990. At that time, James Gosling, Patrick Naughton, and Mike Sheridan (all employees of Sun Microsystems) were given the task of figuring out the next major trend in computing. They concluded that one trend would involve the convergence of computing devices and intelligent consumer appliances. Thus was born the Green project.

The fruits of Green were *Star7*, a handheld wireless device featuring a five-inch color LCD screen, a SPARC processor, a sophisticated graphics capability, and a version of Unix; and *Oak*, a language developed by James Gosling for writing applications to run on Star7, and which he named after an oak tree growing outside of his office window at Sun. To avoid a conflict with another language of the same name, Dr. Gosling changed this language's name to Java.

Sun Microsystems subsequently evolved the Java language and platform until Oracle acquired Sun in early 2010. Check out `http://java.sun.com/` for the latest Java news from Oracle.

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 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 also supports character, double precision floating-point, floating-point, integer, long integer, and short integer primitive types, and via the same `char`, `double`, `float`, `int`, `long`, and `short` reserved words.
- Java also supports many of the same operators, including arithmetic (+, -, *, /, and %) and conditional (?:) operators.
- Java also uses brace characters ({ and }) to delimit blocks of statements.

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

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

- Java supports the byte integer type, does not provided a signed version of the character type, and does not 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).
- Java provides labeled break and continue statements that you will not find in C/C++.

You will learn about single-line and multiline comments in Chapter 2. Also, you will learn about reserved words, primitive types, operators, blocks, and statements (including labeled break and continue) in that chapter.

Java was designed to be a safer language than C/C++. It achieves safety in part by omitting certain C/C++ features. For example, Java does not support *pointers* (variables containing addresses) and does not let you overload operators.

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. (Chapter 2 discusses loops and expressions.)

Suppose you must code a C/C++ while loop that repeats no more than ten times. Being tired, you specify `while (x) x++;` (assume that `x` is an integer-based variable initialized to 0—I discuss variables in Chapter 2) where `x++` adds 1 to `x`'s value. This loop does not stop when x reaches 10; you have introduced a *bug* (a defect).

This problem is less likely to occur in Java because it complains when it sees `while (x)`. This complaint requires you to recheck your expression, and you will then most likely specify `while (x != 10)`. Not only is safety improved (you cannot specify just x), meaning is also clarified: `while (x != 10)` is more meaningful than `while (x)`.

The aforementioned 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 all of these language features.

Java Is a Platform

Java is a platform for executing programs. In contrast to platforms that consist of physical processors (such as an Intel processor) and operating systems (such as Linux), the Java platform consists of a virtual machine and associated execution environment.

The *virtual machine* is a software-based processor that presents its own instruction set. The associated *execution environment* consists of libraries for running programs and interacting with the underlying operating system.

The execution environment includes a huge library of prebuilt classfiles that perform common tasks, such as math operations (trigonometry, for example) 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 instructions (and associated data) that are executed by the virtual machine. These instructions are commonly referred to as *bytecode*.

The compiler stores a program's bytecode and data in files having the `.class` extension. These files are known as *classfiles* because they typically store the compiled equivalent of classes, a language feature discussed in Chapter 2.

A Java program executes via a tool (such as `java`) that loads and starts the virtual machine, and passes the program's main classfile to the machine. The virtual machine uses a *classloader* (a virtual machine or execution environment component) to load the classfile.

After the classfile has been loaded, the virtual machine's *bytecode verifier* component makes sure that the classfile's bytecode is valid and does not compromise security. The verifier terminates the virtual machine when it finds a problem with the bytecode.

Assuming that all is well with the classfile's bytecode, the virtual machine's *interpreter* 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 underlying platform's physical processor.

When the interpreter learns that a sequence of bytecode instructions is executed repeatedly, it informs the virtual machine'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 classfile's bytecode. When that happens, it asks the classloader to load the classfile and the bytecode verifier to verify the bytecode prior to executing that bytecode.

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-based, Linux-based, Mac OS X–based, and other platforms.

NOTE: Java was introduced with the "write once, run anywhere" slogan. Although Java goes to great lengths to enforce portability, it does not always succeed. 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 providing a secure environment in which code executes. The goal is to prevent malicious code from corrupting the underlying platform (and possibly stealing sensitive information).

NOTE: Because many developers are not satisfied with the Java language, but believe that the Java platform is important, they have devised additional languages (such as Groovy) that run on the Java platform. Furthermore, Java version 7 includes an enhanced virtual machine that simplifies adapting even more *dynamic programming languages* (languages that require less-rigid coding; you do not have to define a variable's type before using the variable, for example) to this platform.

Java SE, Java EE, Java ME, and Android

Developers use different editions of the Java platform to create Java programs that run on desktop computers, web browsers, web servers, mobile information devices (such as cell phones), and embedded devices (such as television set-top boxes):

- *Java Platform, Standard Edition (Java SE)*: The Java platform for developing *applications*, which are stand-alone programs that run on desktops. Java SE is also used to develop *applets*, which are programs that run in the context of a web browser.
- *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.
- *Java Platform, Micro Edition (Java ME)*: The Java platform for developing *MIDlets*, which are programs that run on mobile information devices, and *Xlets*, which are programs that run on embedded devices.

Developers also use a special Google-created edition of the Java platform (see `http://developer.android.com/index.html`) to create Android apps that run on Android-enabled devices. This edition is known as the *Android platform*.

Google's Android platform largely consists of Java core libraries (partly based on Java SE) and a virtual machine known as *Dalvik*. This collective software runs on top of a specially modified Linux kernel.

NOTE: Check out Wikipedia's "Android (operating system)" entry (`http://en.wikipedia.org/wiki/Android_%28operating_system%29`) to learn more about the Android OS, and Wikipedia's "Dalvik (software)" entry (`http://en.wikipedia.org/wiki/Dalvik_%28software%29`) to learn more about the Dalvik virtual machine.

In this book, I cover the Java language (supported by Java SE and Android) and Java SE APIs (also supported by Android). Furthermore, I present the source code (typically as code fragments) to Java SE–based applications.

Installing and Exploring the JDK

The *Java Runtime Environment (JRE)* implements the Java SE platform and makes it possible to run Java programs. The public JRE can be downloaded from the Java SE Downloads page (`http://java.sun.com/javase/downloads/index.jsp`).

However, the public JRE does not make it possible to develop Java programs. For that task, you need to download and install the *Java SE Development Kit (JDK)*, which contains development tools (including the Java compiler) and a private JRE.

NOTE: JDK 1.0 was the first JDK to be released (in May 1995). Until JDK 6 arrived, JDK stood for Java Development Kit (SE was not part of the title). Over the years, numerous JDKs have been released, with JDK 7 set for release in fall or winter 2010.

Each JDK's version number identifies a version of Java. For example, JDK 1.0 identifies Java version 1.0, and JDK 5 identifies Java version 5.0. JDK 5 was the first JDK to also provide an internal version number: 1.5.0.

The Java SE Downloads page also provides access to the current JDK, which is JDK 6 Update 20 at time of writing. Click the Download JDK link to download the current JDK's installer program for your platform.

NOTE: Some of this book's code requires JDK 7, which is only available as a preview release (`http://java.sun.com/javase/downloads/ea.jsp`) at time of writing.

The JDK installer installs the JDK in a home directory. (It can also install the public JRE in another directory.) On my Windows XP platform, the home directory is `C:\Program Files\Java\jdk1.6.0_16`—JDK 6 Update 16 was current when I began this book.

TIP: After installing the JDK, you should add the `bin` subdirectory to your platform's `PATH` environment variable. That way, you will be able to execute JDK tools from any directory in your filesystem.

Finally, you might want to create a `projects` subdirectory of the JDK's home directory to organize your Java projects, and create a separate subdirectory within `projects` for each of these projects.

The home directory contains various files (such as `README.html`, which provides information about the JDK, and `src.zip`, which provides the standard class library source code) and subdirectories, including the following three important subdirectories:

- `bin`: This subdirectory contains assorted JDK tools, including the Java compiler tool. You will discover some of these tools shortly.
- `jre`: This subdirectory contains the JDK's private copy of the JRE, which lets you run Java programs without having to download and install the public JRE.
- `lib`: This subdirectory contains library files that are used by JDK tools. For example, `tools.jar` contains the Java compiler's classfiles—the compiler was written in Java.

You will use only a few of the `bin` subdirectory's tools in this book, specifically `javac` (Java compiler), `java` (Java application launcher), `javadoc` (Java documentation generator), and `jar` (Java archive creator, updater, and extractor).

NOTE: `javac` is not the Java compiler. It is a tool that loads and starts the virtual machine, identifies the compiler's main classfile (located in `tools.jar`) to the virtual machine, and passes the name of the source file being compiled to the compiler's main classfile.

You execute JDK tools at the *command line*, passing *command-line arguments* to a tool. Learn about the command line and arguments via Wikipedia's "Command-line interface" entry (`http://en.wikipedia.org/wiki/Command-line_interface`).

Now that you have installed the JDK and know something about its tools, you are ready to explore a small `DumpArgs` application that outputs its command-line arguments to the standard output device.

NOTE: The standard output device is part of a mechanism known as *Standard I/O*. This mechanism, which consists of Standard Input, Standard Output, and Standard Error, and which originated with the Unix operating system, 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 device, which defaults to the keyboard but can be redirected to a file. Text is output to the standard output device, which defaults to the screen but can be redirected to a file. Error message text is output to the standard error device, which defaults to the screen but can be redirected to a file that differs from the standard output file.

Listing 1–1 presents the `DumpArgs` application source code.

Listing 1–1. *Dumping command-line arguments via `main()`'s `args` array to the standard output device*

```
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 1–1's `DumpArgs` application consists of a class named `DumpArgs` and a method within this class named `main()`, which is the application's entry point and provides the code to execute. (You will learn about classes and methods in Chapter 2.)

`main()` is called with an array of *strings* (character sequences) that identify the application's command-line arguments. These strings are stored in `String`-based array variable `args`. (I discuss method calling, arrays, and variables in Chapter 2.)

NOTE: Although the array variable is named `args`, there is nothing special about this name. You could name this variable anything you want.

`main()` first executes `System.out.println("Passed arguments:");`, which calls `System.out`'s `println()` method with the `"Passed arguments:"` string. This method call outputs `Passed arguments:` to the standard output device and then terminates the current line so that subsequent output is sent to a new line immediately below the current line. (I discuss `System.out` in Chapter 7.)

NOTE: `System.out` provides access to a family of `println()` methods and a family of `print()` methods for outputting different kinds of data (such as sequences of characters and integers). Unlike the `println()` methods, the `print()` methods do not terminate the current line; subsequent output continues on the current line.

Each `println()` method terminates a line by outputting a line separator string, which is defined by system property `line.separator`, and which is not necessarily a single newline character (identified in source code via character literal `'\n'`). (I discuss system properties in Chapter 7, `line.separator` in Chapter 10, and character literals in Chapter 2.) For example, on Windows platforms, the line separator string is a carriage return character (whose integer code is 13) followed by a line feed character (whose integer code is 10).

`main()` uses a for loop to repeatedly execute `System.out.println(args[i]);`. The loop executes `args.length` times, which happens to identify the number of strings that are stored in `args`. (I discuss for loops in Chapter 2.)

The `System.out.println(args[i]);` method call reads the string 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`. This method call then outputs this string to standard output.

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

```
javac DumpArgs.java
```

Assuming that that you have 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 will 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 the output destination to a file by specifying the greater than angle bracket (>) followed by a filename. For example, `java DumpArgs Curly Moe Larry >out.txt` stores the `DumpArgs` application's output in a file named `out.txt`.

> **NOTE:** Instead of specifying `System.out.println()`, you could specify `System.err.println()` to output characters to the standard error device. (`System.err` provides the same families of `println()` and `print()` methods as `System.out`.) However, 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.

Congratulations on successfully compiling your first application source file and running the application! Listing 1–2 presents the source code to a second application, which echoes text obtained from the standard input device to the standard output device.

Listing 1–2. *Echoing text read from standard input to standard output*

```
public class EchoText
{
   public static void main(String[] args) throws java.io.IOException
   {
      System.out.println("Please enter some text and press Enter!");
      int ch;
      while ((ch = System.in.read()) != 13)
         System.out.print((char) ch);
      System.out.println();
   }
}
```

After outputting a message that prompts the user to enter some text, `main()` introduces `int` variable `ch` to store each character's integer representation. (You will learn about `int` and integer in Chapter 2.)

`main()` now enters a while loop (discussed in Chapter 2) to read and echo 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 13 (the integer value of the Enter key).

> **NOTE:** When standard input is not redirected to a file, `System.in.read()` returns 13 to indicate that the Enter key has been pressed. On platforms such as Windows, a subsequent call to `System.in.read()` returns integer 10, indicating a line feed character. Whether or not standard input has been redirected, `System.in.read()` returns -1 when there are no more characters to read.

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.println()`. The final `System.out.println();` call terminates the current line without outputting any content.

NOTE: 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 prior to the read operation), `System.in.read()` fails by throwing an object that describes this problem. I acknowledge this possibility by appending `throws java.io.IOException` to the end of the `main()` method header. I discuss `throws` in Chapter 4 and `java.io.IOException` in Chapter 10.

Compile Listing 1–2 via `javac EchoText.java`, and run the application via `java EchoText`. You will be prompted to enter some text. After you input this text and press Enter, the text will be sent to standard output. For example, consider the following output:

```
Please enter some text and press Enter!
Hello Java
Hello Java
```

You can redirect the input source to a file by specifying the less than angle bracket (<) followed by a filename. For example, `java EchoText <EchoText.java` reads its text from `EchoText.java` and outputs this text to the screen.

Run this application and you will only see `EchoText.java`'s first line of text. Each one of this file's lines ends in a carriage return character (13) (followed by a line feed character, 10, on Windows platforms), and `EchoText` terminates after reading a carriage return.

In addition to downloading and installing the JDK, you might want to download the JDK's companion documentation archive file (`jdk-6u18-docs.zip` is the most recent file at time of writing).

After downloading the documentation archive file from the same Java SE Downloads page (`http://java.sun.com/javase/downloads/index.jsp`), unzip this file and move its `docs` directory to the JDK's home directory.

To access the documentation, point your web browser to the documentation's start page. For example, after moving `docs` to my JDK's home directory, I point my browser to `C:\Program Files\Java\jdk1.6.0_16\docs\index.html`. See Figure 1–1.

Scroll a bit down the start page and you discover the "API, Language, and Virtual Machine Documentation" section, which presents a Java 2 Platform API Specification link. Click this link and you can access the standard class library's documentation.

TIP: You can read the online documentation by pointing your web browser to a link such as `http://download.java.net/jdk6/archive/b104/docs/`, which provides the online documentation for JDK 6.

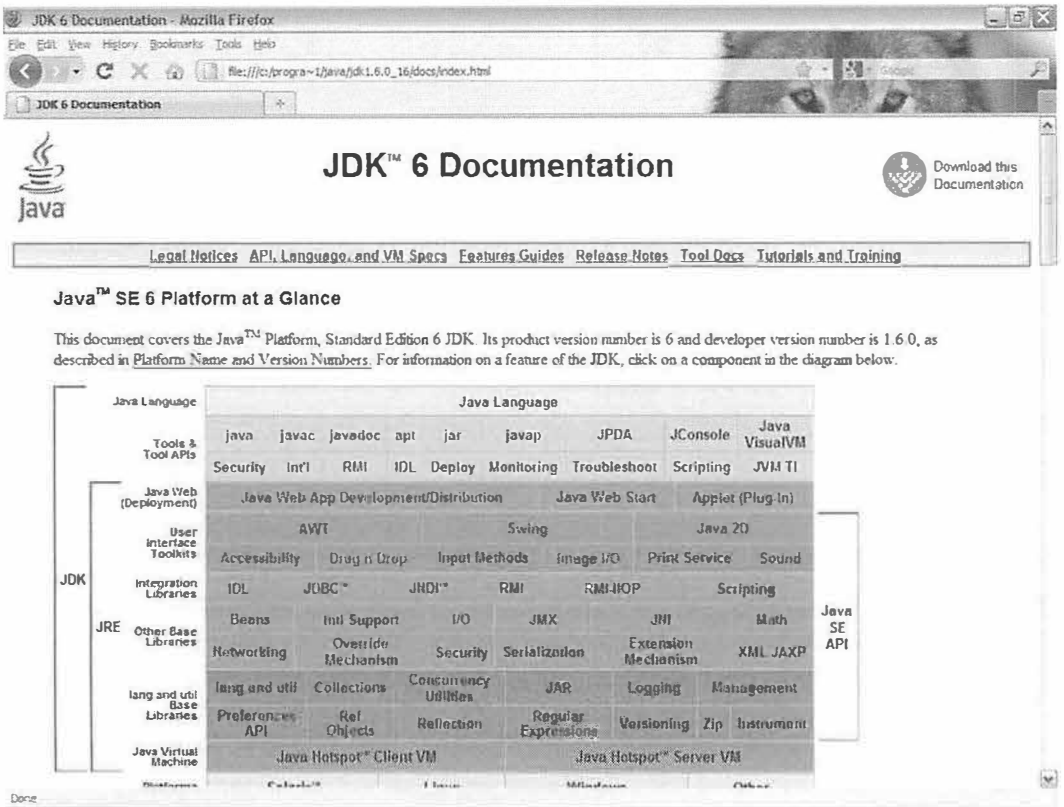


Figure 1–1. *The first part of the Java documentation's start page*

Installing and Exploring Two Popular IDEs

Working with the JDK's tools at the command line is probably okay for small projects. However, this practice is not recommended for large projects, which are hard to manage without the help of an integrated development environment (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. Two popular IDEs are NetBeans and Eclipse.

> **NOTE:** For convenience, I use JDK tools throughout this book, except for this section where I use NetBeans IDE and Eclipse IDE.

NetBeans IDE

NetBeans IDE is an open source, Java-based IDE for developing programs in Java and other languages (such as PHP, Ruby, C++, Groovy, and Scala). Version 6.8 is the current version of this IDE at time of writing.

You should download and install NetBeans IDE 6.8 (or a more recent version) to follow along with this section's NetBeans-oriented example. Begin by pointing your browser to `http://netbeans.org/downloads/` and accomplishing the following tasks:

1. Select an appropriate IDE language (such as English).
2. Select an appropriate platform (such as Linux).
3. Click the Download button underneath the leftmost (Java SE) column.

After a few moments, the current page is replaced by another page that gives you the opportunity to download an installer file. I downloaded the approximately 47MB `netbeans-6.8-ml-javase-windows.exe` installer file for my Windows XP platform.

> **NOTE:** According to the "NetBeans IDE 6.8 Installation Instructions" (`http://netbeans.org/community/releases/68/install.html`), you must install JDK 5.0 Update 19 or JDK 6 Update 14 or newer on your platform before installing NetBeans IDE 6.8. If you do not have a JDK installed, you cannot install the NetBeans IDE.

Start the installer file and follow the instructions. You will need to agree to the NetBeans license, and are given the options of providing anonymous usage data and registering your copy of NetBeans when installation finishes.

Assuming that you have installed the NetBeans IDE, start this Java application. You should discover a splash screen identifying this IDE, followed by a main window similar to that shown in Figure 1–2.

The NetBeans user interface is based on a main window that consists of a menu bar, a toolbar, a workspace area, and a status bar. The workspace area initially presents a Start Page tab, which provides NetBeans tutorials as well as news and blogs.

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

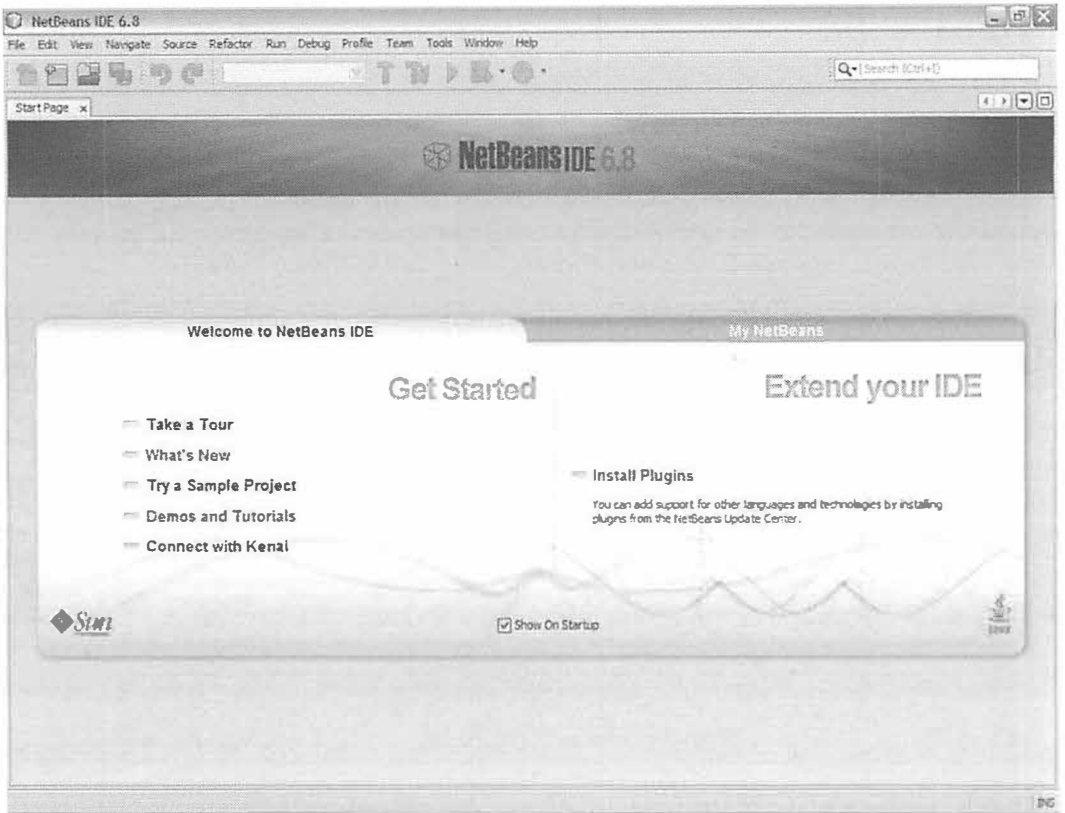


Figure 1–2. *The NetBeans IDE 6.8 main window*

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

1. Select New Project from the File menu.
2. On the resulting New Project dialog box's initial pane, make sure that Java is the selected category in the Categories list, and Java Application is the selected project in the Projects list. Click the Next button.
3. On the resulting pane, enter **DumpArgs** in the Project Name text field. You will notice that dumpargs.Main appears in the text field to the right of the Create Main Class check box. Replace dumpargs.Main with **DumpArgs** and click Finish. (dumpargs names a package, discussed in Chapter 4, and Main names a class stored in this package.)

After a few moments, you will see a workspace similar to that shown in Figure 1–3.

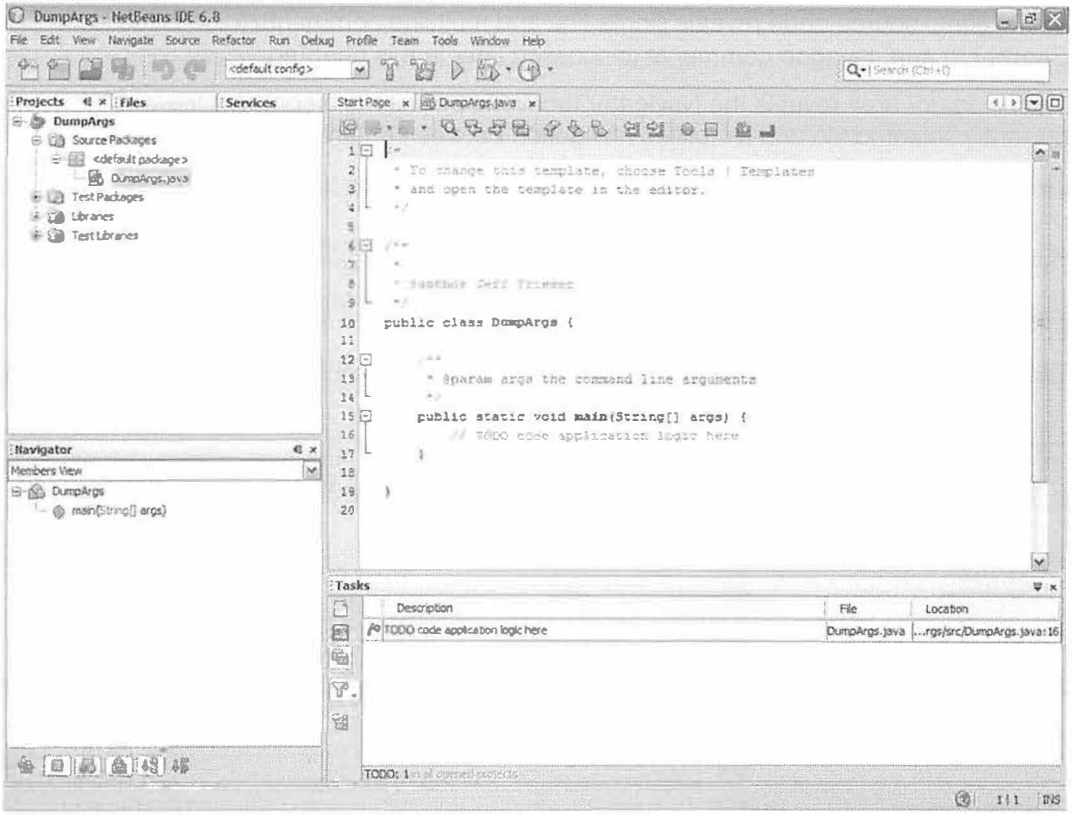


Figure 1–3. *The workspace is divided into multiple work areas.*

After creating the `DumpArgs` project, you will discover that NetBeans has organized the workspace into four main areas: projects, editor, navigator, and tasks.

The *projects area* helps you manage your projects. This window is divided into Projects, Files, and Services tabs:

- The Projects tab is the main entry point to your project's source and resource files. It presents a logical view of important project contents.
- The Files tab presents a directory-based view of your projects, including any files and folders that are not displayed on the Projects tab.
- The Services tab is the main entry point to runtime resources. It shows a logical view of important runtime resources such as the servers, databases, and web services that are registered with the IDE.

The *editor area* helps you edit a project's source files. Each file has its own tab, labeled with the file's name.

Figure 1–3 reveals a single DumpArgs.java tab, which provides access to skeletal source code. You will shortly replace this source code with Listing 1–1.

The skeletal source code reveals single-line and multiline comments (discussed in Chapter 2) and Javadoc comments (discussed later in this chapter).

The *navigator area* reveals the Navigator tab, which presents a compact view of the currently selected file and simplifies navigation between different parts of the file.

The *tasks area* reveals the Tasks tab, which presents a to-do list of items for the project's various files that need to be resolved.

Replace the skeletal DumpArgs.java source code with Listing 1–1, and select Run Main Project from the Run menu to compile and run this application. Figure 1–4 shows this application's results.

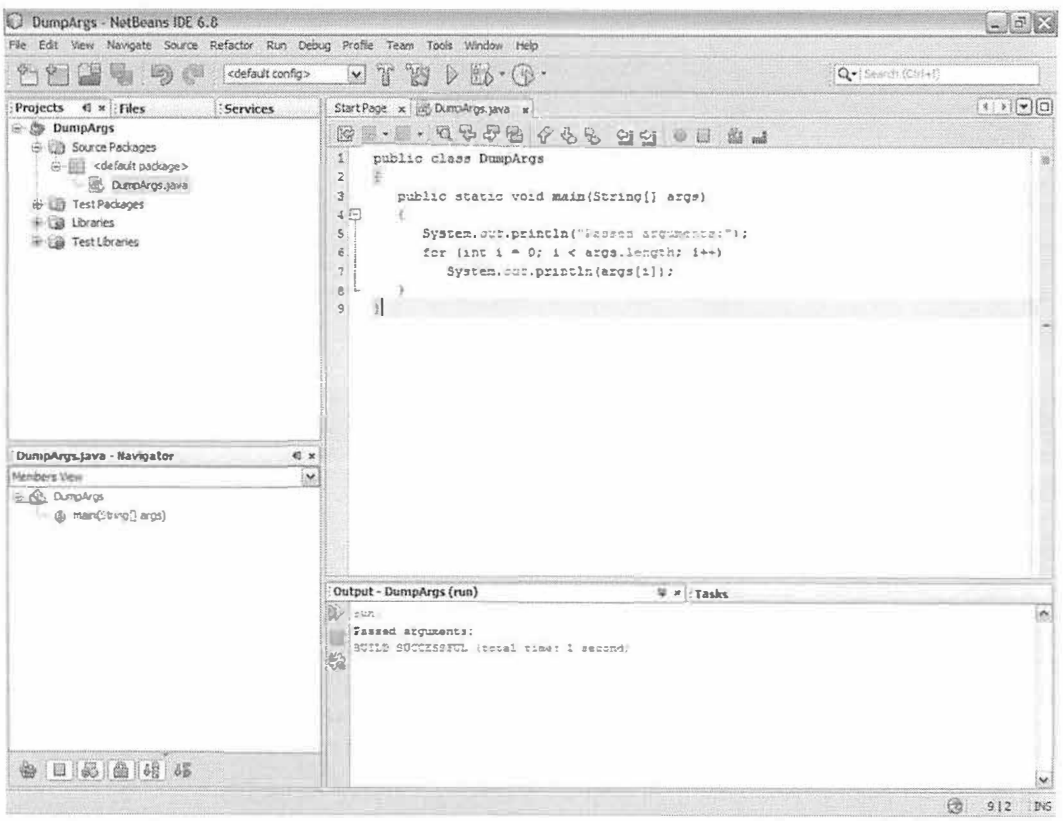


Figure 1–4. *An Output tab appears to the left of the Tasks tab and shows the* `DumpArgs` *application's output.*

Figure 1–4's Output tab reveals only the result of the `System.out.println("Passed arguments:")` method call. To see more output, you must pass command-line arguments to `DumpArgs`. Accomplish this task from within NetBeans IDE 6.8 as follows:

1. Select Project Properties (DumpArgs) from the File menu.
2. In the resulting Project Properties dialog box, select Run in the Categories tree and enter **Curly Moe Larry** in the Arguments text field on the resulting pane. Click the OK button.

Once again, select Run Main Project from the Run menu to run the DumpArgs application. This time, the Output tab should reveal Curly, Moe, and Larry on separate lines below Passed arguments:.

This is all I have to say about the NetBeans IDE. For more information, study the tutorials via the Start Page tab, access IDE help via the Help menu, and explore the NetBeans knowledge base at http://netbeans.org/kb/.

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 Classic is one distribution of this IDE that is available for download; version 3.5.2 is the current version at time of writing.

You should download and install Eclipse Classic to follow along with this section's Eclipse-oriented example. Begin by pointing your browser to http://www.eclipse.org/downloads/ and accomplishing the following tasks:

1. Scroll down the page until you see an Eclipse Classic entry.
2. Click one of the platform links (such as Linux 32 Bit) to the right of this entry.
3. Select a download mirror from the subsequently displayed page and proceed to download the distribution's archive file.

I downloaded the approximately 163MB eclipse-SDK-3.5.2-win32.zip archive file for my Windows XP platform, unarchived this file, moved the resulting eclipse home directory to another location, and created a shortcut to that directory's eclipse.exe file.

> **NOTE:** Unlike NetBeans IDE 6.8, which requires that a suitable JDK be installed before you can run the installer, a JDK does not have to be installed before running eclipse.exe because the Eclipse IDE comes with its own Java compiler. However, you will need at least JDK 6 Update 16 to run most of this book's code (or JDK 7 to run all of the code).

Assuming that you have installed Eclipse Classic, start 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 that shown in Figure 1–5.

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 area initially presents a Welcome tab with icon links for accessing tutorials and more.

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



Figure 1–5. *The Eclipse IDE 3.5.2 main window*

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

1. Select New from the File menu 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 the other defaults and click the Finish button.
3. Click the rightmost (Workbench) icon link to go to the workbench. Eclipse bypasses the Welcome tab and takes you to the workbench the next time you start this IDE.

TIP: To return to the Welcome tab, select Welcome from the Help menu.

After the final step, you will see a workbench similar to that shown in Figure 1–6.

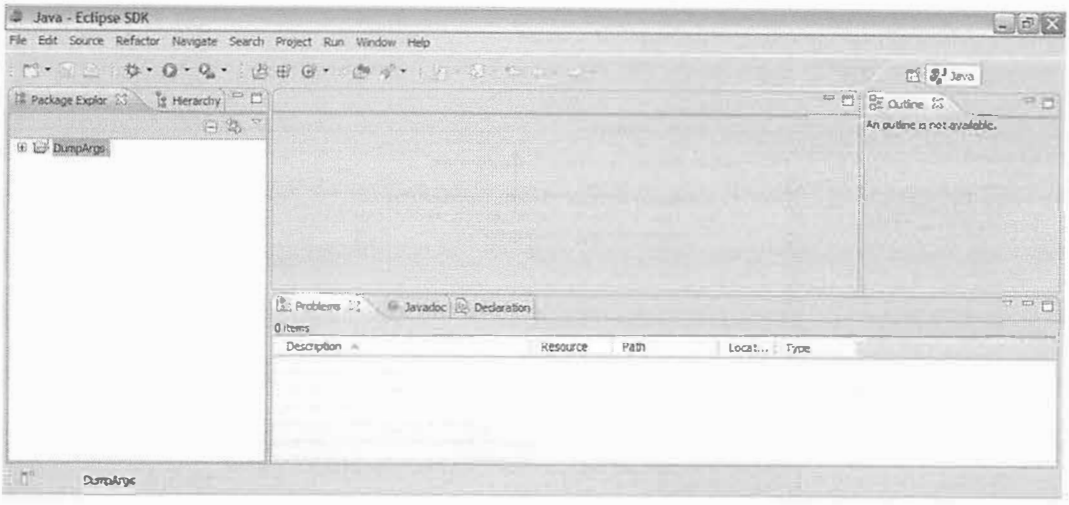


Figure 1–6. *The workbench is divided into multiple work areas.*

On the left side of the workbench, you see a tab titled Package Explorer. This tab identifies the workspace's projects in terms of packages (discussed in Chapter 4). At the moment, only a single DumpArgs entry appears on this tab.

Clicking the + 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 JRE System Library identifies various JRE files that are used to run this application.

We will now add a new file named `DumpArgs.java` to src, as follows:

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

Eclipse responds by displaying an editor tab titled DumpArgs.java. Copy Listing 1–1 into this tab, and then compile and run this application by selecting Run from the Run menu. Figure 1–7 shows the results.

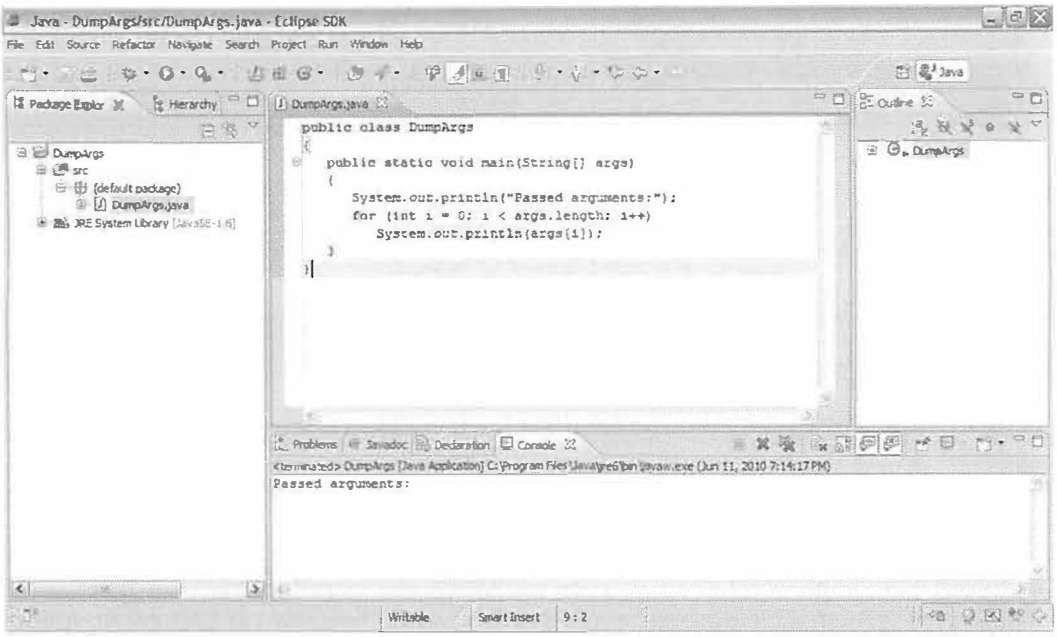


Figure 1–7. *The Console tab at the bottom of the workbench presents the* `DumpArgs` *application's output.*

As with the NetBeans IDE, you must pass command-line arguments to `DumpArgs` to see additional output from this application. Accomplish this task from within Eclipse IDE 3.5.2 as follows:

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.

Once again, select Run 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 I 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 `http://www.eclipse.org/documentation/`.

Four of a Kind

Application development is not an easy task. If you do not plan carefully before you develop an application, you will probably waste your time and money as you endeavor to create it, and waste your users' time and money if it does not meet their needs.

CAUTION: It is extremely important to carefully test your software. You could face a lawsuit if malfunctioning software causes financial harm to its users.

In this section, I present one technique for developing applications efficiently. I present this technique in the context of a Java application that lets you play a simple card game called *Four of a Kind* against the computer.

Understanding Four of a Kind

Before sitting down at the computer and writing code, we need to fully understand the *problem domain* that we are trying to model via that code. In this case, the problem domain is *Four of a Kind*, and we want to understand how this card game works.

Two to four players play *Four of a Kind* with a standard 52-card deck. The object of the game is to be the first player to put down four cards that have the same rank (four aces, for example), which wins the game.

The game begins by shuffling the deck and placing it face down. Each player takes a card from the top of the deck. The player with the highest ranked card (king is highest) deals four cards to each player, starting with the player to the dealer's left. The dealer then starts his/her turn.

The player examines his/her cards to determine which cards are optimal for achieving four of a kind. The player then throws away the least helpful card on a discard pile and picks up another card from the top of the deck. (If each card has a different rank, the player randomly selects a card to throw away.) If the player has four of a kind, the player puts down these cards (face up) and wins the game.

Modeling Four of a Kind in Pseudocode

Now that we understand how *Four of a Kind* works, we can begin to model this game. We will not model the game in Java source code because we would get bogged down in too many details. Instead, we will use pseudocode for this task.

Pseudocode is a compact and informal high-level description of the problem domain. Unlike the previous description of *Four of a Kind*, the pseudocode equivalent is a step-by-step recipe for solving the problem. Check out Listing 1–3.

Listing 1–3. *Four of a Kind pseudocode for two players (human and computer)*

```
1. Create a deck of cards and shuffle the deck.
2. Create empty discard pile.
3. Have each of the human and computer players take a card from the top of the deck.
4. Designate the player with the highest ranked card as the current player.
5. Return both cards to the bottom of the deck.
6. The current player deals four cards to each of the two players in alternating
fashion, with the first card being dealt to the other player.
```

7. The current player examines its current cards to see which cards are optimal for achieving four of a kind. The current player throws the least helpful card onto the top of the discard pile.
8. The current player picks up the deck's top card. If the current player has four of a kind, it puts down its cards and wins the game.
9. Designate the other player as the current player.
10. If the deck has no more cards, empty the discard pile to the deck and shuffle the deck.
11. Repeat at step 7.

Deriving Listing 1–3's pseudocode from the previous description is the first step in achieving an application that implements *Four of a Kind*. This pseudocode performs various tasks, including decision making and repetition.

Despite being a more useful guide to understanding how *Four of a Kind* works, Listing 1–3 is too high level for translation to Java. Therefore, we must refine this pseudocode to facilitate the translation process. Listing 1–4 presents this refinement.

Listing 1–4. *Refined Four of a Kind pseudocode for two players (human and computer)*

```
1. deck = new Deck()
2. deck.shuffle()
3. discardPile = new DiscardPile()
4. hCard = deck.deal()
5. cCard = deck.deal()
6. if hCard.rank() == cCard.rank()
   6.1. deck.putBack(hCard)
   6.2. deck.putBack(cCard)
   6.3. deck.shuffle()
   6.4. Repeat at step 4
7. curPlayer = HUMAN
   7.1. if cCard.rank() > hCard.rank()
        7.1.1. curPlayer = COMPUTER
8. deck.putBack(hCard)
9. deck.putBack(cCard)
10. if curPlayer == HUMAN
   10.1. for i = 0 to 3
         10.1.1. cCards[i] = deck.deal()
         10.1.2. hCards[i] = deck.deal()
   else
   10.2. for i = 0 to 3
         10.2.1. hCards[i] = deck.deal()
         10.2.2. cCards[i] = deck.deal()
11. if curPlayer == HUMAN
   11.01. output(hCards)
   11.02. choice = prompt("Identify card to throw away")
   11.03. discardPile.setTopCard(hCards[choice])
   11.04. hCards[choice] = deck.deal()
   11.05. if isFourOfAKind(hCards)
          11.05.1. output("Human wins!")
          11.05.2. putDown(hCards)
          11.05.3. output("Computer's cards:")
          11.05.4. putDown(cCards)
          11.05.5. End game
   11.06. curPlayer = COMPUTER
```

```
    else
    11.07. choice = leastDesirableCard(cCards)
    11.08. discardPile.setTopCard(cCards[choice])
    11.09. cCards[choice] = deck.deal()
    11.10. if isFourOfAKind(cCards)
              11.10.1. output("Computer wins!")
              11.10.2. putDown(cCards)
              11.10.3. End game
    11.11. curPlayer = HUMAN
12. if deck.isEmpty()
    12.1. if discardPile.topCard() != null
             12.1.1. deck.putBack(discardPile.getTopCard())
             12.1.2. Repeat at step 12.1.
    12.2. deck.shuffle()
13. Repeat at step 11.
```

In addition to being longer than Listing 1–3, Listing 1–4 shows the refined pseudocode becoming more like Java. For example, Listing 1–4 reveals Java expressions (such as `new Deck()`, to create a `Deck` object), operators (such as `==`, to compare two values for equality), and method calls (such as `deck.isEmpty()`, to call `deck`'s `isEmpty()` method to return a Boolean value indicating whether [true] or not [false] the deck identified by `deck` is empty of cards).

Converting Pseudocode to Java Code

Now that you have had a chance to absorb Listing 1–4's Java-like pseudocode, you are ready to examine the process of converting that pseudocode to Java source code. This process consists of a couple of steps.

The first step in converting Listing 1–4's pseudocode to Java involves identifying important components of the game's structure and implementing these components as classes. I will formally introduce classes in Chapter 2.

Apart from the computer player (which is implemented via game logic), the important components are card, deck, and discard pile. I represent these components via `Card`, `Deck`, and `DiscardPile` classes. Listing 1–5 presents `Card`.

> **NOTE:** Do not be concerned if you find this section's Java source code somewhat hard to follow. After you have read the next few chapters, you should find this code easier to understand.

Listing 1–5. *Merging suits and ranks into cards*

```
/**
 *  Simulating a playing card.
 *
 *  @author Jeff Friesen
 */
public enum Card
{
   ACE_OF_CLUBS(Suit.CLUBS, Rank.ACE),
   TWO_OF_CLUBS(Suit.CLUBS, Rank.TWO),
```

```
    THREE_OF_CLUBS(Suit.CLUBS, Rank.THREE),
    FOUR_OF_CLUBS(Suit.CLUBS, Rank.FOUR),
    FIVE_OF_CLUBS(Suit.CLUBS, Rank.FIVE),
    SIX_OF_CLUBS(Suit.CLUBS, Rank.SIX),
    SEVEN_OF_CLUBS(Suit.CLUBS, Rank.SEVEN),
    EIGHT_OF_CLUBS(Suit.CLUBS, Rank.EIGHT),
    NINE_OF_CLUBS(Suit.CLUBS, Rank.NINE),
    TEN_OF_CLUBS(Suit.CLUBS, Rank.TEN),
    JACK_OF_CLUBS(Suit.CLUBS, Rank.JACK),
    QUEEN_OF_CLUBS(Suit.CLUBS, Rank.QUEEN),
    KING_OF_CLUBS(Suit.CLUBS, Rank.KING),
    ACE_OF_DIAMONDS(Suit.DIAMONDS, Rank.ACE),
    TWO_OF_DIAMONDS(Suit.DIAMONDS, Rank.TWO),
    THREE_OF_DIAMONDS(Suit.DIAMONDS, Rank.THREE),
    FOUR_OF_DIAMONDS(Suit.DIAMONDS, Rank.FOUR),
    FIVE_OF_DIAMONDS(Suit.DIAMONDS, Rank.FIVE),
    SIX_OF_DIAMONDS(Suit.DIAMONDS, Rank.SIX),
    SEVEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.SEVEN),
    EIGHT_OF_DIAMONDS(Suit.DIAMONDS, Rank.EIGHT),
    NINE_OF_DIAMONDS(Suit.DIAMONDS, Rank.NINE),
    TEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.TEN),
    JACK_OF_DIAMONDS(Suit.DIAMONDS, Rank.JACK),
    QUEEN_OF_DIAMONDS(Suit.DIAMONDS, Rank.QUEEN),
    KING_OF_DIAMONDS(Suit.DIAMONDS, Rank.KING),
    ACE_OF_HEARTS(Suit.HEARTS, Rank.ACE),
    TWO_OF_HEARTS(Suit.HEARTS, Rank.TWO),
    THREE_OF_HEARTS(Suit.HEARTS, Rank.THREE),
    FOUR_OF_HEARTS(Suit.HEARTS, Rank.FOUR),
    FIVE_OF_HEARTS(Suit.HEARTS, Rank.FIVE),
    SIX_OF_HEARTS(Suit.HEARTS, Rank.SIX),
    SEVEN_OF_HEARTS(Suit.HEARTS, Rank.SEVEN),
    EIGHT_OF_HEARTS(Suit.HEARTS, Rank.EIGHT),
    NINE_OF_HEARTS(Suit.HEARTS, Rank.NINE),
    TEN_OF_HEARTS(Suit.HEARTS, Rank.TEN),
    JACK_OF_HEARTS(Suit.HEARTS, Rank.JACK),
    QUEEN_OF_HEARTS(Suit.HEARTS, Rank.QUEEN),
    KING_OF_HEARTS(Suit.HEARTS, Rank.KING),
    ACE_OF_SPADES(Suit.SPADES, Rank.ACE),
    TWO_OF_SPADES(Suit.SPADES, Rank.TWO),
    THREE_OF_SPADES(Suit.SPADES, Rank.THREE),
    FOUR_OF_SPADES(Suit.SPADES, Rank.FOUR),
    FIVE_OF_SPADES(Suit.SPADES, Rank.FIVE),
    SIX_OF_SPADES(Suit.SPADES, Rank.SIX),
    SEVEN_OF_SPADES(Suit.SPADES, Rank.SEVEN),
    EIGHT_OF_SPADES(Suit.SPADES, Rank.EIGHT),
    NINE_OF_SPADES(Suit.SPADES, Rank.NINE),
    TEN_OF_SPADES(Suit.SPADES, Rank.TEN),
    JACK_OF_SPADES(Suit.SPADES, Rank.JACK),
    QUEEN_OF_SPADES(Suit.SPADES, Rank.QUEEN),
    KING_OF_SPADES(Suit.SPADES, Rank.KING);

    private Suit suit;
    /**
     *  Return <code>Card</code>'s suit.
     *
     *  @return <code>CLUBS</code>, <code>DIAMONDS</code>, <code>HEARTS</code>,
     *  or <code>SPADES</code>
```

```
    */
   public Suit suit() { return suit; }
   private Rank rank;
   /**
    *  Return <code>Card</code>'s rank.
    *
    *  @return <code>ACE</code>, <code>TWO</code>, <code>THREE</code>,
    *  <code>FOUR</code>, <code>FIVE</code>, <code>SIX</code>,
    *  <code>SEVEN</code>, <code>EIGHT</code>, <code>NINE</code>,
    *  <code>TEN</code>, <code>JACK</code>, <code>QUEEN</code>,
    *  <code>KING</code>.
    */
   public Rank rank() { return rank; }
   Card(Suit suit, Rank rank)
   {
      this.suit = suit;
      this.rank = rank;
   }
   /**
    *  A card's suit is its membership.
    *
    *  @author Jeff Friesen
    */
   public enum Suit
   {
      CLUBS, DIAMONDS, HEARTS, SPADES
   }
   /**
    *  A card's rank is its integer value.
    *
    *  @author Jeff Friesen
    */
   public enum Rank
   {
      ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN,
      KING
   }
}
```

Listing 1–5 begins with a Javadoc comment that is used to briefly describe the `Card` class and identify this class's author. I will introduce you to Javadoc comments at the end of this section.

> **NOTE:** One feature of Javadoc comments is the ability to embed HTML tags. These tags specify different kinds of formatting for sections of text within these comments. For example, `<code>` and `</code>` specify that their enclosed text is to be formatted as a code listing. Later in this chapter, you will learn how to convert these comments into HTML documentation.

`Card` is an example of an enum, which is a special kind of class that I discuss in Chapter 5. For now, think of `Card` as a place to create and store `Card` objects that identify all 52 cards that make up a standard deck.

Card declares a nested Suit enum. (I discuss nesting in Chapter 4.) A card's *suit* denotes its membership. The only legal Suit values are CLUBS, DIAMONDS, HEARTS, and SPADES.

Card also declares a nested Rank enum. A card's *rank* denotes its value: ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, and KING are the only legal Rank values.

A Card object is created when Suit and Rank objects are passed to its constructor. (I discuss constructors in Chapter 2.) For example, KING_OF_HEARTS(Suit.HEARTS, Rank.KING) combines Suit.HEARTS and Rank.KING into KING_OF_HEARTS.

Card provides a rank() method for returning a Card's Rank object. Similarly, Card provides a suit() method for returning a Card's Suit object. For example, KING_OF_HEARTS.rank() returns Rank.KING, and KING_OF_HEARTS.suit() returns Suit.HEARTS.

Listing 1–6 presents the Java source code to the Deck class, which implements a deck of 52 cards.

Listing 1–6. *Pick a card, any card*

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 *  Simulate a deck of cards.
 *
 *  @author Jeff Friesen
 */
public class Deck
{
   private Card[] cards = new Card[]
   {
      Card.ACE_OF_CLUBS,
      Card.TWO_OF_CLUBS,
      Card.THREE_OF_CLUBS,
      Card.FOUR_OF_CLUBS,
      Card.FIVE_OF_CLUBS,
      Card.SIX_OF_CLUBS,
      Card.SEVEN_OF_CLUBS,
      Card.EIGHT_OF_CLUBS,
      Card.NINE_OF_CLUBS,
      Card.TEN_OF_CLUBS,
      Card.JACK_OF_CLUBS,
      Card.QUEEN_OF_CLUBS,
      Card.KING_OF_CLUBS,
      Card.ACE_OF_DIAMONDS,
      Card.TWO_OF_DIAMONDS,
      Card.THREE_OF_DIAMONDS,
      Card.FOUR_OF_DIAMONDS,
      Card.FIVE_OF_DIAMONDS,
      Card.SIX_OF_DIAMONDS,
      Card.SEVEN_OF_DIAMONDS,
      Card.EIGHT_OF_DIAMONDS,
      Card.NINE_OF_DIAMONDS,
```

```
    Card.TEN_OF_DIAMONDS,
    Card.JACK_OF_DIAMONDS,
    Card.QUEEN_OF_DIAMONDS,
    Card.KING_OF_DIAMONDS,
    Card.ACE_OF_HEARTS,
    Card.TWO_OF_HEARTS,
    Card.THREE_OF_HEARTS,
    Card.FOUR_OF_HEARTS,
    Card.FIVE_OF_HEARTS,
    Card.SIX_OF_HEARTS,
    Card.SEVEN_OF_HEARTS,
    Card.EIGHT_OF_HEARTS,
    Card.NINE_OF_HEARTS,
    Card.TEN_OF_HEARTS,
    Card.JACK_OF_HEARTS,
    Card.QUEEN_OF_HEARTS,
    Card.KING_OF_HEARTS,
    Card.ACE_OF_SPADES,
    Card.TWO_OF_SPADES,
    Card.THREE_OF_SPADES,
    Card.FOUR_OF_SPADES,
    Card.FIVE_OF_SPADES,
    Card.SIX_OF_SPADES,
    Card.SEVEN_OF_SPADES,
    Card.EIGHT_OF_SPADES,
    Card.NINE_OF_SPADES,
    Card.TEN_OF_SPADES,
    Card.JACK_OF_SPADES,
    Card.QUEEN_OF_SPADES,
    Card.KING_OF_SPADES
};
private List<Card> deck;
/**
 *  Create a <code>Deck</code> of 52 <code>Card</code> objects. Shuffle
 *  these objects.
 */
public Deck()
{
   deck = new ArrayList<Card>();
   for (int i = 0; i < cards.length; i++)
   {
      deck.add(cards[i]);
      cards[i] = null;
   }
   Collections.shuffle(deck);
}
/**
 *  Deal the <code>Deck</code>'s top <code>Card</code> object.
 *
 *  @return the <code>Card</code> object at the top of the
 *  <code>Deck</code>
 */
public Card deal()
{
   return deck.remove(0);
}
/**
```

```
    *  Return an indicator of whether or not the <code>Deck</code> is empty.
    *
    *  @return true if the <code>Deck</code> contains no <code>Card</code>
    *  objects; otherwise, false
    */
   public boolean isEmpty()
   {
      return deck.isEmpty();
   }
   /**
    *  Put back a <code>Card</code> at the bottom of the <code>Deck</code>.
    *
    *  @param card <code>Card</code> object being put back
    */
   public void putBack(Card card)
   {
      deck.add(card);
   }
   /**
    *  Shuffle the <code>Deck</code>.
    */
   public void shuffle()
   {
      Collections.shuffle(deck);
   }
}
```

Deck initializes a private cards array to all 52 Card objects. Because it is easier to implement Deck via a list that stores these objects, Deck's constructor creates this list and adds each Card object to the list. (I discuss List and ArrayList in Chapter 8.)

Deck also provides deal(), isEmpty(), putBack(), and shuffle() methods to deal a single Card from the Deck (the Card is physically removed from the Deck), determine whether or not the Deck is empty, put a Card back into the Deck, and shuffle the Deck's Cards.

Listing 1–7 presents the source code to the DiscardPile class, which implements a discard pile on which players can throw away a maximum of 52 cards.

Listing 1–7. *A garbage dump for cards*

```
/**
 *  Simulate a pile of discarded cards.
 *
 *  @author Jeff Friesen
 */
public class DiscardPile
{
   private Card[] cards;
   private int top;
   /**
    *  Create a <code>DiscardPile</code> that can accommodate a maximum of 52
    *  <code>Card</code>s. The <code>DiscardPile</code> is initially empty.
    */
   public DiscardPile()
   {
      cards = new Card[52]; // Room for entire deck on discard pile (should
```

```
                                // never happen).
      top = -1;
   }
   /**
    *  Return the <code>Card</code> at the top of the <code>DiscardPile</code>.
    *
    *  @return <code>Card</code> object at top of <code>DiscardPile</code> or
    *  null if <code>DiscardPile</code> is empty
    */
   public Card getTopCard()
   {
      if (top == -1)
         return null;
      Card card = cards[top];
      cards[top--] = null;
      return card;
   }
   /**
    *  Set the <code>DiscardPile</code>'s top card to the specified
    *  <code>Card</code> object.
    *
    *  @param card <code>Card</code> object being thrown on top of the
    *  <code>DiscardPile</code>
    */
   public void setTopCard(Card card)
   {
      cards[++top] = card;
   }
   /**
    *  Identify the top <code>Card</code> on the <code>DiscardPile</code>
    *  without removing this <code>Card</code>.
    *
    *  @return top <code>Card</code>, or null if <code>DiscardPile</code> is
    *  empty
    */
   public Card topCard()
   {
      return (top == -1) ? null : cards[top];
   }
}
```

`DiscardPile` implements a discard pile on which to throw `Card` objects. It implements the discard pile via a stack metaphor: the last `Card` object thrown on the pile sits at the top of the pile and is the first `Card` object to be removed from the pile.

This class stores its stack of `Card` objects in a private `cards` array. I found it convenient to specify 52 as this array's storage limit because the maximum number of `Card`s is 52. (Game play will never result in all `Card`s being stored in the array.)

Along with its constructor, `DiscardPile` provides `getTopCard()`, `setTopCard()`, and `topCard()` methods to remove and return the stack's top `Card`, store a new `Card` object on the stack as its top `Card`, and return the top `Card` without removing it from the stack.

The constructor demonstrates a single-line comment, which starts with the `//` character sequence. This comment documents that the `cards` array has room to store the entire `Deck` of `Card`s. I will formally introduce single-line comments in Chapter 2.

The second step in converting Listing 1–4's pseudocode to Java involves introducing a `FourOfAKind` class whose `main()` method contains the Java code equivalent of this pseudocode. Listing 1–8 presents `FourOfAKind`.

Listing 1–8. *FourOfAKind application source code*

```
/**
 *  <code>FourOfAKind</code> implements a card game that is played between two
 *  players: one human player and the computer. You play this game with a
 *  standard 52-card deck and attempt to beat the computer by being the first
 *  player to put down four cards that have the same rank (four aces, for
 *  example), and win.
 *
 *  <p>
 *  The game begins by shuffling the deck and placing it face down. Each
 *  player takes a card from the top of the deck. The player with the highest
 *  ranked card (king is highest) deals four cards to each player starting
 *  with the other player. The dealer then starts its turn.
 *
 *  <p>
 *  The player examines its cards to determine which cards are optimal for
 *  achieving four of a kind. The player then throws away one card on a
 *  discard pile and picks up another card from the top of the deck. If the
 *  player has four of a kind, the player puts down these cards (face up) and
 *  wins the game.
 *
 *  @author Jeff Friesen
 *  @version 1.0
 */
public class FourOfAKind
{
   /**
    *  Human player
    */
   final static int HUMAN = 0;
   /**
    *  Computer player
    */
   final static int COMPUTER = 1;
   /**
    *  Application entry point.
    *
    *  @param args array of command-line arguments passed to this method
    */
   public static void main(String[] args)
   {
      System.out.println("Welcome to Four of a Kind!");
      Deck deck = new Deck(); // Deck automatically shuffled
      DiscardPile discardPile = new DiscardPile();
      Card hCard;
      Card cCard;
      while (true)
      {
         hCard = deck.deal();
         cCard = deck.deal();
         if (hCard.rank() != cCard.rank())
            break;
```

```
      deck.putBack(hCard);
      deck.putBack(cCard);
      deck.shuffle(); // prevent pathological case where every successive
   }                  // pair of cards have the same rank
   int curPlayer = HUMAN;
   if (cCard.rank().ordinal() > hCard.rank().ordinal())
      curPlayer = COMPUTER;
   deck.putBack(hCard);
   hCard = null;
   deck.putBack(cCard);
   cCard = null;
   Card[] hCards = new Card[4];
   Card[] cCards = new Card[4];
   if (curPlayer == HUMAN)
      for (int i = 0; i < 4; i++)
      {
         cCards[i] = deck.deal();
         hCards[i] = deck.deal();
      }
   else
      for (int i = 0; i < 4; i++)
      {
         hCards[i] = deck.deal();
         cCards[i] = deck.deal();
      }
   while (true)
   {
      if (curPlayer == HUMAN)
      {
         showHeldCards(hCards);
         int choice = 0;
         while (choice < 'A' || choice > 'D')
         {
            choice = prompt("Which card do you want to throw away (A, B, " +
                            "C, D)? ");
            switch (choice)
            {
               case 'a': choice = 'A'; break;
               case 'b': choice = 'B'; break;
               case 'c': choice = 'C'; break;
               case 'd': choice = 'D';
            }
         }
         discardPile.setTopCard(hCards[choice-'A']);
         hCards[choice-'A'] = deck.deal();
         if (isFourOfAKind(hCards))
         {
            System.out.println();
            System.out.println("Human wins!");
            System.out.println();
            putDown("Human's cards:", hCards);
            System.out.println();
            putDown("Computer's cards:", cCards);
            return; // Exit application by returning from main()
         }
         curPlayer = COMPUTER;
      }
```

```
      else
      {
         int choice = leastDesirableCard(cCards);
         discardPile.setTopCard(cCards[choice]);
         cCards[choice] = deck.deal();
         if (isFourOfAKind(cCards))
         {
            System.out.println();
            System.out.println("Computer wins!");
            System.out.println();
            putDown("Computer's cards:", cCards);
            return; // Exit application by returning from main()
         }
         curPlayer = HUMAN;
      }
      if (deck.isEmpty())
      {
         while (discardPile.topCard() != null)
            deck.putBack(discardPile.getTopCard());
         deck.shuffle();
      }
   }
}
/**
 *  Determine if the <code>Card</code> objects passed to this method all
 *  have the same rank.
 *
 *  @param cards array of <code>Card</code> objects passed to this method
 *
 *  @return true if all <code>Card</code> objects have the same rank;
 *  otherwise, false
 */
static boolean isFourOfAKind(Card[] cards)
{
   for (int i = 1; i < cards.length; i++)
      if (cards[i].rank() != cards[0].rank())
         return false;
   return true;
}
/**
 *  Identify one of the <code>Card</code> objects that is passed to this
 *  method as the least desirable <code>Card</code> object to hold onto.
 *
 *  @param cards array of <code>Card</code> objects passed to this method
 *
 *  @return 0-based rank (ace is 0, king is 13) of least desirable card
 */
static int leastDesirableCard(Card[] cards)
{
   int[] rankCounts = new int[13];
   for (int i = 0; i < cards.length; i++)
      rankCounts[cards[i].rank().ordinal()]++;
   int minCount = Integer.MAX_VALUE;
   int minIndex = -1;
   for (int i = 0; i < rankCounts.length; i++)
      if (rankCounts[i] < minCount && rankCounts[i] != 0)
      {
```

```
            minCount = rankCounts[i];
            minIndex = i;
         }
      for (int i = 0; i < cards.length; i++)
         if (cards[i].rank().ordinal() == minIndex)
            return i;
      return 0; // Needed to satisfy compiler (should never be executed)
   }
   /**
    *  Prompt the human player to enter a character.
    *
    *  @param msg message to be displayed to human player
    *
    *  @return integer value of character entered by user.
    */
   static int prompt(String msg)
   {
      System.out.print(msg);
      try
      {
         int ch = System.in.read();
         // Erase all subsequent characters including terminating \n newline
         // so that they do not affect a subsequent call to prompt().
         while (System.in.read() != '\n');
         return ch;
      }
      catch (java.io.IOException ioe)
      {
      }
      return 0;
   }
   /**
    *  Display a message followed by all cards held by player. This output
    *  simulates putting down held cards.
    *
    *  @param msg message to be displayed to human player
    *  @param cards array of <code>Card</code> objects to be identified
    */
   static void putDown(String msg, Card[] cards)
   {
      System.out.println(msg);
      for (int i = 0; i < cards.length; i++)
         System.out.println(cards[i]);
   }
   /**
    *  Identify the cards being held via their <code>Card</code> objects on
    *  separate lines. Prefix each line with an uppercase letter starting with
    *  <code>A</code>.
    *
    *  @param cards array of <code>Card</code> objects to be identified
    */
   static void showHeldCards(Card[] cards)
   {
      System.out.println();
      System.out.println("Held cards:");
      for (int i = 0; i < cards.length; i++)
         if (cards[i] != null)
```

```
            System.out.println((char) ('A'+i) + ". " + cards[i]);
        System.out.println();
    }
}
```

Listing 1–8 follows the steps outlined by and expands on Listing 1–4's pseudocode. Because of the various comments, I do not have much to say about this listing. However, there are a couple of items that deserve mention:

- `Card`'s nested `Rank` enum stores a sequence of 13 `Rank` objects beginning with `ACE` and ending with `KING`. These objects cannot be compared directly via `>` to determine which object has the greater rank. However, their integer-based *ordinal* (positional) values can be compared by calling the `Rank` object's `ordinal()` method. For example, `Card.ACE_OF_SPADES.rank().ordinal()` returns 0 because `ACE` is located at position 0 within `Rank`'s list of `Rank` objects, and `Card.KING_OF_CLUBS.rank().ordinal()` returns 12 because `KING` is located at the last position in this list.

- The `leastDesirableCard()` method counts the ranks of the `Cards` in the array of `Card` objects passed to this method, and stores these counts in a `rankCounts` array. For example, given two of clubs, ace of spades, three of clubs, and ace of diamonds, this array identifies one two, two aces, and one three. This method then searches `rankCounts` from smallest index (representing ace) to largest index (representing king) for the first smallest nonzero count (there might be a tie, as in one two and one three)—a zero count represents no `Cards` having that rank in the array of `Card` objects. Finally, the method searches the array of `Card` objects to identify the object whose rank ordinal matches the index of the smallest nonzero count, and returns the index of this `Card` object.

  This behavior implies that the least desirable card is always the smallest ranked card. Furthermore, if there are multiple cards of the same rank, and if this rank is smaller than the rank of another card in the array, this method will choose the first (in a left-to-right manner) of the multiple cards having the same rank as the least desirable card. However, if the rank of the multiple cards is greater than the rank of another card, this other card will be chosen as least desirable.

I previously stated that Listing 1–5 begins with a Javadoc comment that describes the `Card` class and identifies this class's author. A *Javadoc comment* is a documentation item that documents a class, a method, or other program entity.

A Javadoc comment begins with `/**` and ends with `*/`. Sandwiched between these *delimiters* (a pair of characters that mark the start and stop of some section) are text, HTML tags (such as `<p>` and `<code>`), and *Javadoc tags*, which are `@`-prefixed instructions. The following list identifies three common tags:

- `@author` identifies the source code's author.
- `@param` identifies one of a method's parameters (discussed in Chapter 2).
- `@return` identifies the kind of value that a method returns.

The JDK provides a `javadoc` tool that extracts all Javadoc comments from one or more source files and generates a set of HTML files containing this documentation in an easy-to-read format. These files serve as the program's documentation.

For example, suppose that the current directory contains `Card.java`, `Deck.java`, `DiscardPile.java`, and `FourOfAKind.java`. To extract all of the Javadoc comments that appear in these files, specify the following command:

```
javadoc *.java
```

The `javadoc` tool responds by outputting the following messages:

```
Loading source file Card.java...
Loading source file Deck.java...
Loading source file DiscardPile.java...
Loading source file FourOfAKind.java...
Constructing Javadoc information...
Standard Doclet version 1.6.0_16
Building tree for all the packages and classes...
Generating Card.html...
Generating Card.Rank.html...
Generating Card.Suit.html...
Generating Deck.html...
Generating DiscardPile.html...
Generating FourOfAKind.html...
Generating package-frame.html...
Generating package-summary.html...
Generating package-tree.html...
Generating constant-values.html...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating allclasses-noframe.html...
Generating index.html...
Generating help-doc.html...
Generating stylesheet.css...
```

Furthermore, it generates a series of files, including the `index.html` entry-point file. If you point your web browser to this file, you should see a page that is similar to the page shown in Figure 1–8.

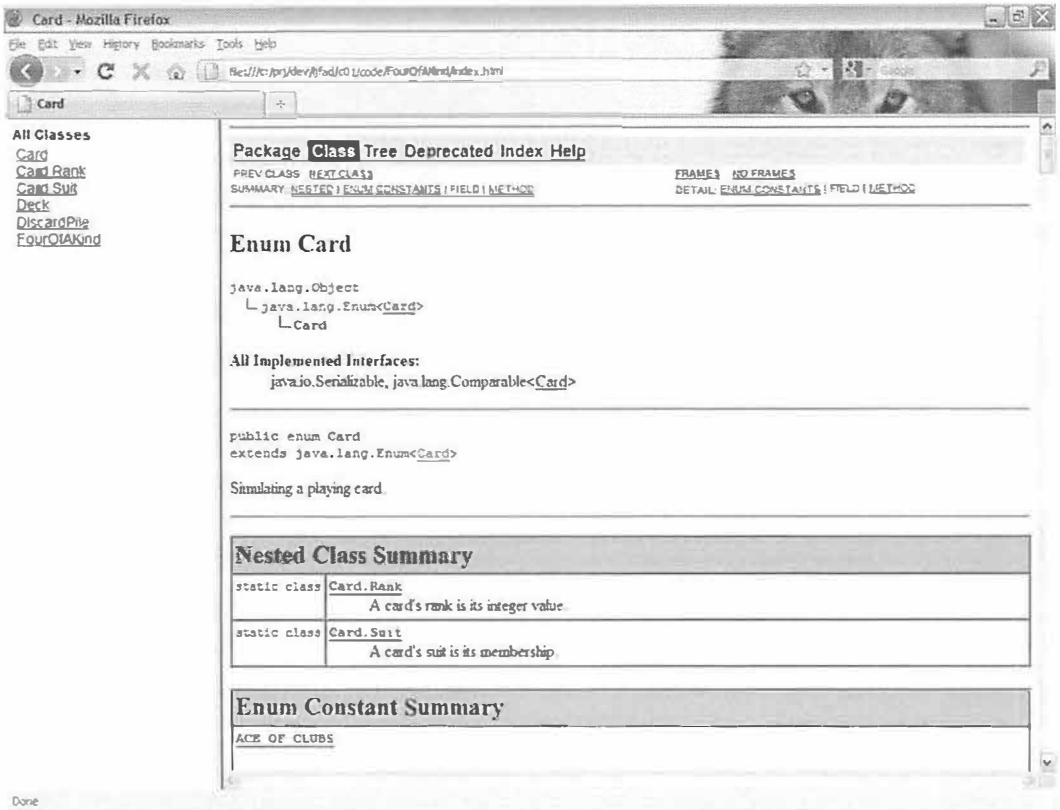


Figure 1–8. *The entry-point page into the generated Javadoc for* `FourOfAKind` *and supporting classes*

`javadoc` defaults to generating HTML-based documentation for public classes and public/protected members of classes. You will learn about public classes and public/protected members of classes in Chapter 2.

For this reason, `FourOfAKind`'s documentation reveals only the public `main()` method. It does not reveal `isFourOfAKind()` and the other package-private methods. If you want to include these methods in the documentation, you must specify `-package` with `javadoc`:

```
javadoc -package *.java
```

> **NOTE:** The standard class library's documentation was also generated by `javadoc` and adheres to the same format.

Compiling, Running, and Distributing FourOfAKind

Unlike the previous DumpArgs and EchoText applications, which each consist of one source file, FourOfAKind consists of Card.java, Deck.java, DiscardPile.java, and FourOfAKind.java. You can compile all four source files via the following command line:

```
javac FourOfAKind.java
```

The javac tool launches the Java compiler, which recursively compiles the source files of the various classes it encounters during compilation. Assuming successful compilation, you should end up with six classfiles in the current directory.

> **TIP:** You can compile all Java source files in the current directory by specifying `javac *.java`.

After successfully compiling FourOfAKind.java and the other three source files, specify the following command line to run this application:

```
java FourOfAKind
```

In response, you see an introductory message and the four cards that you are holding. The following output reveals a single session:

```
Welcome to Four of a Kind!

Held cards:
A. SIX_OF_CLUBS
B. QUEEN_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. NINE_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. FOUR_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. KING_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES
```

```
Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. QUEEN_OF_CLUBS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. KING_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. TWO_OF_HEARTS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. FIVE_OF_DIAMONDS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. JACK_OF_CLUBS
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Held cards:
A. SIX_OF_CLUBS
B. TWO_OF_SPADES
C. SIX_OF_HEARTS
D. SIX_OF_SPADES

Which card do you want to throw away (A, B, C, D)? B

Human wins!

Human's cards:
SIX_OF_CLUBS
SIX_OF_DIAMONDS
SIX_OF_HEARTS
```

```
SIX_OF_SPADES

Computer's cards:
SEVEN_OF_HEARTS
TEN_OF_HEARTS
SEVEN_OF_CLUBS
SEVEN_OF_DIAMONDS
```

Although *Four of a Kind* is not much of a card game, you might decide to share the `FourOfAKind` application with a friend. However, if you forget to include even one of the application's five supporting classfiles, your friend will not be able to run the application.

You can overcome this problem by bundling `FourOfAKind`'s six classfiles into a single *JAR (Java ARchive) file*, which is a ZIP file that contains a special directory and the `.jar` file extension. You can then distribute this single JAR file to your friend.

The JDK provides the `jar` tool for working with JAR files. To bundle all six classfiles into a JAR file named `FourOfAKind.jar`, you could specify the following command line, where `c` tells `jar` to create a JAR file and `f` identifies the JAR file's name:

```
jar cf FourOfAKind.jar *.class
```

After creating the JAR file, try to run the application via the following command line:

```
java -jar FourOfAKind.jar
```

Instead of the application running, you will receive an error message having to do with `java` not knowing which of the JAR file's six classfiles is the *main classfile* (the file whose class's `main()` method executes first).

You can provide this knowledge via a text file that is merged into the JAR file's *manifest*, a special file named `MANIFEST.MF` that stores information about the contents of a JAR file, and which is stored in the JAR file's `META-INF` directory. Consider Listing 1–9.

Listing 1–9. *Identifying the application's main class*

```
Main-Class: FourOfAKind
```

Listing 1–9 tells `java` which of the JAR's classfiles is the main class file. (You must leave a blank line after `Main-Class: FourOfAKind`.)

The following command line, which creates `FourOfAKind.jar`, includes `m` and the name of the text file providing manifest content:

```
jar cfm FourOfAKind.jar manifest *.class
```

This time, `java -jar FourOfAKind.jar` succeeds and the application runs because `java` is able to identify `FourOfAKind` as the main classfile.

EXERCISES

The following exercises are designed to test your understanding of what Java means, the JDK, NetBeans, Eclipse, and Java application development:

1. What is Java?
2. What is a virtual machine?
3. What is the purpose of the Java compiler?
4. True or false: A classfile's instructions are commonly referred to as bytecode.
5. What does the virtual machine'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. What is the JRE?
10. What is the difference between the public and private JREs?
11. What is the JDK?
12. Which JDK tool is used to compile Java source code?
13. Which JDK tool is used to run Java applications?
14. What is the purpose of the JDK's `jar` tool?
15. What is Standard I/O?
16. What is an IDE?
17. Identify two popular IDEs.
18. What is pseudocode?
19. How would you list `FourOfAKind.jar`'s *table of contents* (list of directories and files contained in the JAR file)?
20. Modify `FourOfAKind` to give each player the option of picking up the top card from the deck or discard pile (assuming that the discard pile is not empty; it is empty when the human player goes first and starts his/her first turn). Display the discard pile's top card for the human player's benefit. (Do not display the deck's top card.)

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.

Developers use different editions of the Java platform to create Java programs that run on desktop computers, web browsers, web servers, mobile information devices, and embedded devices. These editions are known as Java SE, Java EE, and Java ME.

Developers also use a special Google-created edition of the Java platform to create Android apps that run on Android-enabled devices. This edition, known as the Android platform, largely consists of Java core libraries and a virtual machine known as Dalvik.

The public JRE implements the Java SE platform and makes it possible to run Java programs. The JDK provides tools (including the Java compiler) for developing Java programs and also includes a private copy of the JRE.

Working with the JDK's tools at the command line is not recommended for large projects, which are hard to manage without the help of an integrated development environment. Two popular IDEs are NetBeans and Eclipse.

Application development is not an easy task. All applications except for the most trivial require careful planning or you will probably waste your (and your users') time and money. One way to develop applications efficiently involves using pseudocode.

`FourOfAKind` gave you a significant taste of the Java language. Although much of its source code is probably hard to understand right now, you will find it much easier to grasp after reading Chapter 2, which introduces you to Java's language fundamentals.

Chapter 2

Learning Language Fundamentals

Aspiring Android developers need to understand the Java language. *Java* is an object-oriented language in which developers use objects to represent *entities* (things that exist, such as vehicles, checking and savings bank accounts, and buttons and other user interface components). Language features supporting this paradigm are the focus of this chapter and Chapter 3. More advanced features are discussed in Chapters 4 and 5. Additional advanced but minor features are more appropriately discussed in later chapters.

Classes

Object-oriented applications represent entities as *objects* (entity abstractions). Each object *encapsulates* (combines into a single unit) an entity's attributes and behaviors. For example, a checking bank account object might encapsulate a balance attribute (with a current value of $50) along with deposit and withdrawal behaviors.

> **NOTE:** Encapsulation exists in stark contrast to the separation of attributes and behaviors in C and other structured programming languages. In such a language, the developer cannot merge an entity's attributes and behaviors into an object. Instead, the developer must separately declare attributes via suitable *data structures* (organizations of data) and behaviors via suitable functions, to which data structure instances that contain attribute values are passed.

Objects do not pop out of thin air; they must be *instantiated* (created) from something. Languages such as C++ and Java refer to this *something* as a *class*, a template for manufacturing objects (also known as *class instances*, or *instances* for short). This section introduces you to Java's language features for architecting classes.

Declaring Classes

Because you cannot instantiate objects from a class that does not exist, you must declare the class. The declaration consists of a header followed by a body. At minimum, the header consists of reserved word `class` followed by a name that identifies the class (so that it can be referred to from elsewhere in the source code). The body starts with an open brace character ({) and ends with a close brace (}). Sandwiched between these *delimiters* (a pair of characters that mark the start and stop of some section) are field, method, and other kinds of declarations. Consider Listing 2–1.

Listing 2–1. *Declaring a skeletal* `CheckingAccount` *class*

```
class CheckingAccount
{
   // field, method, and other member declarations
}
```

Listing 2–1 declares a class named `CheckingAccount`. By convention, a class's name 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.

> **NOTE:** A class declaration is an example of *type*, a template for a set of data values and the operations that can be legally performed on these values.

`CheckingAccount` is one example of an *identifier*, which is a name that identifies a class or other source code entity. Identifiers consist of letters (`A-Z`, `a-z`, or equivalent uppercase/lowercase letters in other human languages), digits (`0-9` or equivalent digits in other human languages), connecting punctuation characters (such as the underscore), and currency symbols (such as the dollar sign). However, an identifier can only begin with a letter, a currency symbol, or a connecting punctuation character. Also, an identifier's maximum length cannot exceed the length of the line in which it appears.

> **NOTE:** Identifiers are expressed in *Unicode* (`http://en.wikipedia.org/wiki/Unicode`), a universal character set that encodes the various symbols making up the world's written languages.

`Additional` examples include `temperature`, `Temperature`, `_class`, `first$name`, and `loopCounter1`. `temperature` and `Temperature` are two different identifiers because Java is a case-sensitive language. In contrast, `6x` and `door^color` are not identifiers because the former character sequence begins with a digit, and the latter character sequence contains an illegal character (^).

You can choose almost any identifier to name classes and other source code entities. However, Java reserves the following identifiers, which are commonly referred to as *reserved words*, for special uses: `abstract`, `assert`, `boolean`, `break`, `byte`, `case`, `catch`, `char`, `class`, `const`, `continue`, `default`, `do`, `double`, `enum`, `else`, `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, and while. Attempting to use any of these reserved words outside of their usage contexts results in compiler error messages.

The class body is currently empty because we have yet to explore fields, methods, and other class members. This fact is pointed out by the *single-line comment*, a line of documentation that begins with //. The compiler ignores everything from // to the end of the line.

Introducing Fields

After declaring a class, you can declare *variables* (memory locations whose values can change) in the class's body. Some of these variables are used in an object context to describe entity attributes. Other variables are used in a class context to describe *class attributes* (attributes that are shared by all created objects, such as a variable that contains a count of all objects created from the class). Regardless of its purpose, a variable that is introduced into a class's body is known as a *field*. This section shows you how to declare fields, how to initialize them to nondefault values, and how to declare read-only fields.

Declaring Fields

You can declare a field within a class's body by minimally specifying a type name, followed by an identifier that names the field, followed by a semicolon character (;). Listing 2–2 presents a pair of field declarations.

Listing 2–2. *Declaring* `owner` *and* `balance` *fields in the* `CheckingAccount` *class*

```
class CheckingAccount
{
   String owner; // name of person who owns this checking account
   int balance; // number of dollars that can be withdrawn
}
```

Listing 2–2 declares two fields named `owner` and `balance`. 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.

A field's *type* identifies the kind of values that can be assigned to the field. The `owner` field is of type `String`, which is one of Java's predefined classes. `String` objects contain sequences of characters; the character sequence in any `String` object assigned to `owner` will contain the name of the account's owner.

The `balance` field is of type integer, which is implied by reserved word `int`. (In common practice, we say that `balance` is of type `int`.) This field can store the account balance as a whole number without a fraction.

Integer is one of several *primitive types* (types whose values are not objects) that are supported by Java. Table 2–1 describes all of Java's primitive types except for void (discussed later).

Table 2–1. *Primitive Types*

Primitive Type	Reserved Word	Size	Min Value	Max Value
Boolean	`boolean`	--	--	--
Character	`char`	16-bit	Unicode 0	Unicode 2^{16} - 1
Byte integer	`byte`	8-bit	-128	+127
Short integer	`short`	16-bit	-2^{15}	$+2^{15}$ - 1
Integer	`int`	32–bit	-2^{31}	$+2^{31}$ - 1
Long integer	`long`	64-bit	-2^{63}	$+2^{63}$ - 1
Floating-point	`float`	32–bit	IEEE 754	IEEE 754
Double precision floating-point	`double`	64-bit	IEEE 754	IEEE 754

Table 2–1 expresses primitive type sizes 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. A group of eight bits is known as a *byte*.

Also, Unicode 0 is shorthand for "the first Unicode character," and IEEE 754 (`http://en.wikipedia.org/wiki/IEEE_754`) refers to a standard for representing floating-point numbers 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 eight-bit byte for performance efficiency), each primitive type's implementation has a specific size.

> **NOTE:** Unlike their C and C++ counterparts, Java's primitive types have the same size in each Java implementation, which partly accounts for the portability of Java applications.

Except for Boolean, whose only values are true and false, each primitive type has a minimum and a maximum value. By studying these limits, you can deduce that the character type is unsigned (all values are positive). In contrast, each numeric type is signed (the type supports positive and negative values).

NOTE: Developers who argue that everything in Java should be an object are not 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.

The `owner` and `balance` fields are examples of non-array fields because each field can hold only one value—an *array* is a multivalue variable; each *array element* (storage slot) holds one of these values. Java also lets you declare array-based fields, which, as Listing 2–3 reveals, are identified by the presence of square brackets ([and]).

Listing 2–3. *Declaring `cities` and `temperatures` array-based fields in a `WeatherData` class*

```
class WeatherData
{
   String country;
   String[] cities;
   double[][]temperatures;
}
```

Listing 2–3 declares a `WeatherData` class that holds temperature extremes for a variety of cities in a specific country. The one-dimensional `cities` array contains the names of these cities; the two-dimensional `temperatures` array contains the maximum and minimum temperature values for each city.

A *one-dimensional array* is a sequential list of values; it is identified in source code by one pair of square brackets. A *two-dimensional array* is a table of values; it is identified in source code by two pairs of square brackets. These brackets can appear on either side of the field name, but are often shown with the type name. (Java also supports higher-dimensional arrays.)

`CheckingAccount`'s `owner` and `balance` fields, and `WeatherData`'s `country`, `cities`, and `temperatures` fields are examples of *instance fields* because they associate with objects. Each `CheckingAccount` instance is given its own copy of `owner` and `balance` when the object is created. Similarly, each `WeatherData` instance is given its own copy of `country`, `cities`, and `temperatures` when the object is created. Modifying any field's value does not affect the value in any other copy of that field.

In many situations, instance fields are all that you need. However, you might encounter a situation where you need a single copy of a field no matter how many objects are created.

For example, suppose you want to track the number of `CheckingAccount` objects that have been created, and introduce a `counter` field (initialized to 0) into this class. You also place code in the class's constructor (which I will present when I discuss constructors) to increase `counter`'s value by 1 when an object is created. However, because each object has its own copy of the `counter` field, this field's value never advances past 1.

You can solve this problem by declaring `counter` to be a *class field*, a field that associates with a class instead of with that class's objects. Listing 2–4 accomplishes this task by prefixing `counter`'s declaration with the `static` reserved word.

Listing 2–4. *Adding a `counter` class field to `CheckingAccount`*

```
class CheckingAccount
{
   String owner;
   int balance;
   static int counter;
}
```

Listing 2–4's `static` prefix implies that there is only one copy of the `counter` field, not one copy per object. Each time an object is created, `counter` will increase by 1, and you will get an accurate tally of all `CheckingAccount` objects that have been created.

Each of `owner` and `balance` is created when a `CheckingAccount` object is created, and destroyed when the object is destroyed. In contrast, `counter` is created when `CheckingAccount` is loaded into memory, and destroyed when this class is removed from memory (when the application ends). This quality is known as *lifetime*.

Each of `owner` and `balance` can be accessed only from an instance context (such as a constructor). In contrast, `counter` can be accessed from instance and class contexts (such as a class method, discussed later in this chapter). This quality is known as *scope*.

Initializing Fields

Listing 2–4's `owner` and `balance` fields are initialized to default null and 0 values (respectively) when an object is created from the `CheckingAccount` class. The `counter` field is initialized to 0 when this class is loaded into memory.

It is common to initialize an instance field to a (potentially unique) value from within the class's constructor. However, you might want to explicitly assign some reasonable nonunique default value to this field, and give yourself the option of overriding that default value via the constructor. In contrast, you often explicitly initialize class fields.

Regardless of how you initialize a field, the value results from evaluating an *expression*, which is a combination of variables, method *calls* (invocations), *literals* (values specified verbatim), and operators. The expression's type must agree with the field's type when assigning the expression to the field. Otherwise, the compiler reports an error.

Simple Expressions

A *simple expression* is a value expressed as a variable name (value is read from the variable), a method call (value is returned from the method), or a literal in source code.

Java supports several kinds of literals: string, Boolean, character, integer, floating-point, and `null` (which I will discuss later in this chapter).

A *string literal* consists of a sequence of Unicode characters surrounded by a pair of double quotes ("). Example: `"The quick brown fox jumps over the lazy dog."`

A string literal might also contain *escape sequences*, which are special *syntax* (rules for combining symbols into language features) for representing certain printable and nonprintable characters that otherwise cannot appear in the literal. For example, string literal `"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 linefeed)
`\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 represent, as a character literal, an escape sequence (`'\''`, for example) 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, then 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 default decimal, hexadecimal, octal, and (starting with Java version 7) binary formats:

- The decimal format is the default format. Example: `255`
- The hexadecimal format requires that the literal be prefixed with `0x` or `0X` and continue with hexadecimal digits (`0-9`, `A-F`, `a-f`). Example: `0xFF`

- The octal format requires that the literal be prefixed with `0` and continue with octal digits (`0`-`7`). Example: `077`
- The binary format requires that the literal be prefixed with `0b` or `0B` and continue with `0`s and `1`s. Example: `0b11111111`

Java version 7 also adds another nice feature where integer literals are concerned: it lets you insert underscores between digits to improve readability. Example: `123_456_789`. Although you can insert multiple successive underscores between digits (as in `0b1111__0000`), you cannot specify a leading underscore (as in `_123`) because the compiler would treat the literal as an identifier. Also, you cannot specify a trailing underscore (as in `123_`).

Finally, a *floating-point literal* consists of an integer part, a decimal point (represented by the period character (`.`)), 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), `10F` (floating-point), `10D` (double precision floating-point), and `3.0E+23` (double precision floating-point).

Suppose you need to create a class whose objects are used to log messages to a specific file. Each instance will have a field that identifies the target file, initialized to a default filename. Listing 2–5 presents this class and shows you how to initialize this field.

Listing 2–5. *Initializing the `Logger` class's `filename` field*

```
class Logger
{
   String filename = "log.txt";
}
```

Listing 2–5 uses simple initialization to initialize a non-array field. You can also use simple initialization with array-based fields. For example, Listing 2–6 shows you how to initialize array-based `cities` and `temperatures` fields.

Listing 2–6. *Initializing the `WeatherData` class's fields*

```
class WeatherData
{
   String country = "United States";
   String[] cities = {"Chicago", "New York", "Los Angeles"};
   double[][] temperatures = {{0.0, 0.0}, {0.0, 0.0}, {0.0, 0.0}};
}
```

Listing 2–6's `WeatherData` class defaults its objects to providing weather data for three American cities. The `cities` array is initialized to a comma-separated list of three `String` literals, which are specified between braces; the `temperatures` array is initialized to a three-row by two-column table of double precision floating-point zeros, which is specified as a three-element row array with each comma-separated element containing a two-element column array.

Compound Expressions

A *compound expression* is a sequence of simple expressions and operators, where an *operator* is a sequence of instructions (symbolically represented in source code) that transforms one or more values, known as *operands*, into another value.

For example, the + symbol denotes the addition or string concatenation operator, depending on the types of its two operands. When this symbol appears between numeric operands (6+4, for example), addition is implied. Similarly, when + appears between string literals (as in `"A"+"B"`), string concatenation is implied.

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

These 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 is sandwiched between the binary operator's two or the ternary operator's three operands.

Table 2–3 describes all supported operators. (I will explain precedence after this table.)

Table 2–3. *Operators*

Operator	Symbol	Description	Precedence
Addition	+	Given *operand1* + *operand2*, where each operand must be of character or numeric type, add *operand2* to *operand1* and return the sum.	10
Array index	[]	Given *variable*[*index*], where *index* must be of integral type, read value from or store value into *variable*'s storage element at location *index*.	13
Assignment	=	Given *variable* = *operand*, which must be assignment-compatible (their types must agree), store *operand* in *variable*.	0
Bitwise AND	&	Given *operand1* & *operand2*, 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 if each operand's corresponding bit is 1. Otherwise, the result bit is set to 0.	6
Bitwise complement	~	Given ~*operand*, where *operand* must be of character or integer type, flip *operand*'s bits (1s to 0s and 0s to 1s) and return the result.	12
Bitwise exclusive OR	^	Given *operand1* ^ *operand2*, 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 if 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

Operator	Symbol	Description	Precedence
Bitwise inclusive OR	`|`	Given *operand1* `|` *operand2*, 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 if either (or both) of the operands' corresponding bits is 1. Otherwise, the result bit is set to 0.	4
Cast	`(`*type*`)`	Given `(`*type*`)` *operand*, convert *operand* to an equivalent value that can be represented by *type*. For example, you could use this operator to convert a floating-point value to a 32–bit integer value.	12
Compound assignment	`+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=`	Given *variable operator operand*, where *operator* is one of the listed compound operator symbols, and where *operand* is assignment-compatible with *variable*, perform the indicated operation using *variable*'s value as *operator*'s left operand value, and store the resulting value in *variable*.	0
Conditional	`?:`	Given *operand1* `?` *operand2* `:` *operand3*, where *operand1* must be of Boolean type, return *operand2* if *operand1* is true or *operand3* if *operand1* is false. The types of *operand2* and *operand3* must agree.	1
Conditional AND	`&&`	Given *operand1* `&&` *operand2*, where each operand must be of Boolean type, return true if both operands are true. Otherwise, return false. If *operand1* is false, *operand2* is not examined. This is known as *short-circuiting*.	3
Conditional OR	`||`	Given *operand1* `||` *operand2*, where each operand must be of Boolean type, return true if at least one operand is true. Otherwise, return false. If *operand1* is true, *operand2* is not examined. This is known as *short-circuiting*.	2
Division	`/`	Given *operand1* `/` *operand2*, where each operand must be of character or numeric type, divide *operand1* by *operand2* and return the quotient.	11
Equality	`==`	Given *operand1* `==` *operand2*, where both operands must be comparable (you cannot compare an integer with a string literal, for example), compare both operands for equality. Return true if these operands are equal. Otherwise, return false.	7
Inequality	`!=`	Given *operand1* `!=` *operand2*, where both operands must be comparable (you cannot compare an integer with a string literal, for example), compare both operands for inequality. Return true if these operands are not equal. Otherwise, return false.	7

Operator	Symbol	Description	Precedence
Left shift	<<	Given *operand1* << *operand2*, where each operand must be of character or integer type, shift *operand1*'s binary representation left by the number of bits that *operand2* specifies. For each shift, a 0 is shifted into the rightmost bit and the leftmost bit is discarded. Only the five low-order bits of *operand2* are used when shifting a 32–bit integer (to prevent shifting more than the number of bits in a 32–bit integer). Only the six low-order bits of *operand2* 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 is equivalent to (but faster than) multiplying by a multiple of 2.	9
Logical AND	&	Given *operand1* & *operand2*, where each operand must be of Boolean type, return true if both operands are true. Otherwise, return false. In contrast to conditional AND, logical AND does not perform short-circuiting.	6
Logical complement	!	Given !*operand*, where *operand* must be of Boolean type, flip *operand*'s value (true to false or false to true) and return the result.	12
Logical exclusive OR	^	Given *operand1* ^ *operand2*, where each operand must be of Boolean type, return true if one operand is true and the other operand is false. Otherwise, return false.	5
Logical inclusive OR	|	Given *operand1* | *operand2*, where each operand must be of Boolean type, return true if at least one operand is true. Otherwise, return false. In contrast to conditional OR, logical inclusive OR does not perform short-circuiting.	4
Member access	.	Given *identifier1*.*identifier2*, access the *identifier2* member of *identifer1*. You will learn about this operator later in this chapter.	13
Method call	()	Given *identifier*(*argument list*), call the method identified by *identifier* and matching parameter list. You will learn about calling methods later in this chapter.	13
Multiplication	*	Given *operand1* * *operand2*, where each operand must be of character or numeric type, multiply *operand1* by *operand2* and return the product.	11
Object creation	new	Given new *identifier*(*argument list*), allocate memory for object and call constructor specified as *identifier*(*argument list*). Given new *identifier*[*integer size*], allocate a one-dimensional array of values.	12
Postdecrement	--	Given *variable*--, where *variable* must be of character or numeric type, subtract 1 from *variable*'s value (storing the result in *variable*) and return the original value.	13

Operator	Symbol	Description	Precedence
Postincrement	++	Given *variable*++, where *variable* must be of character or numeric type, add 1 to *variable*'s value (storing the result in *variable*) and return the original value.	13
Predecrement	--	Given --*variable*, where *variable* must be of character or numeric type, subtract 1 from its value, store the result in *variable*, and return this value.	12
Preincrement	++	Given ++*variable*, where *variable* must be of character or numeric type, add 1 to its value, store the result in *variable*, and return this value.	12
Relational greater than	>	Given *operand1* > *operand2*, where each operand must be of character or numeric type, return true if *operand1* is greater than *operand2*. Otherwise, return false.	8
Relational greater than or equal to	>=	Given *operand1* >= *operand2*, where each operand must be of character or numeric type, return true if *operand1* is greater than or equal to *operand2*. Otherwise, return false.	8
Relational less than	<	Given *operand1* < *operand2*, where each operand must be of character or numeric type, return true if *operand1* is less than *operand2*. Otherwise, return false.	8
Relational less than or equal to	<=	Given *operand1* <= *operand2*, where each operand must be of character or numeric type, return true if *operand1* is less than or equal to *operand2*. Otherwise, return false.	8
Relational type checking	instanceof	Given *operand1* instanceof *operand2*, where *operand1* is an object and *operand2* is a class (or other user-defined type), return true if *operand1* is an instance of *operand2*. Otherwise, return false.	8
Remainder	%	Given *operand1* % *operand2*, where each operand must be of character or numeric type, divide *operand1* by *operand2* and return the remainder.	11
Signed right shift	>>	Given *operand1* >> *operand2*, where each operand must be of character or integer type, shift *operand1*'s binary representation right by the number of bits that *operand2* 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 five low-order bits of *operand2* are used when shifting a 32–bit integer (to prevent shifting more than the number of bits in a 32–bit integer). Only the six low-order bits of *operand2* 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 is equivalent to (but faster than) dividing by a multiple of 2.	9

Operator	Symbol	Description	Precedence
String concatenation	+	Given *operand1* + *operand2*, where at least one operand is of `String` type, append *operand2*'s string representation to *operand1*'s string representation and return the concatenated result.	10
Subtraction	-	Given *operand1* - *operand2*, where each operand must be of character or numeric type, subtract *operand2* from *operand1* and return the difference.	10
Unary minus	-	Given -*operand*, where *operand* must be of character or numeric type, return *operand*'s arithmetic negative.	12
Unary plus	+	Like its predecessor, but return *operand*. Rarely used.	12
Unsigned right shift	>>>	Given *operand1* >>> *operand2*, where each operand must be of character or integer type, shift *operand1*'s binary representation right by the number of bits that *operand2* specifies. For each shift, a zero is shifted into the leftmost bit and the rightmost bit is discarded. Only the five low-order bits of *operand2* are used when shifting a 32-bit integer (to prevent shifting more than the number of bits in a 32-bit integer). Only the six low-order bits of *operand2* are used when shifting a 64-bit integer (to prevent shifting more than the number of bits in a 64-bit integer). The shift does not preserve negative values. Furthermore, it is equivalent to (but faster than) dividing by a multiple of 2.	9

In Table 2–3's operator descriptions, "integer type" refers to any of the byte integer, short integer, integer, or long integer types, 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's rightmost column presents a value that indicates the operator's *precedence* (level of importance): 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 when evaluating `40+2*4`.

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, `(40+2)*4` results in addition being performed before multiplication, and `40/(2-4)` results in subtraction being performed before divison.

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 `6*3/2` is evaluated as if it was `(6*3)/2` because * and / are left-to-right associative operators. In contrast, expression `a=b=c=10` is evaluated as if it was `a=(b=(c=10))`—`10` is assigned to `c`, `c`'s new value (10) is assigned to `b`, and `b`'s new value (10) is assigned to `a`—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.

Whenever an operator encounters two operands of different types, it attempts to convert one (and possibly both) of the operands to a type that is suitable for performing its operation. For example, if you attempt to add a 16-bit short integer value to a 32–bit integer value, the addition operator will first convert the short integer value to a 32–bit integer value, and then add together both 32–bit integer values.

Similarly, if you attempt to add a 32–bit integer value to a 32–bit floating-point value, the operator will first convert the 32–bit integer value to a 32–bit floating-point value, and then add together both floating-point values. The operator always converts the value with a more limited representation (such as a 16-bit short integer) to an equivalent value in a less limited representation (such as a 32–bit integer).

Sometimes, you need to explicitly perform a conversion, and this is where the cast operator comes into play. For example, suppose your class contains the following field declarations:

```
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 `'\u0123'`, which would not fit. Because of the potential for data loss, the compiler complains.

The solution to this problem involves introducing a `(byte)` cast, which explicitly tells the compiler that the developer is aware of the potential for data loss but wants the conversion to occur:

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

Not all conversions can be performed. For example, the subtraction operator cannot subtract a string literal from a 32–bit integer. Also, attempting to cast a floating-point value to a `String`, as in `String s = (String) 20.0;`, does not work. In these situations, the compiler provides suitable error messages.

The mathematical 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. Java does not detect overflows and underflows.

Dividing (via `/` or `%`) a numeric value by 0 also results in interesting behavior. In the first case, dividing an integer value by integer 0 causes the operator to throw an `ArithmeticException` object (I briefly discuss `ArithmeticException` in Chapter 4 when I cover exceptions). Secondly, dividing a 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).

Suppose you have created a `ReportWriter` class that outputs reports to the printer. During the testing phase, you want to make sure that reports are properly generated without wasting paper, so you would like the output to be directed to a file. An additional requirement is that you would like to be able to change the file's name or the printer's name from application code. Listing 2–7 presents a possible solution.

Listing 2–7. *Initializing the `ReportWriter` class's `outputDevice` field with the help of the conditional operator*

```
class ReportWriter
{
   static boolean test = true;
   static String outputDevice = (test) ? "file" : "printer";
}
```

Listing 2–7's `ReportWriter` class satisfies the first requirement by providing a `test` class field. When this class loads, `true` is assigned to `test`, overriding its default false value. The expression assigned to the `outputDevice` class field uses the conditional operator to examine `test`. Finding this field to contain true, the operator subsequently returns string literal `"file"`, which is assigned to `outputDevice`. If `test` is found to contain false, `"printer"` is returned and assigned to `outputDevice`.

To satisfy the second requirement, simply assign the appropriate destination string literal to `outputDevice` from the application's code.

Read-only Fields

Java provides reserved word `final` for declaring that a field is read-only.

Each object receives its own copy of a read-only instance field. This field must be initialized, as part of the field's declaration or in the class's constructor. If initialized in the constructor, the read-only instance field is known as a *blank final* because it does not have a value 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 not truly constants.

If you want a true *constant*, which is a single read-only value that is available to all objects, you need to create a read-only class field. You can accomplish this task by including the reserved word `static` with `final` in that field's declaration.

Listing 2–8 shows you how to declare a read-only class field.

Listing 2–8. *Declaring a true constant in the `Employee` class*

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

Listing 2–8's `RETIREMENT_AGE` declaration is an example of a *compile-time constant*. Because there is only one copy of its value (thanks to the `static` reserved word), and

because this value will never change (thanks to the `final` reserved word), the compiler is free to optimize the compiled code by inserting the constant value into all calculations where it is used. The code runs faster because it does not have to access a read-only class field.

Introducing Methods

After declaring a class, you can introduce *methods* (named bodies of code) into the class's body. Some of these methods can associate with objects in order to describe entity behaviors. Other methods can associate with classes. This section shows you how to declare methods, how to implement their bodies, and how to overload methods.

Declaring Methods

You can declare a method within a class's body by minimally specifying a return type, followed by an identifier that names the method, followed by a parameter list, followed by a body. Listing 2–9 presents a simple method declaration.

Listing 2–9. *Declaring a `printBalance()` method in the `CheckingAccount` class*

```
class CheckingAccount
{
   String owner;
   int balance;
   static int counter;
   void printBalance()
   {
      // code that outputs the balance field's value
   }
}
```

Listing 2–9 declares a method named `printBalance` in `CheckingAccount`'s body. By convention, a method's name begins with a lowercase letter. Furthermore, the first letter of each subsequent word in a multiword method name is capitalized.

A method's *parameter list* identifies the number, order, and types of values that are passed to the method when the method is called via a parentheses-delimited and comma-separated list of variable declarations. Each value passed to the method is called an *argument*; the variable that receives the argument is called a *parameter*. Because arguments are not passed to `printBalance()`, its parameter list is empty.

> **NOTE:** The method's name and the types of its parameters are known as its *signature*.

The method's body specifies code that is to be executed when the method is called. In this example, I chose to leave the body empty, but have supplied a single-line comment to describe what the body should contain.

Finally, a method's *return type* identifies the kind of values returned by the method. Because `printBalance()` does not return a value, its return type is set to void via

reserved word void. (Think of void as a special kind of primitive type, even though variables cannot be declared to be void, which is why I did not include it in Table 2–1.)

Because a CheckingAccount class should also provide methods for making deposits and withdrawals, Listing 2–10 introduces deposit() and withdraw() methods into this class.

Listing 2–10. *Declaring deposit() and withdraw() methods*

```
class CheckingAccount
{
   String owner;
   int balance;
   static int counter;
   void printBalance()
   {
      // code that outputs the balance field's value
   }
   int deposit(int amount)
   {
      // code that adds the specified amount to balance, and returns the new balance
   }
   int withdraw(int amount)
   {
      /* code that subtracts the specified amount from balance, and returns the new
         balance */
   }
}
```

Listing 2–10's deposit() and withdraw() methods each specify a nonempty parameter list consisting of a single amount parameter. As with any parameter, amount's lifetime ranges from the point where execution enters the method to the point where execution exits the method and returns to the method's *caller* (the code that called the method). Also, amount's scope is the entire method.

The deposit() and withdraw() methods also specify, via their int return types, that they return integer values.

Because we are not yet ready to code these methods' bodies, I have introduced comments to document their tasks. In contrast to printBalance() and deposit(), withdraw() reveals a *multiline comment*, one or more lines of documentation starting with /* and ending with */. The compiler ignores everything from /* through */.

> **CAUTION:** You cannot place a multiline comment inside another multiline comment: `/*/* Nesting multiline comments is illegal! */*/`

CheckingAccount's printBalance(), deposit(), and withdraw() methods are examples of instance methods because they associate with objects. Each method is called with a hidden argument that refers to the current object. The hidden argument allows the method to access the object's instance fields and call other instance methods. Furthermore, instance methods can access class fields and call class methods.

In many situations, instance methods are all that you need. However, when writing an application, you must introduce at least one *class method*, a method that associates with a class instead of with that class's objects, into one of the application's classes.

Specifically, an application must specify `public static void main(String[] args)` to serve as the application's entry point. The `static` reserved word is what makes this method a class method. (I will explain reserved word `public` later in this chapter.)

> **NOTE:** `main()` is called with an array of `String` objects, where each object contains the character representation of a command-line argument.

Because class methods are not called with a hidden argument that refers to the current object, `main()` cannot access an object's instance fields or call its instance methods. This method can only access class fields and call class methods.

Many methods 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. To declare a method that takes a variable number of arguments, specify three consecutive periods after the type name of the method's rightmost parameter. For example, Listing 2–11 places these periods after the `double` type name.

Listing 2–11. *Declaring a `sum()` method that takes a variable number of arguments*

```
double sum(double... values)
{
   // code that sums the passed double arguments and returns the sum
}
```

You will shortly learn how to codify this method, along with the methods that were previously introduced. Later on, you will learn how to call all of these methods.

Implementing Methods

Listing 2–10's empty `printBalance()` method accomplishes nothing. Furthermore, its empty `deposit()` and `withdraw()` methods result in compiler errors because these methods have non-void return types and do not return values. To make these methods useful and legal, you must introduce statements into their bodies.

A *statement* is one or more instructions that perform a task. It can be expressed as a simple statement or as a compound statement.

A *simple statement* is a single instruction terminated by a semicolon. In addition to the empty simple statement, Java supplies local variable declaration, assignment, method-call, decision, *loop* (repeated execution), break and continue, and method-return simple statements. (Chapter 4 introduces additional kinds of simple statements.)

A *compound statement* is a (possibly empty) sequence of simple and other compound statements sandwiched between open and close brace delimiters. A method body is an example. Compound statements can appear wherever simple statements appear, and are alternatively referred to as *blocks*.

Empty Statements, Local Variable Declarations, Assignments, and Method Calls

An *empty statement* is nothing followed by a semicolon. Although this kind of statement appears to be useless, it is useful, as I will explain when I discuss loop statements.

A *local variable declaration* introduces a variable into a method's body or other compound statement. This *local variable*'s lifetime and scope range from its point of declaration to the end of the compound statement in which it is declared. Only statements appearing after the declaration can access the local variable.

> **NOTE:** A local variable is similar to a parameter in that both kinds of variables only exist during a method's execution. However, a parameter contains a passed argument, whereas a local variable contains whatever value is necessary to help the method accomplish its task.

This statement is similar to an instance field declaration in that it minimally begins with a type name, continues with an identifier that names the local variable, and ends with a semicolon. By convention, a local variable's name begins with a lowercase letter. Furthermore, the first letter of each subsequent word in a multiword local variable name is capitalized.

> **CAUTION:** You cannot declare multiple local variables with the same name in the same scope because the compiler will report an "already defined" error message.

An *assignment* is a close relative of the local variable declaration statement. Instead of starting with a type, this statement begins with the name of a variable that has already been declared, continues with the assignment operator (=) or a compound assignment operator (such as +=), and concludes with an expression and a semicolon.

A *method call* executes a method, possibly passing arguments and possibly receiving a value in return. Regardless of whether or not a value is returned, a method call is a statement. If you do not assign the method's return value to a variable, the return value is lost. However, you occasionally might be more interested in what the method accomplishes than in what the method returns.

Listing 2–12's `printBalance()` method demonstrates these statements except for empty.

Listing 2–12. *Declaring and using local variables in* `printBalance()`

```
void printBalance()
{
   int magnitude = (balance < 0) ? -balance : balance;
   String balanceRep = (balance < 0) ? "(" : "";
   balanceRep += magnitude;
   balanceRep += (balance < 0) ? ")" : "";
   System.out.println(balanceRep);
}
```

Listing 2–12 introduces several statements into `printBalance()` to create a suitable string-based representation of the `balance` field's value and output this string. If this value is positive, it is represented and printed as is. If this value is negative, its *magnitude* (absolute or positive value) is represented and printed between parentheses.

This listing's first two statements declare a pair of local variables: `magnitude` and `balanceRep`. An expression initializes the `magnitude` variable to the `balance` field's magnitude, whereas another expression initializes `balanceRep` to `"("` or `""` (the empty string), depending on whether the `balance` field's value is negative or positive.

Unlike fields, which are initialized to default values, and parameters, which are initialized to arguments, local variables are not implicitly initialized. Before you can read a local variable's value, you must assign a value to the variable. Otherwise, the compiler reports an error about the local variable not having been initialized.

Moving on, two assignment statements follow the local variable declaration statements. These statements append a string representation of the `balance` field's magnitude followed by `")"` or `""` to `balanceRep`. Alternatively, I could have expressed the first of these statements as `balanceRep = balanceRep + magnitude;`.

Listing 2–12 concludes by calling the `System.out.println()` method to output `balanceRep`'s character sequence to the standard output device. (Chapter 1 briefly introduces the concept of standard input/output along with `System.out.println()` and related methods.)

Decisions

Listing 2–12 used the conditional operator to determine whether to assign `-balance` or `balance` to `magnitude`. Although this operator is useful for initializing a variable to one of two values, it cannot be used for choosing between two statements to execute. Java supplies the if-else statement for this purpose.

The if-else statement 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 statement to execute (if the Boolean expression evaluates to true), followed by reserved word `else`, followed by another statement to execute (if the Boolean expression evaluates to false).

Listing 2–13 demonstrates if-else.

Listing 2–13. *A revised `printBalance()` method*

```
void printBalance()
{
   if (balance < 0)
      System.out.println("(" + -balance + ")");
   else
```

```
        System.out.println(balance);
}
```

Listing 2–13's if-else statement results in the first `System.out.println()` method call executing if `balance`'s value is less than 0, and the second `System.out.println()` method call executing if `balance`'s value is greater than or equal to 0.

Each of *`statement1`* and *`statement2`* describes another statement to execute. If you do not need the else part in the preceding syntax, you can omit `else` and *`statement2`* from the syntax. The resulting statement is called if.

NOTE: When if and if-else are used together, and the source code is not properly indented, it can be difficult to determine which if associates with the else. 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? For example:

```
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*.

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

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

If the first Boolean expression is true, *statement1* executes. Otherwise, if the second Boolean expression is true, *statement2* executes. This pattern continues until one of these expressions is true and its corresponding statement executes, or the final `else` is reached and *statementN* (the default statement) executes.

Listing 2–14 demonstrates chained if-else.

Listing 2–14. *A revised `printBalance()` method using chained if-else*

```
void printBalance()
{
   if (balance < 0)
      System.out.println("(" + -balance + ")");
   else
   if (balance == 0)
      System.out.println("zero balance");
   else
      System.out.println(balance);
}
```

Look closely at Listing 2–14 and you will see that its chained if-else statement is actually an if-else statement, where the statement following the else part (the first `else`) is another if-else statement.

Chaining if-else statements together leads to verbosity that, in some cases, can be made more concise by using a switch statement. This statement lets you write code for choosing one of several statements to execute, and has the following syntax:

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

The switch statement consists of reserved word `switch`, followed by a *selector expression* in parentheses, followed by a body of cases. The selector expression is

typically any expression that evaluates to an integral value. For example, it might evaluate to a 32–bit integer or to a 16-bit character.

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 I have yet to discuss).

After evaluating the selector expression, switch compares this value with each case's value until it finds a match. If there is a match, the case's statement is executed. For example, if 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 will 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 such a 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.

Listing 2–15 demonstrates switch.

Listing 2–15. *Using switch to output a compass direction*

```
class Compass
{
   static final int NORTH = 0;
   static final int SOUTH = 1;
   static final int WEST = 2;
   static final int EAST = 3;
   void printDirection(int dir)
   {
      switch (dir)
      {
         case NORTH: System.out.println("You are travelling north."); break;
         case SOUTH: System.out.println("You are travelling south."); break;
         case EAST : System.out.println("You are travelling east."); break;
         case WEST : System.out.println("You are travelling west."); break;
         default   : System.out.println("Unknown direction");
      }
   }
}
```

Listing 2–15's `Compass` class is an example of an *enumerated type*, a named sequence of related constants. `NORTH`, `SOUTH`, `EAST`, and `WEST` are `Compass`'s set of constants.

Java version 5 introduced the enum as an improved enumerated type that overcomes problems with the listing's form of enumerated type. This feature includes a change to the switch statement, which I will discuss when I cover enums in Chapter 5.

> **NOTE:** Java version 7 introduces the ability to switch on a string-based selector expression. In this situation, each case's value is a string literal. I will demonstrate this form of the switch statement in the next section.

Loops

It is sometimes necessary to execute a statement repeatedly. This repeated execution is called a *loop*.

Java provides three kinds of loop statements: for, while, and do-while.

The for statement 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 initialization section, followed by an optional test section, followed by an optional update section. A non-optional semicolon separates each of the first two sections from the next section.

The initialization section consists of a comma-separated list of local 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.

Listing 2–16 uses the for statement to iterate over the array of command-line arguments that is passed to the `main()` method. Each argument is read from the array, and Java version 7's enhanced switch statement uses the argument to determine a course of action.

Listing 2–16. *Using for with switch on a string-based selector expression to process command-line arguments*

```
public static void main(String[] args)
{
   for (int i = 0; i < args.length; i++)
      switch (args[i])
      {
         case "-v":
         case "-V": System.out.println("version 1.0");
                    break;
         default  : showUsage();
      }
}
```

Listing 2–16's for statement presents an initialization section that declares local variable `i`, a test section that compares `i`'s current value to the length of the `args` array (every array has a `length` field that returns the number of elements in the array) to ensure that this value is less than the array's length, and an update section that increments `i` by 1. The loop continues until `i`'s value equals the array's length.

Each *iteration* (loop execution) accesses one of the array's values via the `args[i]` expression. This expression returns the array's `i`th value (which happens to be a `String` object in this example). The first value is stored in `args[0]`.

The `args[i]` expression serves as the switch statement's selector expression. If this `String` object contains `-V`, the second case is executed, which calls `System.out.println()` to output a version number message. The subsequent break statement keeps execution from falling into the default case, which calls `showUsage()` to output usage information when `main()` is called with unexpected arguments.

If this `String` object contains `-v`, the lack of a break statement following the first case causes execution to fall through to the second case, calling `System.out.println()`. This example demonstrates the occasional need to group cases to execute common code.

The while statement has the following syntax:

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

This statement consists of reserved word `while`, followed by a parenthesized Boolean expression header, followed by a statement to repeatedly execute.

The while statement first evaluates the Boolean expression. If it is true, while executes the other statement. Once again, the Boolean expression is evaluated. If it is still true, while re-executes 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. Listing 2–17 provides a demonstration.

Listing 2–17. *Prompting the user to enter a specific character via a while statement*

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

```
   ch = System.in.read();
}
```

Listing 2–17 begins by initializing local 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, prior to the comparison.)

If ch does not contain C (which it does not 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 either the C key or the c key. It next reads the entered character via `System.in.read()` (discussed in Chapter 1), saving the character's integer value in ch.

Following 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. Seeing 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. Seeing 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++;
}
```

The do-while statement has the following syntax:

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

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

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

Listing 2–18 demonstrates do-while in another example of prompting the user to enter a specific uppercase letter or its lowercase equivalent.

Listing 2–18. *Prompting the user to enter a specific character via a do-while statement*

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

Listing 2–18 is similar to Listing 2–17. This time, however, the compound statement is executed prior to the test. As a result, it is no longer necessary to initialize ch—ch is assigned System.in.read()'s return value prior to the Boolean expression's evaluation.

It is sometimes useful for a loop statement to execute the empty statement repeatedly. The actual work performed by the loop statement takes place in the statement header. Listing 2–19 presents an example.

Listing 2–19. *Reading and outputting lines of text*

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

Listing 2–19 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 code fragment is supposed to output Hello on ten lines. Instead, only one instance of this string appears—the empty statement is executed ten times:

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

Break and Continue

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 is not desirable from the point of view of your application's users.

> **CAUTION:** An infinite loop can also arise from a loop header's Boolean expression comparing a floating-point value against a nonzero value via the equality or inequality operator, because many floating-point values have inexact internal representations. For example, the following code fragment never ends because `0.1` does not 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 is 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.

Listing 2–20 uses break with an if decision statement to exit a `while(true)`-based infinite loop when the user presses the C or c key.

Listing 2–20. *Breaking out of an infinite loop*

```
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 2–21 reveals this scenario.

Listing 2–21. *Prematurely breaking out of a for-based loop*

```
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 2–21 uses for and if to search an array of employee IDs to determine if a specific employee ID exists. If this ID is found, if's compound statement assigns `true` to `found`. Because there is no point in continuing the search, it then uses break to quit the loop.

The continue statement skips the remainder of the current loop iteration, reevaluates the header'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 should not process blank lines, while skips the current iteration when a blank line is detected, as demonstrated in Listing 2–22.

Listing 2–22. *Skipping the remainder of the current iteration*

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

Listing 2–22 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.

Look carefully at Listing 2–22 and you should realize that the continue statement is not needed. Instead, this listing can be shortened via *refactoring* (rewriting source code to improve its readability, organization, or reusability), as demonstrated in Listing 2–23.

Listing 2–23. *A refactored if statement*

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

Listing 2–23'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()`.

Unlike break, continue does not appear to be a necessary part of the language. As you have just seen, it is possible to remove continue by inverting the Boolean expression. If you find yourself relying too much on continue, perhaps you need to refactor your code.

Java provides labeled versions of break and continue as disciplined versions of *goto*, a statement that transfers execution to a labeled statement. (Many developers hate goto because undisciplined use of this statement results in unreadable/unmaintainable code.)

> **NOTE:** Java reserves the `goto` identifier so that it cannot be used to name any source code entity. However, GOTO is not reserved, so it seems pointless to reserve `goto`.

The labeled break statement consists of `break`, followed by an identifier for which a matching *label* (an identifier followed by a colon) must exist. Furthermore, the label must immediately precede a loop statement.

Labeled break is useful for breaking out of *nested loops* (loops within loops). For example, Listing 2–24 reveals the labeled break statement transferring execution to the first statement that follows the outer for loop.

Listing 2–24. *Breaking out of nested for loops*

```
outer:
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 3; j++)
      if (i == 1 && j == 1)
         break outer;
      else
         System.out.println("i=" + i + ", j=" + j);
System.out.println("Both loops terminated.");
```

When `i`'s value is `1` and `j`'s value is `1`, `break outer;` is executed to terminate both for loops. This statement transfers execution to the first statement after the outer for loop, which happens to be `System.out.println("Both loops terminated.");`.

The following output is generated:

```
i=0, j=0
i=0, j=1
i=0, j=2
i=1, j=0
Both loops terminated.
```

The labeled continue statement consists of `continue`, followed by an identifier for which a matching *label* (an identifier followed by a colon) must exist. Furthermore, the label must immediately precede a loop statement.

Labeled continue is useful for terminating the current and future iterations of nested loops and beginning a new iteration of the labeled loop. For example, Listing 2–25 reveals the labeled continue statement terminating the inner for loop's iterations.

Listing 2–25. *Continuing the outer for loop*

```
outer:
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 3; j++)
      if (i == 1 && j == 1)
         continue outer;
      else
         System.out.println("i=" + i + ", j=" + j);
System.out.println("Both loops terminated.");
```

When `i`'s value is `1` and `j`'s value is `1`, `continue outer;` is executed to terminate the inner for loop and continue with the outer for loop at its next value of `i`. Both loops continue until they finish.

The following output is generated:

```
i=0, j=0
i=0, j=1
i=0, j=2
i=1, j=0
i=2, j=0
i=2, j=1
i=2, j=2
Both loops terminated.
```

Method Return

A void method's execution flows from its first statement to its last statement. However, Java's return statement lets a method exit prior to the last statement. As Listing 2–26 reveals, this statement consists of reserved word `return` followed by a semicolon.

Listing 2–26. *Returning from a method*

```
class Employee
{
   int salary;
   void setSalary(int empSalary)
   {
      if (empSalary < 0)
      {
         System.out.println("salary cannot be negative");
         return;
      }
      salary = empSalary;
   }
}
```

Listing 2–26's `setSalary()` method uses an if statement to detect an attempt to assign a negative value to the `salary` field. In this case, an error message is output and return prematurely exits the method so that the negative value cannot be assigned.

This form of the return statement is not legal in a method that returns a value. For such methods, Java provides a version of return that lets the method return a value (whose type must match the method's return type). Listing 2–27 demonstrates this version.

Listing 2–27. *Returning a value from a method*

```
int deposit(int amount)
{
   if (amount <= 0)
   {
      System.out.println("cannot deposit a negative or zero amount");
      return balance;
   }
   balance += amount;
   return balance;
}
```

Listing 2–27's `deposit()` method uses an if statement to detect an attempt to assign a negative or zero value to the `balance` field (of the previously presented `CheckingAccount` class), and outputs an error message when this attempt is detected. Furthermore, it returns `balance`'s current value. If there is no problem, `balance` is updated and its new value is returned.

I previously declared a `sum()` method whose parameter list indicates that this method takes a variable number of arguments. I did not present a body for this method because I had not covered statements (especially return). Now that you have been introduced to the return statement, take a look at Listing 2–28.

Listing 2–28. *Declaring the body of a variable-argument `sum()` method*

```
double sum(double... values)
{
   int total = 0;
   for (int i = 0; i < values.length; i++)
      total += values[i];
   return total;
}
```

Listing 2–28's implementation totals the number of arguments passed to this method. (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.

Overloaded 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 call, it compares the called method's arguments list with each overloaded method's parameter list as it looks for the correct method to call.

Two same-named methods are overloaded if their parameter lists differ in number or order of parameters. For example, Java's `String` class provides overloaded `public int indexOf(int ch)` and `public int indexOf(int ch, int fromIndex)` methods. These methods differ in parameter counts. (I explore `String` in Chapter 7.)

Two same-named methods are overloaded if at least one parameter differs in type. For example, Java's `Math` class provides overloaded `public static double abs(double a)`

and `public static int abs(int a)` methods. One method's parameter is a `double`; the other method's parameter is an `int`. (I explore `Math` in Chapter 6.)

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 does not have enough information to choose which method to call when it encounters `sum(1.0, 2.0)` in source code.

Introducing Constructors

Constructors are named blocks of code, declared in class bodies, for constructing objects by initializing their instance fields and performing other initialization tasks.

A constructor declaration specifies the same name as the class and a (potentially empty) parameter list. However, it does not specify a return type. A return type is not necessary because the result of a constructor call is always a newly created object whose type is the constructor's class. Listing 2–29 shows you how to declare a constructor.

Listing 2–29. *Introducing a constructor into the* `CheckingAccount` *class*

```
class CheckingAccount
{
   String owner;
   int balance;
   static int counter;
   CheckingAccount(String acctOwner, int acctBalance)
   {
      owner = acctOwner;
      balance = acctBalance;
      counter++; // keep track of created CheckingAccount objects
   }
}
```

Listing 2–29 declares a two-parameter constructor. When this constructor is called during object creation, `acctOwner` and `acctBalance` arguments are passed to this constructor's parameters, and subsequently assigned to `CheckingAccount`'s `owner` and `balance` instance fields.

> **CAUTION:** You cannot include the reserved word `static` in a constructor declaration because constructors are used to initialize objects—you can initialize class fields from within a constructor, but assigning an expression to a class field would probably make more sense. You cannot include the reserved word `final` in a constructor declaration because constructors cannot be inherited. (I discuss inheritance in the next chapter.)

Constructors can be overloaded, just like regular methods. For example, Listing 2–30 reveals a second `CheckingAccount` constructor.

Listing 2–30. *An overloaded* `CheckingAccount` *constructor*

```
CheckingAccount(String acctOwner)
{
   owner = acctOwner;
   balance = 100;
   counter++; // keep track of created CheckingAccount objects
}
```

Listing 2–30's constructor declaration specifies a single `acctOwner` parameter, which is assigned to the `owner` field. Furthermore, it assigns `100` to `balance`. (Perhaps this second constructor is called when creating a new checking account, where the initial balance must default to a minimum of $100.)

Instead of duplicating constructor code, you can have the overloaded constructor call a previously declared constructor to reuse existing code. Listing 2–31 provides a demonstration.

Listing 2–31. *Calling a* `CheckingAccount` *constructor from another* `CheckingAccount` *constructor*

```
CheckingAccount(String acctOwner)
{
   this(acctOwner, 100);
}
```

Listing 2–31's constructor declaration also specifies a single `acctOwner` parameter. Furthermore, it uses reserved word `this` to call the previous constructor with `acctOwner`'s value and `100`.

> **CAUTION:** You must use `this` to call another constructor—you cannot use the class's name, as in `CheckingAccount()`. The `this()` constructor call (if present) must be the first statement that is 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 non-constructor method—constructors can be called only by other constructors and during object creation.

If a class does not declare a constructor, the class is assigned a default noargument constructor; this constructor is called during object creation. In addition to taking no arguments, this constructor's body is empty: it contains no initialization code for initializing objects. There is no default constructor when a constructor is declared.

Introducing Other Initializers

You previously learned how to initialize fields by assigning expressions to them. These kinds of *initializers* are known as *class field initializers* and *instance field initializers*. You also learned how to use constructors to initialize instance fields. Additionally, Java supports class initializers and instance initializers.

Class Initializers

A *class initializer* is a `static`-prefixed compound statement that is introduced into a class body. It is used to initialize a loaded class via a sequence of statements. For example, I once used a class initializer to load a custom database driver class. Listing 2–32 shows the loading details.

Listing 2–32. *Loading a database driver via a class initializer*

```
class JDBCFilterDriver implements Driver
{
   static private Driver d;
   static
   {
      // Attempt to load JDBC-ODBC Bridge Driver and register that
      // driver.
      try
      {
         Class c = Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         d = (Driver) c.newInstance();
         DriverManager.registerDriver(new JDBCFilterDriver());
      }
      catch (Exception e)
      {
         System.out.println(e);
      }
   }
   //...
}
```

Listing 2–32's `JDBCFilterDriver` class uses its class initializer to load and instantiate the class that describes Java's JDBC-ODBC Bridge Driver, and to register a `JDBCFilterDriver` instance with Java's database driver. Although much of this listing is probably meaningless to you right now, it illustrates the usefulness of class initializers.

Instance Initializers

An *instance initializer* is a compound statement that is introduced into a class body, as opposed to being introduced into 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 2–33.

Listing 2–33. *Initializing a pair of arrays via an instance initializer*

```
class Graphics
{
   double[] sines = { 0.0, 0.0, /* ... */ 0.0 };   // should be 360 entries
   double[] cosines = { 0.0, 0.0, /* ... */ 0.0 }; // should be 360 entries
   {
      for (int i = 0; i < sines.length; i++)
      {
         sines[i] = Math.sin(Math.toRadians(i));
         cosines[i] = Math.cos(Math.toRadians(i));
      }
   }
}
```

Listing 2–33's `Graphics` class uses an instance initializer to initialize an object's `sines` and `cosines` arrays to the sines and cosines of angles ranging from 0 through (ideally) 360—not all array elements are present, for brevity. It is faster to read array elements than to repeatedly call `Math.sin()` and `Math.cos()` elsewhere; performance matters.

Because the code in Listing 2–33's instance initializer could just as easily have been placed in a constructor, what is the advantage in using an instance initializer? Apart from possibly clarifying source code, instance initializers are useful in anonymous classes (discussed Chapter 4), which cannot declare constructors.

Initialization Order

A class's body can contain multiple instance field initializers, instance class initializers, constructors, instance initializers, and class initializers. Furthermore, class fields and instance fields initialize to default values. Understanding the order in which all of this initialization occurs is important to preventing confusion, so check out Listing 2–34.

Listing 2–34. *A complete initialization demo*

```
// InitDemo.java

public class InitDemo
{
   static boolean bool1;
   boolean bool2;
   static byte byte1;
   byte byte2;
   static char char1;
   char char2;
   static double double1;
   double double2;
   static float float1;
   float float2;
   static int int1;
   int int2;
   static long long1;
   long long2;
   static short short1;
   short short2;
   static String string1;
   String string2;
   static
   {
      System.out.println("[class] bool1 = " + bool1);
      System.out.println("[class] byte1 = " + byte1);
      System.out.println("[class] char1 = " + char1);
      System.out.println("[class] double1 = " + double1);
      System.out.println("[class] float1 = " + float1);
      System.out.println("[class] int1 = " + int1);
      System.out.println("[class] long1 = " + long1);
      System.out.println("[class] short1 = " + short1);
      System.out.println("[class] string1 = " + string1);
      System.out.println();
   }
   {
```

```
        System.out.println("[instance] bool2 = " + bool2);
        System.out.println("[instance] byte2 = " + byte2);
        System.out.println("[instance] char2 = " + char2);
        System.out.println("[instance] double2 = " + double2);
        System.out.println("[instance] float2 = " + float2);
        System.out.println("[instance] int2 = " + int2);
        System.out.println("[instance] long2 = " + long2);
        System.out.println("[instance] short2 = " + short2);
        System.out.println("[instance] string2 = " + string2);
        System.out.println();
    }
    static
    {
        bool1 = true;
        byte1 = 127;
        char1 = 'A';
        double1 = 1.0;
        float1 = 2.0F;
        int1 = 1000000000;
        long1 = 10000000000000L;
        short1 = 32767;
        string1 = "abc";
    }
    {
        bool2 = true;
        byte2 = 127;
        char2 = 'A';
        double2 = 1.0;
        float2 = 2.0F;
        int2 = 1000000000;
        long2 = 10000000000000L;
        short2 = 32767;
        string2 = "abc";
    }
    InitDemo()
    {
        System.out.println("InitDemo() called");
        System.out.println();
    }
    static double double3 = 10.0;
    double double4 = 10.0;
    static
    {
        System.out.println("[class] double3 = " + double3);
        System.out.println();
    }
    {
        System.out.println("[instance] double4 = " + double3);
        System.out.println();
    }
    public static void main(String[] args)
    {
        System.out.println ("main() started");
        System.out.println();
        System.out.println("[class] bool1 = " + bool1);
        System.out.println("[class] byte1 = " + byte1);
        System.out.println("[class] char1 = " + char1);
```

```
        System.out.println("[class] double1 = " + double1);
        System.out.println("[class] double3 = " + double3);
        System.out.println("[class] float1 = " + float1);
        System.out.println("[class] int1 = " + int1);
        System.out.println("[class] long1 = " + long1);
        System.out.println("[class] short1 = " + short1);
        System.out.println("[class] string1 = " + string1);
        System.out.println();
        for (int i = 0; i < 2; i++)
        {
           System.out.println("About to create InitDemo object");
           System.out.println();
           InitDemo id = new InitDemo();
           System.out.println("id created");
           System.out.println();
           System.out.println("[instance] id.bool2 = " + id.bool2);
           System.out.println("[instance] id.byte2 = " + id.byte2);
           System.out.println("[instance] id.char2 = " + id.char2);
           System.out.println("[instance] id.double2 = " + id.double2);
           System.out.println("[instance] id.double4 = " + id.double4);
           System.out.println("[instance] id.float2 = " + id.float2);
           System.out.println("[instance] id.int2 = " + id.int2);
           System.out.println("[instance] id.long2 = " + id.long2);
           System.out.println("[instance] id.short2 = " + id.short2);
           System.out.println("[instance] id.string2 = " + id.string2);
           System.out.println();
        }
     }
}
```

Listing 2–34's `InitDemo` class declares one class field and one instance field for each of the primitive types plus `String`. It also introduces one explicitly initialized class field, one explicitly initialized instance field, three class initializers, three instance initializers, and one constructor. If you compile and run this code, you will observe the following output:

```
[class] bool1 = false
[class] byte1 = 0
[class] char1 =  // A char value defaults to Unicode 0, which cannot be printed.
[class] double1 = 0.0
[class] float1 = 0.0
[class] int1 = 0
[class] long1 = 0
[class] short1 = 0
[class] string1 = null

[class] double3 = 10.0

main() started

[class] bool1 = true
[class] byte1 = 127
[class] char1 = A
[class] double1 = 1.0
[class] double3 = 10.0
[class] float1 = 2.0
[class] int1 = 1000000000
[class] long1 = 1000000000000
```

```
[class] short1 = 32767
[class] string1 = abc

About to create InitDemo object

[instance] bool2 = false
[instance] byte2 = 0
[instance] char2 =
[instance] double2 = 0.0
[instance] float2 = 0.0
[instance] int2 = 0
[instance] long2 = 0
[instance] short2 = 0
[instance] string2 = null

[instance] double4 = 10.0

InitDemo() called

id created

[instance] id.bool2 = true
[instance] id.byte2 = 127
[instance] id.char2 = A
[instance] id.double2 = 1.0
[instance] id.double4 = 10.0
[instance] id.float2 = 2.0
[instance] id.int2 = 1000000000
[instance] id.long2 = 10000000000000
[instance] id.short2 = 32767
[instance] id.string2 = abc

About to create InitDemo object

[instance] bool2 = false
[instance] byte2 = 0
[instance] char2 =
[instance] double2 = 0.0
[instance] float2 = 0.0
[instance] int2 = 0
[instance] long2 = 0
[instance] short2 = 0
[instance] string2 = null

[instance] double4 = 10.0

InitDemo() called

id created

[instance] id.bool2 = true
[instance] id.byte2 = 127
[instance] id.char2 = A
[instance] id.double2 = 1.0
[instance] id.double4 = 10.0
[instance] id.float2 = 2.0
[instance] id.int2 = 1000000000
```

```
[instance] id.long2 = 1000000000000
[instance] id.short2 = 32767
[instance] id.string2 = abc
```

As you study this output, you will discover some interesting facts about initialization:

- Class fields initialize to default values just after a class is loaded.
- All class initialization occurs prior to the `main()` method being called.
- Class initialization is performed in a top-down manner. (Attempting to access a class field prior to its declaration causes the compiler to report an illegal forward reference.)
- Instance fields initialize to default values at the start of object creation.
- All instance initialization occurs prior to a constructor returning.
- Instance initialization is performed in a top-down manner. (Attempting to access an instance field prior to its declaration causes the compiler to report an illegal forward reference.)

Interface Versus Implementation

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). *X* also provides an *implementation* (the code within exposed methods along with optional helper methods and optional supporting fields that should not be exposed) that codifies the interface. *Helper methods* are methods that assist exposed methods and should not 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 do not belong to the class's interface, so that you are 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.
- *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. (I will discuss packages in Chapter 4.)

- *Private*: A field, method, or constructor that is declared `private` cannot be accessed from beyond the class in which it is 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.

> **NOTE:** A class that is declared `public` must be stored in a file with the same name. For example, a `public Employee` class must be stored in `Employee.java`. A source file can only contain one `public` class.

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 2–35 demonstrates this pattern in the context of an `Employee` class declaration.

Listing 2–35. *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 2–35 presents an interface consisting of the `public` constructor and `public` setter/getter methods. The implementation consists of the `private` `name` field and constructor/method code.

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 are 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 has been determined that the first and last names will be accessed more often than the entire name. Listing 2–36 reveals these changes.

Listing 2–36. *Revising implementation without affecting existing interface*

```
class Employee
{
   private String firstName;
   private String lastName;
   Employee(String name)
   {
      setName(name);
   }
   Employee(String firstName, String lastName)
   {
      setName(firstName + " " + lastName);
   }
   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));
   }
   String getName()
   {
      return getFirstName() + " " + getLastName();
   }
   void setFirstName(String empFirstName)
   {
      firstName = empFirstName;
   }
   String getFirstName()
   {
      return firstName;
   }
   void setLastName(String empLastName)
   {
      lastName = empLastName;
   }
   String getLastName()
   {
      return lastName;
   }
}
```

Listing 2–36 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 `getLastName()` will be called more frequently than `getName()`, it is faster (in each case) to have the first two methods set/get `firstName`'s and `lastName`'s values rather than having to merge either value into/extract this value from `name`'s value.

Listing 2–36 also reveals `setName()` calling `setFirstName()` and `setLastName()`, and `getName()` calling `getFirstName()` and `getLastName()`, rather than directly accessing the `firstName` and `lastName` fields. Although avoiding direct access to these fields is not necessary in this example, imagine another implementation change that adds more

code to `setFirstName()`, `setLastName()`, `getFirstName()`, and `getLastName()`; 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 2–35 to that shown in Listing 2–36, because the original interface remains intact, although the interface has been extended. This lack of breakage results from hiding Listing 2–35's implementation, especially the `name` field.

TIP: Get into the habit of developing useful interfaces while hiding implementations because it will save you much trouble when maintaining your classes.

Objects

You previously learned that objects are instantiated from classes. You also discovered Java's `new` operator in Table 2–3, and observed brief examples of this operator's usage in Listings 2–32 and 2–34. This section explores object and array creation via `new`, and also focuses on accessing fields, calling methods, and garbage collection.

Creating Objects and Arrays

The `new` operator is used to create, from a class that has already been loaded into memory, an object whose instance fields default to zero values, which you interpret as literal value `false`, `'\u0000'`, `0`, `0L`, `0.0`, `0.0F`, or `null` (depending on field type). Furthermore, this operator calls the specified constructor to initialize instance fields to appropriate nondefault values and perform other kinds of initialization. For example, `new Employee("John", "Doe")` creates an object from Listing 2–36's `Employee` class, and calls its `Employee(String firstName, String lastName)` constructor to initialize this object's instance fields to the employee's first and last names.

The `new` operator creates the object in a special region of memory known as the *heap*. Furthermore, `new` returns this object's *reference* (a value that helps the Java virtual machine locate the heap-based object). One virtual machine implementation might implement references as physical memory addresses, whereas another implementation might use *handles* (numeric identifiers that are used to locate object addresses). It does not matter how the reference is implemented because it cannot be directly accessed. About the only thing you can do with a reference is to use `==`/`!=` to compare it with `null`, a special literal value that does not refer to any object.

NOTE: The reference that is returned by `new` is represented literally by reserved word `this`. Wherever `this` appears in source code, it represents the current object.

References are lost unless they are stored in fields or local variables, or passed as arguments to a method. For example, `Employee emp = new Employee("John", "Doe");` stores the `Employee` reference that `new` returns in `Employee` variable `emp` (the reference and variable types must agree). This variable is known as a *reference variable*.

> **NOTE:** The `String` class is treated specially by Java. For example, you have previously learned that you can assign a string literal to a `String` variable, as in `String s = "abc";`. Behind the scenes, however, Java treats this statement as a shorthand for `String s = new String("abc");`, in which a reference to a `String` object is assigned to `s`. (I will say more about this topic when I discuss `String` in Chapter 7.)

The `new` operator is also used to create a one-dimensional array of values in the heap. When creating the array, specify `new`, followed by a name that identifies the type of the values that are stored in the array, followed by an integral expression between a pair of square brackets that specifies the size of the array (the number of elements).

> **NOTE:** An array is implemented as a special Java object whose read-only `length` field contains the array's size.

For example, you can use `new` to create a one-dimensional array of object references, as demonstrated in Listing 2–37.

Listing 2–37. *Creating a one-dimensional array of `Employee` object references*

```
Employee[] empArray = new Employee[10];
```

When you create a one-dimensional array, `new` zeros the bits in each array element's storage location, which you interpret as literal value `false`, `'\u0000'`, `0`, `0L`, `0.0`, `0.0F`, or `null` (depending on element type). In the previous example, each of `empArray`'s elements is initialized to null.

You cannot access objects via this array until you assign nonnull object references to its elements, which Listing 2–38 demonstrates.

Listing 2–38. *Storing `Employee` object references in a one-dimensional array*

```
for (int i = 0; i < empArray.length; i++)
   empArray[i] = new Employee("John Doe #" + i);
```

> **NOTE:** You can combine `new` with the array initialization syntax if desired. For example, `Employee[] empArray = new Employee[] {new Employee ("John Doe")};` creates a single-element `Employee` array initialized to a single `Employee` object reference.

Suppose you want to use `new` to create a two-dimensional array, perhaps to assign to the `temperatures` field in Listing 2–3's `WeatherData` class. The way you accomplish this task in Java is to first create a one-dimensional row array (the outer array), and then

create a one-dimensional column array (the inner array) for each row, as shown in Listing 2–39.

Listing 2–39. *Creating a two-dimensional array*

```
// 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 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 contain row arrays—you must specify two pairs of empty brackets as part of the expression following `new`.)

After creating a two-dimensional array, you will want to populate its elements with suitable values. For example, Listing 2–40 initializes each `temperatures` element, which is accessed as `temperatures[row][col]`, to a random temperature value.

Listing 2–40. *Initializing a two-dimensional array*

```
for (int row = 0; row < temperatures.length; row++)
   for (int col = 0; col < temperatures[row].length; col++)
      temperatures[row][col] = Math.round(Math.random()*100);
```

You can subsequently output these values in a tabular format by calling Listing 2–41's for loop (this listing makes no attempt to align the temperature values in perfect columns).

Listing 2–41. *Outputting a two-dimensional array's values*

```
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();
}
```

Accessing Fields

When you codify a class, you can often specify the field's name as is in an expression. For example, Listing 2–12's `printBalance()` instance method specifies the name of `CheckingAccount`'s `balance` instance field in statement `int magnitude = (balance < 0) ? -balance : balance;`.

However, you cannot always access a field by simply specifying its name. For example, you cannot access a class field from another class in this manner. Also, you cannot access a field that is hidden from you. For example, you cannot access a `private` field from another class.

The following rules will help you learn how to access fields in different contexts, and even if such access is possible:

- Specify the name of a class field as is from anywhere within the same class as the class field. Example: `counter`
- Specify the name of the class field's class, followed by the member access operator (`.`), followed by the name of the class field from outside the class provided that its access control permits this form of access—the field is `public`, for example. Example: `CheckingAccount.counter`
- Specify the name of an instance field as is from any instance method, constructor, or instance initializer in the same class as the instance field. Example: `balance`
- 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 within the same class as the instance field, or from outside the class provided that its access control permits this form of access. Example: `CheckingAccount ca = new CheckingAccount(); int bal = ca.balance;`

CAUTION: Accessing an instance field (such as `ca.balance`) is usually not a good idea because the field exposes implementation details. However, if the class is declared as a private member of another class (a topic discussed in Chapter 4), and if its fields are frequently accessed by its enclosing class's methods, performance is often improved if these fields are accessed directly rather than being accessed via getter method calls. Although direct access opens up implementation details to the enclosing class, this is not a problem because these details are hidden to clients of the enclosing class.

Although the latter rule might seem to imply that you can access an instance field from a class context, this is not the case. Instead, you access the field from an object context.

The previous access rules are not exhaustive because there exist two more access scenarios to consider: declaring a local variable (or even a parameter) 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 find that you have declared a local variable or parameter that shadows a field, you can rename the local variable/parameter, or you can use the member access operator with reserved word `this` (instance field) or class name (class field) to explicitly identify the field. Listing 2–42 uses `this` to qualify an instance field name.

Listing 2–42. *Qualifying the `name` field to circumvent shadowing by a same-named parameter*

```
class Employee
{
   String name;
   void setName(String name)
   {
      this.name = name; // Assign the name argument to the name field.
```

```
    }
}
```

Listing 2–42's `setName()` method uses `this.name` to refer to the `name` field, not to `setName()`'s `name` parameter.

Calling Methods

When you codify a class, you can often specify the method's name as is in a method-call statement. For example, the constructor in Listing 2–35's `Employee` class specifies the name of `Employee`'s `setName()` method in method-call statement `setName(name);`.

However, you cannot always call a method by simply specifying its name. For example, you cannot call a class method from another class in this manner. Also, you cannot call a method that is hidden from you. For example, you cannot call a private method from another class.

The following rules will help you learn how to call methods in different contexts, and even if such calling is possible:

- Specify the name of a class method as is from anywhere within the same class as the class method. Example: `main(new String[0]); // No arguments are passed`
- 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 provided that its access control permits this form of access—the method is declared `public`, for example. Example: `Math.sin(Math.toRadians(45));`
- Specify the name of an instance method as is from any instance method, constructor, or instance initializer in the same class as the instance field. Example: `processLine(line);`
- 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 within the same class as the instance field, or from outside the class provided that its access control permits this form of access. Example: `CheckingAccount ca = new CheckingAccount("John Doe"); ca.printBalance();`

Although the latter rule might seem to imply that you can call an instance method from a class context, this is not the case. Instead, you call the method from an object context.

NOTE: Field access and method call rules are combined in statement `System.out.println();`, where the leftmost member access operator accesses the `out` class field (of type `PrintStream`) in the `System` class, and where the rightmost member access operator calls this field's `println()` method.

Method-Call Stack

Method calls require a special area of memory known as the *method-call stack*. You can think of the method-call stack as a simulation of a pile of clean trays in a cafeteria—you *pop* (remove) the clean tray off of the top of the pile and the dishwasher will *push* (insert) the next clean tray onto the top of the pile.

When a method is called, the virtual machine pushes its arguments and the address of the first statement to execute following the called 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.

Recursive Calls

A method normally executes statements that may include calls to other methods. However, it is 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! (the ! is the mathematical symbol for factorial) equals 3×2×1 or 6.

Your first approach to writing this method might consist of the code presented in Listing 2–43.

Listing 2–43. *A nonrecursive approach to obtaining a factorial*

```
int factorial(int n)
{
   int product = 1;
   for (int i = 2; i <= n; i++)
      product *= i;
   return product;
}
```

Although this code accomplishes its task, `factorial()` could also be written in Listing 2–44's recursive style.

Listing 2–44. *A recursive approach to obtaining a factorial*

```
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 Listing 2–44, 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*factorial(1)` now becomes 2*1 (2)

`3*factorial(2)` now becomes 3*2 (6)

`4*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 file system, 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 terminate.

Argument Passing

A method call includes a list of (potentially no) arguments being passed to the method. Arguments are passed using a style of argument passing that is known as *pass-by-value*, which is demonstrated in Listing 2–45.

Listing 2–45. *A demonstration of pass-by-value*

```
Employee emp = new Employee("John Doe");
int recommendedAnnualSalaryIncrease = 1000;
printReport(emp, recommendAnnualSalaryIncrease);
printReport(new Employee("Jane Doe"), 1500);
```

Pass-by-value passes the value of a variable (the reference value stored in `emp` or the 1000 value stored in `recommendedAnnualSalaryIncrease`, for example) or the value of some other expression (such as `new Employee("Jane Doe")` 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.

Chained Instance Method Calls

Two or more instance method calls can be chained together via the member access operator. For example, `new CheckingAccount().deposit(1000).printBalance();`. This capability lets you express code more compactly, and is also quite readable. However, you have to re-architect your methods somewhat differently, as Listing 2–46 reveals.

Listing 2–46. *Implementing instance methods so that calls to these methods can be chained together*

```
class CheckingAccount
{
   int balance;
   CheckingAccount deposit(int amount)
   {
      balance += amount;
      return this;
   }
   CheckingAccount printBalance()
   {
      int magnitude = (balance < 0) ? -balance : balance;
      String balanceRep = (balance < 0) ? "(" : "";
      balanceRep += magnitude;
      balanceRep += (balance < 0) ? ")" : "";
      System.out.println(balanceRep);
      return this;
   }
}
```

According to Listing 2–46, you must specify the name of the class as the method's return type. For example, each of `deposit()` and `printBalance()` must specify `CheckingAccount` as the return type. Also, you must specify `return this;` (return a reference to the current object) as the instance method's last statement.

Garbage Collection

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 solves this problem by using a *garbage collector*, code that runs in the background and checks for unreferenced objects. When it discovers an unreferenced object, the garbage collector removes it 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 `emp`'s method terminates.

An application can 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`.

Java's garbage collector eliminates a form of memory leakage in C++ implementations that do not rely on a garbage collector. In these implementations, the developer must destroy dynamically created objects before they go out of scope. If they vanish before destruction, they remain in the heap. Eventually, the heap fills and the application halts.

Although this form of memory leakage is not a problem in Java, a related form of leakage is problematic: continually creating objects and forgetting to remove even one reference to each object causes the heap to fill up and the application to eventually come to a halt.

This form of memory leakage is a major problem for applications that run for lengthy periods of time—a web server is one example. For shorter-lived applications, you will normally not notice this form of memory leakage. However, it is a good habit to assign `null` to reference variables when their referenced objects are no longer required.

EXERCISES

The following exercises are designed to test your understanding of Java's language fundamentals:

1. What does a class declaration contain?
2. Is `transient` a reserved word? Is `delegate` a reserved word?
3. What is a variable?
4. Identify Java's only unsigned primitive type.
5. What is the difference between an instance field and a class field?
6. What is an array?
7. How do you declare a one-dimensional array variable? How do you declare a two-dimensional array variable?
8. Define scope.
9. Is string literal `"The quick brown fox \jumps\ over the lazy dog."` legal or illegal? Why?
10. What is the purpose of the cast operator?
11. Which operator is used to create an object?
12. Can you nest multiline comments?

13. True or false: When declaring a method that takes a variable number of arguments, you must specify the three consecutive periods just after the rightmost parameter's type name.
14. Given a two-dimensional array `x`, what does `x.length` return?
15. What is the difference between the while and do-while statements?
16. Initialize the `sines` and `cosines` array declarations using the `new` syntax.
17. Why is it okay for an expression assigned to an instance field to access a class field that is declared after the instance field? In contrast, it is not okay for the expression to access another instance field that is declared after the instance field.
18. What is required to create an array of objects?
19. How do you prevent a field from being shadowed?
20. How do you chain together instance method calls?
21. 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 the *tail call*. Provide another example of tail recursion.
22. Merge the various `CheckingAccount` code fragments into a complete application.

Summary

Before you can write applications, you need to understand the fundamentals of the Java language. These fundamentals revolve around the concepts of class and object, where a class is nothing more than a template from which objects are created.

Classes are declared with reserved word `class` and a name. The class declaration's brace-delimited body is populated with a combination of field, method, and constructor declarations.

A field is a variable that stores a value. It initializes to a default value, although you can explicitly initialize it to a value with the help of an expression, which is a combination of variables, method calls, literals, and operators.

There are two kinds of fields: instance and class. Instance fields are used to describe entity attributes, whereas class fields are used to describe class attributes. A class field is differentiated from an instance field by being declared `static`.

A method is a named block of code that is called with an optional list of arguments, and possibly returns a value. Arguments are passed to parameters, which are another kind of variable. Parameters only exist while the method executes.

A method uses statements to perform tasks. A statement can be a simple statement, such as empty, local variable declaration, assignment, method-call, decision, loop, break and continue, and method-return.

A statement can be a compound statement, which contains simple and other compound statements that are placed between braces. A compound statement is also known as a block and can appear anywhere that a simple statement can appear.

A constructor is a named body of code (it is named after its class) that constructs objects by initializing their instance fields and performing other initialization tasks. Unlike methods, constructors do not have return types because they do not return anything.

Constructors (and methods) can be overloaded: they have the same names but different parameter lists. Constructors use `this()` to call other constructors in the same class. A class that does not declare a constructor is assigned a default noargument constructor.

In addition to letting you initialize fields by assigning expressions to them, and providing constructors for initializing objects, Java provides instance and class initializers to perform object initialization (when a constructor is not available) and class initialization.

Certain initialization rules are followed when multiple field initializers, constructors, instance initializers, and class initializers are mixed together in a class. For example, class initialization occurs prior to instance initialization.

Every class exposes an interface and also provides an implementation that codifies that interface. When designing a class, your goal is to expose a useful interface while hiding details of that interface's implementation. Hiding details is known as information hiding.

Java supports implementation hiding by providing four levels of access control: public, protected, private, and package-private. Except for package-private, each access control level is indicated via a same-named reserved word.

An object encapsulates an entity's attributes and behaviors as initialized fields and methods. The `new` operator creates an object from a class and calls its constructor to initialize these fields and perform other kinds of initialization.

The `new` operator creates the object in a special region of memory known as the heap. Furthermore, `new` returns this object's reference. Virtual machine implementations can implement references as physical memory addresses or as handles.

The `new` operator also creates a one-dimensional array of values in the heap. To create a two-dimensional array, create a one-dimensional row array and then, for each row array element, create a one-dimensional column array and assign its reference to the element.

There are rules for accessing fields. For example, to access a public class field from outside the class, you must specify the name of the class field's class, followed by the member access operator, followed by the class field's name.

A local variable or parameter shadows a field when it has the same name as the field. To access the field, either rename the local variable/parameter, or use the member access operator with reserved word `this` (instance field) or the class name (class field).

There are also rules for calling methods. For example, to call a public class method from outside the class, you must specify the name of a class method's class, followed by the member access operator, followed by the class method's name and argument list.

Method calls require a special area of memory known as the method-call stack. When a method is called, the virtual machine pushes its arguments and the address of the first statement to execute following the called method onto the method-call stack.

A method normally executes statements that may include calls to other methods. However, it is occasionally convenient to have a method call itself. This scenario is known as recursion.

A method call includes a list of (potentially no) arguments being passed to the method. Arguments are passed using a style of argument passing that is known as pass-by-value.

Two or more instance method calls can be chained together via the member access operator. Accomplish this task by having each method specify its class name as the method's return type, and by having the method return `this`.

Java uses a garbage collector to destroy objects. When it discovers an unreferenced object, the garbage collector removes it from the heap, making more heap space available.

Although the garbage collector eliminates the need to explicitly destroy objects, all references to objects that are no longer needed must be nullified, or else they will remain in the heap. This scenario is known as a memory leak.

Now that you know how to architect classes and work with objects, you are ready to write object-based applications. However, this knowledge is not enough to write object-oriented applications. To acquire that knowledge, you need to proceed to Chapter 3.

Index

Symbols and Numerics

$ boundary matcher, 441
^ boundary matcher, 441
? argument
  wildcards, generics, 209
@ prefix
  annotations, 190, 225
  Javadoc tags, 34
@interface syntax
  declaring annotation type, 193, 225
* symbol
  import statement, 153
%-prefixed character sequences, 509
  System.out.printf method, 510

A

\A boundary matcher, 441
abs method
  BigDecimal class, 236
  BigInteger class, 240
  Math class, 228, 231, 232
absolute pathnames, 451
  isAbsolute method, 453
abstract classes, 120–121, 133
Abstract Factory design pattern, 155
abstract methods, 120–121
abstract reserved word, 121
  interfaces, 126
abstract types, 133
Abstract Window Toolkit (AWT), 406, 408
AbstractExecutorService class, 386
accept method
  FileFilter interface, 458
  FilenameFilter interface, 457
access control levels, 82
access operators
  member access operator, 53
AccessibleObject class, 276
accessing fields, 87–89
acos method, Math, 228
activeCount method, ThreadGroup, 294
actual type arguments, 203, 205, 207
  array type, 203
  concrete parameterized type, 203
  concrete type, 203
  Enum class, 222
  type parameter, 203
  wildcards, 203, 209, 210
add method
  BigDecimal class, 236
  BigInteger class, 240
  Collection interface, 318
  List interface, 325
  ListIterator class, 327
  Queue interface, 346
  Set interface, 332
addAll method
  Collection interface, 319
  List interface, 325
addition operator, 51
after method, Date, 415, 416
agile software development, 131
aliases method, Charset, 517
allOf method, EnumSet, 339
and method, BitSet, 373
AND operators
  bitwise AND operator, 51
    BitSet class, 374
  conditional AND operator, 52
  logical AND operator, 53
  while statement, 68
andNot method, BitSet, 373
Android, 1
Android apps, 1, 41
Android platform, 5
Annotation interface, 274
annotation types

adding elements to, 194
declaring, 193–195
Deprecated, 191
Documented, 198
identifying applicable application elements, 196
Inherited, 198
Override, 191
Retention, 196
Stub, 193
SuppressWarnings, 191, 193, 196
Target, 196
annotations, 190–200, 225
@ prefix, 190
annotating source code, 193–195
annotating undesirable application elements, 197
getAnnotation method, 199, 276
getAnnotations method, 270, 274
getDeclaredAnnotations method, 270, 274
identifying retention/lifetime of, 196
isAnnotationPresent method, 199, 276
marker annotations, 194
meta-annotations, 196–198
processing, 198–200
anonymous classes, 146–148
enhancing enums, 219
APIs
Break Iterator, 409
Collator, 413
Internationalization, 397–428
Joda Time, 420
Math, 227–243
Preferences, 428–431
Properties, 429
References, 257–266, 268
Reflection, 269–277
Regular Expressions, 434–447
Threading, 287–311
apostrophe in patterns, 427
append method, PartsDB, 470
append methods
StringBuffer class, 282, 284
Writer class, 513
Appendable interface, 513
applets, Java SE, 5
application development
converting pseudocode to Java code, 23–36
Four of a Kind game, 20–40
modeling in pseudocode, 21–23
applications
Android apps, 1
PackageInfo, 244–247
stopping, 290
applyPattern method, MessageFormat, 428
apt tool, 200
argument passing
method calls, 91
arguments, methods, 58, 60
ArithmeticException, 165, 389
division by zero, 56, 294
Array class, 276
methods, 369–370
array index operator, 51
array type, generics, 203
ArrayBlockingQueue class, 392
arraycopy method, System, 285, 286
ArrayList class, 329–330
fail-fast behavior, 327
arrays
covariance, 211
creating objects and, 85–87
description, 47
enhanced for loop statement, 322
initializing array-based fields, 50
one-dimensional array, 47
reification, 211
shuffling array of integers, 434
toArray method, 320, 339
toCharArray method, 280
two-dimensional array, 47
Arrays class, 369–371, 434
asList method, 369
binarySearch method, 370
fill method, 370
sort method, 370
ArrayStoreException, 211, 212, 285, 320
ASCII
character sets and encodings, 512
asin method, Math, 228
asList method, Arrays, 369
assert reserved word, 182
AssertionError
declaring assertions, 182
internal invariants, 184
assertions, 181–190, 225
avoiding, 188–189
class invariants, 188
control-flow invariants, 184–185
declaring, 182–183

design-by-contract development, 185–188
enabling/disabling, 189–190
internal invariants, 183–184
postconditions, 187–188
preconditions, 185–187
preventing bugs, 181
throwing assertion error
with detail message, 183
without detail message, 182
using, 183–188
when not to use, 188
assignment, 61
assignment operator, 51
compound assignment operator, 52
associativity, operators, 55
atan method, Math, 228
atomic operations, 298
atomic variables, 397
AtomicLong class, 397
Atomic-prefixed classes, 397
@author, Javadoc tag, 35
autoboxing, 323–324
auto-detect charset, 517
automatic resource management, 176
available method, IntputStream, 476
availableCharsets method, 516
await methods, CountDownLatch, 390, 391
awaitTermination method, 383
AWT (Abstract Window Toolkit), 406
Swing, 408

B

\b boundary matcher, 441
Babel Fish, Yahoo!, 408
back reference, capturing groups, 441
backslash (\)
double backslash characters in pathnames, 450
escape sequence for, 49
representing root directory, Windows, 159
backspace
escape sequence for, 49
base class see superclass
before method, Date, 415, 416
BigDecimal class, 234–239, 268
constructors, 236, 239
convenience constants, 236
floating-point values, 234
methods, 236
round method, Math class, 234
ROUND_-prefixed constants, 236
superclass of, 257
big-endian format, 494
BigInteger class, 239–243, 268
constructors, 240
convenience constants, 240
factorials, 241–243
methods, 240
RSA encryption, 243
superclass of, 257
bin directory, JDK, 7
binary format
integer literals, 50
binary operator, 51
binary search, 370
binary strings
prepending zeros to, 256
toAlignedBinaryString, 256
toBinaryString method, 255
binarySearch method, Arrays, 370
Bitmap class
compress method, 479
BitmapFactory class
decodeFile method, 479
bits, 46
BitSet class, 373–376
methods, 373–374
bitsets, 337
EnumSet class, 338
variable-length bitsets, 376
BitStreamInputStream class, 492
BitStreamOutputStream class, 492
bitwise AND operator, 51, 374
bitwise complement operator, 51
bitwise exclusive OR operator, 51, 374
bitwise inclusive OR operator, 52, 374
Bloch's algorithm, 359
BLOCKED value, State enum, 288
BlockingQueue interface, 392, 393
classes implementing, 392
offer method, 396
poll method, 396
put method, 393, 396
take method, 393, 396
Boolean class, 248–250
Boolean literal, 49
Boolean type, 46
getBoolean method, 275
nextBoolean method, 433

portability of Java applications, 46
booleanValue method, 248, 249
boundaries
detecting logical boundaries within text, 409
boundary matchers, 441–442
zero-length matches, 442
bounds, generic types
recursive type bound, 208
type parameter bounds, 205–207
boxing
autoboxing and unboxing, 323–324
Break Iterator API, 409
break iterators, 409–413
using character-based break iterator, 411
break reserved word, 70
switch statement, 65
break statement, 65, 70
breaking out of finite loop, 70
breaking out of infinite loop, 70
labeled break statement, 72
BreakIterator class, 409
methods, 410–412
BrokenBarrierException, 390
Bubble Sort algorithm, 187
buckets
hash function, 357
hashtable capacity, 358
BufferedInputStream class, 492, 493
read method, 492, 493
BufferedOutputStream class, 492
write method, 492, 493
BufferedReader class, 522
readLine method, 522
buffers
StringBuffer class, 281–284
bugs, preventing, 181
byte cast, 56
Byte class, 255–257
Byte integer type, 46, 233
ByteArrayInputStream class, 477, 478
ByteArrayOutputStream class, 477, 478–479
toByteArray method, 478, 479
bytecode, 4
bytecode verifier, 4
bytes, 46, 511
getByte method, 275
nextBytes method, 433
skipBytes method, 465
byteValue method, 255

C

C/C++
compared with Java, 2
cache
clearCache methods, 403
SoftReference objects, 261–262
Calendar class, 417–420
clear method, 418
displayWeekdayNames method, 420
factory methods, 417
get methods, 418
getDisplayName method, 418
getDisplayNames method, 418, 420
getTime method, 420
set methods, 418
setTime method, 420
calendars
GregorianCalendar class, 417
internationalization, 415
JapaneseImperialCalendar class, 417
outputting calendar pages, 418
Thai Buddhist calendar, 417
call method, Callable interface, 385
calculating Euler's number, 389
Callable instance
Executors class returning, 387
Callable interface, 382
call method, 385
callable tasks, 382, 385
invokeAll method, 383
submit method, 384
submitting callable task to executor, 386
callbacks, 116
camel-casing, 44
cancel method, Future interface, 385
CancellationException, 385
candidate bundle names, 401, 402
canEncode method, Charset, 517
canExecute method, File, 456
canonical pathname
getCanonicalFile method, 452
getCanonicalPath method, 452
obtaining abstract pathname information, 453
canRead method, File, 454
canWrite method, File, 454
capacity
array, 283
ArrayList class, 329
ensureCapacity method, 283
hashtable, 358

capacity method, StringBuffer, 283
capturing groups, 440–441
  back reference, 441
Car class, inheritance, 98, 99
Card enum, Four of a Kind game, 25
Card object, Four of a Kind game, 26, 28
  leastDesirableCard method, 34
  rank method, 26
  suit method, 26
caret
  detecting logical boundaries within text, 409
carriage return
  escape sequence for, 49
  line terminator, 439
case reserved word
  equalsIgnoreCase method, 279
  switch statement, 65
  toLowerCase method, 280
  toUpperCase method, 280
case statement, 65
cast operator, 52
  conversions, 56
casting
  byte cast, 56
  downcasting, 121–123
  upcasting, 117–119
catch clause
  coding empty catch clauses, 171
  final rethrow, 172
  handling exceptions, 168, 169, 170
  multicatch, 171
  naming conventions, 169
  wrapping exceptions, 171
catch reserved word, 168
cause (wrapped exception), 164
ceil method, Math, 228
CEILING constant, RoundingMode enum, 238
chained instance method calls, 92
Character class, 142, 250–251
  Unicode-related complexity, 250
character classes, 439–440
character encodings, 511–512
  accessing default, 514
  Charset class finding, 516
  dumping to standard output, 516
character literal, 49
character sets, 511–512
Character type, 46
  getChar method, 275
  toCharArray method, 280
characters, 511
  getCharacterInstance method, 409, 410
  readChar method, 464
  setCharAt method, 283
  using character-based break iterator, 411
  writeChars method, 465
charAt method
  CharSequence interface, 436
  String class, 278, 281
  StringBuffer class, 283
CharSequence interface, 436
Charset class, 516–517
  java.nio.charset package, 514, 515
CharsetDecoder class, 515
CharsetEncoder class, 514
charsets, 517
  auto-detect charset, 517
charValue method, Character, 250
checked exceptions, 164, 168
checkError method, PrintStream, 511
child class *see* subclasses
child thread, 310
childValue method, InheritableThreadLocal, 310
class attributes, fields, 45
Class class, 270–274
  forName method, 270, 272, 273, 277
  getAnnotations method, 270, 274
  getDeclaredAnnotations method, 270, 274
  methods, 270–272
  obtaining Class object, 277
  processing annotations, 199
class extension *see* implementation inheritance
class field initialization, 48, 75, 82
class fields, 48, 94
  accessing, 88
  read-only class fields, 57
class header declaration
  strictfp used in, 234
class initialization, 77, 82
class invariants
class literals, 277
class methods, 60
  calling methods, 89
  main method, 60
class reserved word, 44
CLASS retention policy, 196, 197

ClassCastException, 108, 122, 317, 321
generics, 200, 201, 202, 205, 209, 211
classes, 43–85
see also objects
abstract classes, 120–121, 133
access control levels, 82
anonymous classes, 146–148
camel-casing of name, 44
character classes, 439–440
compatible class change, 500
composition, 112
constructors, 75–76
declaring, 44–45
declaring fields, 45–48
declaring methods, 58–60
declaring variables, 45
description, 94
extending, 98–102
fields, 45–58
final reserved word, 98, 101
generics, 200–214
generified classes, 200
getClass method, 103, 277
getDeclaringClass method, 274
getSuperclass method, 272
identifiers, 44
immutable classes, 100
implementation, 82
implementing interfaces, 127–129
implementing methods, 60–74
incompatible changes, 500
information hiding, 82
inheritance, 97–116
initialization order, 78–82, 102
initializing fields, 48–57
initializing instance fields, 75
inner classes, 139
interfaces, 82, 125–136
interfaces within, 150
intersection character classes, 440
local classes, 148–150
method overloading, 74
methods, 58–75
multiple inheritance, 102
naming convention for, 44
negation character classes, 439
nested classes, 139–150
nonstatic member classes, 142–145
packages, 151
predefined character classes, 440
primitive wrapper classes, 247–257
range character classes, 439
read-only fields, 57–58
static member classes, 139–142
stream classes, 473–475
subtraction character classes, 440
top-level classes, 139
types, 44
union character classes, 439
utility classes, 234
wrapper classes, 115, 116
classes, list of
see also exception classes
AbstractExecutorService, 386
AccessibleObject, 276
Array, 276
ArrayBlockingQueue, 392
ArrayList, 329–330
Arrays, 369–371, 434
BigDecimal, 234–239
BigInteger, 239–243
BitSet, 373–376
BitStreamInputStream, 492
BitStreamOutputStream, 492
Boolean, 248–250
BreakIterator, 409
BrokenBarrierException, 390
BufferedInputStream, 492, 493
BufferedOutputStream, 492
BufferedReader, 522
Byte, 255–257
ByteArrayInputStream, 477, 478
ByteArrayOutputStream, 477, 478–479
Character, 142, 250–251
Class, 270–274
ClassLoader, 402
Collator, 413
Collections, 371–372, 434
ConcurrentHashMap, 392
ConcurrentLinkedQueue, 392
Constructor, 274–275
CountDownLatch, 390
CyclicBarrier, 390
DataInputStream, 494
DataOutputStream, 494
DateFormat, 424
DateFormatSymbols, 425
DecimalFormat, 424
DecimalFormatSymbols, 424
Dictionary, 372
Double, 251–254
DumpArgs, 8

EchoText, 10
Enum, 220–222
EnumMap, 366
EnumSet, 337–339
Error, 164
EventQueue, 407
Exception, 164
Exchanger, 390
Executors, 387
Field, 275
File, 449–462
FileDescriptor, 466
FileInputStream, 479–481
FileOutputStream, 479
FileReader, 520–522
FileUtils, 522
FileWriter, 518–520
FilterInputStream, 488, 491
FilterOutputStream, 485
Float, 251–254
Format, 421
Formatter, 511
FutureTask, 386
HashMap, 145, 356–362
HashSet, 333–337
Hashtable, 372
IdentityHashMap, 362–364
Image, 406
ImageIcon, 408
InheritableThreadLocal, 310
InputStream, 473, 476–477
InputStreamReader, 515
Integer, 255–257
JOptionPane, 408
LineNumberInputStream, 474
LineNumberReader, 474
LinkedHashMap, 362
LinkedHashSet, 337
LinkedList, 330–332
ListResourceBundle, 402, 405
Long, 255–257
Matcher, 436
Math, 227–233
MessageFormat, 426
Method, 275–276
Number, 257
NumberFormat, 235, 421
Object, 103–112
ObjectInputStream, 497, 498
ObjectOutputStream, 497
OutputStream, 473, 475–476
OutputStreamWriter, 514
Package, 243–247
Pattern, 435
PhantomReference, 263–266
PipedInputStream, 481, 482
PipedOutputStream, 481, 482
Preferences, 429
PrintStream, 473, 474, 508–511
PrintWriter, 474
PriorityQueue, 348–350
Properties, 372, 403, 429
PropertyResourceBundle, 402, 403
Random, 432
RandomAccessFile, 462–473
Reader, 513
ReentrantLock, 394
Reference, 259
ReferenceQueue, 259, 260
RuleBasedCollator, 414
Semaphore, 390
Short, 255–257
SimpleDateFormat, 425
SoftReference, 260–263
Stack, 372
StoppableThread, 301
StrictMath, 233–234
String, 278–281
StringBuffer, 281–284
StringBufferInputStream, 474
StringReader, 474
StringTokenizer, 501
Subset, 142
System, 284–287
Thread, 288–289
ThreadGroup, 294
ThreadLocal, 308
Throwable, 163–164
Toolkit, 406
TreeMap, 355–356
TreeSet, 332–333
Vector, 372
WeakHashMap, 364–365
WeakReference, 263
Writer, 513
ZipOutputStream, 475
classfiles, 4
  javac identifying, 7
classloader, 4, 155
ClassLoader class, 402
ClassNotFoundException, 270, 273, 506

CLASSPATH environment variable, 154, 155, 159
classpath option, 154, 155, 159
cleanup, exceptions, 172–178
clear method
  BitSet class, 373
  Calendar class, 418
  Collection interface, 319
  Map interface, 351
  Reference class, 259
clearCache methods, ResourceBundle, 403
clone method
  BitSet class, 373
  Date class, 415
  Enum class, 220
  Object class, 103, 104–106
Cloneable interface, 104
CloneNotSupportedException, 104, 164
cloning
  deep copying/cloning, 105
  shallow copying/cloning, 104
close method
  FileInputStream class, 481, 488, 491
  FileOutputStream class, 488
  IntputStream class, 476
  OutputStream class, 475, 476
  PartsDB class, 470
  RandomAccessFile class, 464, 470
closed range/interval
  SortedSet interface, 343
closures, 148
code
  see also source code
  agile software development, 131
  converting pseudocode to Java code, 23–36
  native code, 4
  refactoring, 71
coercion polymorphism, 116
CollationKey class
Collator API, 413
Collator class, 413
collators, 413–414
  getCollationKey method, 414
  RuleBasedCollator class, 414
Collection interface, 318–321
  containsAll method, 319, 321
  hierarchy of interfaces, 316
  methods, 318–320
  remove method, 319, 321
  removeAll method, 320, 321
  retainAll method, 320, 321
  toArray methods, 320, 339, 367
collections, 315
  iterating over, 322
  storing primitive types in, 323
Collections class, 371–372, 434
  generic methods, 212
  methods, 371
  performance, 372
  shuffle methods, 434
collections framework, 315
  Collection interface, 318–321
  Comparable vs. Comparator, 316
  components, 315
  concurrent collections, 392–394
  core interfaces, 315, 316
    hierarchy, 316
  generic methods, 212
  generics, 200–202
  implementation classes, 315, 316
  Iterable interface, 318
  List interface, 325–328
  Map interface, 351–353
  Map.Entry interface, 354–355
  primitive wrapper classes, 248
  Queue interface, 346–347
  Set interface, 332
  sorted collection, 316, 317
  SortedMap interface, 367–369
  SortedSet interface, 339–346
  utility classes, 315, 316, 369–372
collisions
  hash functions, 357
command line
  executing JDK tools, 7
comments
  multiline comment, 59
  preventing bugs, 181
  single-line comment, 45
    Four of a Kind game, 29
Comparable interface, 317
  rules, 317
  sorted sets, 343, 344, 345
Comparator interface, 317
  methods, 317, 318
comparator method
  SortedMap interface, 367
  SortedSet interface, 340, 342
compare method, 318
  sorted sets, 345
compareTo method

Boolean class, 248
Comparable interface, 317
sorted sets, 344, 345
using comparator with priority queue, 350
Date class, 415
Enum class, 221, 222
File class, 462
sorted sets, 345
String class, 413
type parameter bounds, 207
comparison operators, 54
compatible class change, 500
compile method, Pattern, 435, 439
compiler, Java, 4
compile-time search, 154
complement operators
bitwise, 51
logical, 53
composition, 112
compound assignment operator, 52
compound expressions, 51–57
compound messages, 426, 428
formatting and outputting, 427
compound statement, 60, 95
compress method, Bitmap, 479
concat method, String, 278
concatenation
string concatenation operator, 51, 55
concrete parameterized type, generics, 203
concrete type, generics, 203
concurrency utilities, 381–397, 447
atomic variables, 397
concurrent collections, 392–394
executors, 381–389
locks, 394–396
synchronizers, 390–392
concurrent collections, 392–394, 447
BlockingQueue interface, 392
ConcurrentHashMap class, 392
ConcurrentLinkedQueue class, 392
ConcurrentMap interface, 392
ConcurrentModificationException, 327
conditional AND/OR operators, 52
connect method
FileWriter class, 519
PipedInputStream class, 482, 483
PipedOutputStream class, 482, 483
Console class, 156
Console tab, Eclipse, 20
constant interfaces, 160
constants
BigDecimal class, 236
BigInteger class, 240
Double class, 251
Float class, 251
Locale class, 398
primitive wrapper classes, 232, 248
read-only class fields, 57
RoundingMode enum, 238
Constructor class, 274–275
constructors, 75–76, 95
see also factory methods
ArrayList class, 329
BigDecimal class, 236, 239
BigInteger class, 240
Boolean class, 248
BufferedInputStream class, 493
BufferedOutputStream class, 493
Byte class, 255
ByteArrayInputStream class, 478
ByteArrayOutputStream class, 478
Character class, 250
Date class, 415
Double class, 251
enhancing an enum, 216
EnumMap class, 366
File class, 450, 451
FileInputStream class, 479
FileOutputStream class, 479
FilterOutputStream class, 485
Float class, 251
getConstructors method, 270
getDeclaredConstructors method, 270
HashMap class, 358
HashSet class, 334
InheritableThreadLocal class, 310
initializing fields, 48
InputStreamReader class, 515
Integer class, 255
LinkedList class, 331
Locale class, 398
Long class, 255
MessageFormat class, 426
OutputStreamWriter class, 514
PhantomReference class, 263
PipedInputStream class, 482
PipedOutputStream class, 482
PriorityQueue class, 348
Random class, 432
RandomAccessFile class, 462–464
Reference class, 260

ReferenceQueue class, 260
Short class, 255
SoftReference class, 260
SortedSet interface, 340
String class, 278, 280
StringBuffer class, 282
ThreadLocal class, 309
Throwable class, 163
TreeMap class, 355
TreeSet class, 332
WeakReference class, 263
contains method, Collection, 319
containsAll method, Collection, 319, 321
containsKey/containsValue methods, Map, 351
continue reserved word, 71
continue statement, 71
labeled continue statement, 72
Control class, 403
control-flow invariants, 184–185
conversions
byte cast, 56
cast operator, 56
operands of different types in operator, 56
Coordinated Universal Time *see* UTC
copy method, 176
copying
deep copying/cloning, 105
shallow copying/cloning, 104
copylist method, generics, 209, 210, 212, 213
core interfaces, collections framework, 315, 316
cos method, Math, 228
countdown latches, 390
countDown method, 390, 391
CountDownLatch class, 390
countTokens method, StringTokenizer, 502
covariance, arrays, 211
covariant return type, 123–124
createImage method, Toolkit, 406
createNewFile method, 459
createTempFile method, 459, 460
currency
formatting numbers as, 421
NumberFormat class, 235
current method, BreakIterator, 410
currentThread method, 288, 289, 290
currentTimeMillis method, 285, 286, 415
cursor position, ListIterator, 328
cyclic barriers, 390
CyclicBarrier class, 390

D

\d predefined character class, 440
daemon thread, 289
Dalvik virtual machine
Android platform, 5
data I/O, 449
data stream classes *see* stream classes
database driver
loading via class initializer, 77
databases
accessing, 530
creating flat file database, 467–472
DataInputStream class, 494
DataOutputStream class, 494
Date class, 415–416
methods, 415–416
date formatters, 424–425
DateFormat class, 424
setTimeZone method, 425
DateFormatSymbols class, 425
dates
internationalization, 415
Unix epoch, 425
daylight saving time, 417
deadlock, 306–308
deal method, Four of a Kind game, 28
decimal format, integer literals, 49
decimal numbers
formatting numbers as, 421
DecimalFormat class, 424
DecimalFormatSymbols class, 424
decimals
BigDecimal class, 234–239
Deck class, Four of a Kind game, 26, 28
methods, 28
decodeFile method, BitmapFactory, 479
Decorator design pattern, 115
deep copying/cloning, 105
default reserved word
switch statement, 65
defaultReadObject method, 505
defaultWriteObject method, 505
delete method, File, 459
deleteOnExit method, File, 459, 460
delimiters, 44
Javadoc comments, 34
Deprecated annotation type, 191

@deprecated javadoc tag, 190, 191
deprecation warnings, 192, 193
derived class *see* subclass
deserialization, 496–508
  custom, 501–505
  default, 496–501
  externalization, 506–508
  problematic deserialization, 501, 503
design patterns
  Abstract Factory, 155
  Decorator, 115
design-by-contract development, 185–188
  class invariants, 188
  postconditions, 187–188
  preconditions, 185–187
diamond operator (<>), generics, 202
Dictionary class, 372, 375
directories
  isDirectory method, 454
  methods for learning about, 454
  methods for obtaining content, 456
  mkdir method, 459
  mkdirs method, 459
  obtaining directory information, 455
disableassertions (da) option, 190
disblesystemassertions (dsa) option, 190
DiscardPile class, Four of a Kind game, 28, 29
  setTopCard method, 29
  topCard method, 29
disconnect method, FileWriter, 519
displayName method, Charset, 517
displayWeekdayNames method, 420
divide method
  BigDecimal class, 236, 389
  BigInteger class, 240
division operator, 52
  division by zero, 56
do reserved word, 69
documentation archive file, JDK, 11
Documented meta-annotation type, 198
Double class, 251–254
  methods, 252
Double precision floating-point type, 46
  getDouble method, 275
  nextDouble method, 433
double quote (")
  escape sequences, 49
doubleToIntBits method, 252
doubleToLongBits method, 252
doubleValue method, 252
do-while loop, 69
DOWN constant, RoundingMode enum, 238
downcasting, 121–123
Drawable interface, 125, 126
DumpArgs application, 7
  NetBeans IDE, 14
DumpArgs class, 8
dumpBitset method, 375
dumpPositions method, 412
dumpRecords method, 472
dumpSeparator method, 375

E

E constant, 227
early binding, 119
EchoText class, 10
Eclipse Classic, 17
Eclipse IDE, 17–20
  Console tab, 20
  Package Explorer tab, 19
  workbench, 19
editor area, NetBeans IDE, 15
element method, Queue, 347
elements
  adding to annotation type, 194
  annotating undesirable application elements, 197
  NoSuchElementException, 321
elements method, Vector, 372
ElementType enum, 196
else reserved word, 62
  chaining if-else statements, 64
  if-else statement, 62
empty method, Stack, 372
empty statement, 61
  loop statement excuting, 69
enableassertions (ea) option, 189
enablesystemassertions (esa) option, 190
encapsulation
  information hiding, 82
  objects, 43
  problem with implementation inheritance, 112
enclosing class
  interfaces within classes, 150
  local classes, 149
  nonstatic member classes, 142
  static member classes, 139
encodings, character, 511–512
  default character encodings, 514

end method, Matcher, 436
endianness
  big-endian format, 494
endsWith method, String, 278
enhanced for loop statement, 322
enqueue method, Reference, 259
ensureCapacity method, StringBuffer, 283
Enterprise Edition, Java, 5
entities, 43
entity attributes, fields, 45
Entry interface, maps, 354–355
entrySet method
  Map interface, 351, 353, 354
  SortedMap interface, 367
Enum class, 220–222
  extending, 221–222
  methods, 220–222
enum reserved word, 215, 337
enumerate method, ThreadGroup, 294
enumerated type, 214, 226
  example of, 65, 337
  problem with, 214–215
Enumeration interface, 372, 375
EnumMap class, 366
enums, 214–222, 226
  anonymous subclasses, 219
  ElementType enum, 196
  enhancing, 216–220
  RetentionPolicy enum, 196
  RoundingMode enum, 237
  State enum, 288
  TimeUnit enum, 385
  tokens, 218
  toString method, 216, 218
  type safety, 216
  values method, 218, 219, 221
EnumSet class, 337–339
  methods, 339
equality operator, 52, 106
equals method
  accessing another instances private members, 336
  Boolean class, 248
  Collection interface, 319
  Comparator interface, 318
  Date class, 415
  Double class, 252, 253
  Enum class, 220
  File class, 462
  Float class, 252, 253
  HashSet class, 335, 336
  Map interface, 351
  Map.Entry interface, 354
  Object class, 103, 107–109, 110
  Set interface, 332
  SortedSet interface, 345, 346
  String class, 278
equalsIgnoreCase method, String, 279
erasure, generics, 211–212
Error class, 164
error codes
  ignoring, 163
  representing exceptions, 162–163
errors
  AssertionError, 182, 184
  exception and error class hierarchy, 163–164
  ExceptionInInitializerError, 273
  exceptions, 162
  handling, 163
  LinkageError, 270, 273
  System.err, 10
  throwing assertion error
    with detail message, 183
    without detail message, 182
escape sequences, 49
Euler's number, calculating, 387
event-dispatching thread, 407
EventQueue class
  invokeLater method, 407, 408
Exception class, 164, 179
exception classes
  ArithmeticException, 165, 389
  ArrayStoreException, 211, 212, 285, 320
  BrokenBarrierException, 390
  CancellationException, 385
  ClassCastException, 108, 122, 317, 321
  ClassNotFoundException, 270, 273, 506
  CloneNotSupportedException, 104, 164
  ConcurrentModificationException, 327
  Exception class, 164, 179
  ExecutionException, 383, 384, 385
  FileNotFoundException, 170, 171, 174, 176, 463, 479, 519
  IllegalAccessException, 272, 275, 276
  IllegalArgumentException, 167
  IllegalMonitorStateException, 303
  IllegalStateException, 321, 327, 328
  IllegalThreadStateException, 289
  IndexOutOfBoundsException, 278, 280, 283, 285, 325, 326
  InstantiationException, 272

InterruptedException, 260, 289, 383, 384, 385
InvalidClassException, 500, 503, 507
InvocationTargetException, 276
IOException, 11, 164, 170, 171, 174, 176, 453, 470
MissingResourceException, 402, 403, 405
NoSuchElementException, 321, 327, 340, 347, 367, 368, 371
NotSerializableException, 496, 497
NullPointerException, 167, 275, 276, 317
NumberFormatException, 254
OptionalDataException, 498
PatternSyntaxException, 435, 436, 437
RejectedExecutionException, 382
ReportCreationException, 171
RuntimeException, 165, 179
TimeoutException, 384, 385
UnsupportedOperationException, 320
exception handling
run method throwing, 294
setDefaultUncaughtExceptionHandler method, 295
setUncaughtExceptionHandler method, 295
ExceptionInInitializerError, 273
exceptions, 161–178, 179
checked exceptions, 164
cleanup, 172–178
custom classes, 165–166
errors, 162
exception and error class hierarchy, 163–164
final rethrow, 172
getExceptionTypes method, 275
handling, 163, 168–172
multicatch, 171
naming conventions, 165
representing in source code, 162–166
representing via error codes or objects, 162–163
rethrowing, 172
runtime exceptions, 165
throwing, 163, 166–168
throws Exception clause, 199
unchecked exceptions, 165
when not to use assertions, 189
wrapped exception (cause), 164
wrapping, 171
Exchanger class, 390
exchangers, 390
exclusion
mutual exclusion, 299
threads, 302
execute method, Executor, 382
executing code, threads, 287
execution environment, 3
ExecutionException, 383, 384, 385
Executor interface, 382
executors, 381–389, 447
RejectedExecutionException, 382
ScheduledExecutorService interface, 386
shutdownNow method, 389
submitting callable task to, 386
Executors class, 387
ExecutorService interface, 383
methods, 383–384
exists method, File, 454
exp method, Math, 228
expressions
compound expressions, 51–57
escape sequences, 49
simple expressions, 48–50
ext directory
storing packages, 154
extends reserved word, 98, 130
Enum class, 221
type parameter bounds, 205
Externalizable interface, 506
readExternal method, 506, 508
writeExternal method, 506, 507
externalization, 506–508
initiating, 507

F

factorials
BigInteger class, 241–243
factory methods
see also constructors
BreakIterator class, 409
Calendar class, 417
Collator class, 413
DateFormat class, 424
NumberFormat class, 421
ResourceBundle class, 401
TimeZone class, 417, 425
fail-fast behavior
ArrayList/LinkedList classes, 327
false reserved word

Boolean literal, 49
Field class, 275
fields, 45–58, 94
accessing, 87–89
class fields, 48
declaring in classes, 45–48
enhancing an enum, 216, 217
flat file database, 467
getDeclaredFields method, 271
getFields method, 271
initialization order, 82
initializing array-based fields, 50
initializing in classes, 48–57
initializing instance fields, 75
naming convention for, 45
read-only class field, 57
read-only fields, 57–58
read-only instance fields, 57
types, 45
variable declared within class, 467
File class
canExecute method, 456
canRead method, 454
canWrite method, 454
compareTo method, 462
comparing two File objects, 462
constructors, 450, 451
createNewFile method, 459
createTempFile method, 459, 460
delete method, 459
deleteOnExit method, 459, 460
equals method, 462
exists method, 454
getAbsoluteFile method, 452
getAbsolutePath method, 452
getCanonicalFile method, 452, 462
getCanonicalPath method, 452
getFreeSpace method, 456
getName method, 452, 454
getParent method, 452
getParentFile method, 452
getPath method, 452, 454
getTotalSpace method, 456
getUsableSpace method, 456
hashCode method, 462
I/O, 449–462
isAbsolute method, 453
isDirectory method, 454
isFile method, 454
isHidden method, 454
lastModified method, 455
length method, 455
list method, 456, 457
listFiles method, 456, 457, 458
listRoots method, 449
logging library using packages, 157
methods for
creating and manipulating files, 459
learning about file or directory, 454
learning about stored pathname, 452
obtaining directory content, 456
mkdir method, 459
mkdirs method, 459
renameTo method, 460
setExecutable method, 461
setLastModified method, 460
setReadable method, 461
setReadOnly method, 460
setWritable method, 461
toString method, 453
file descriptors
getFD method, 464
file pointers
getFilePointer method, 464
RandomAccessFile class, 464
FileDescriptor class, 466
FileFilter interface, 458
FileInputStream class, 174, 479–481
close method, 481, 488, 491
read method, 488, 489, 490, 491, 492
FilenameFilter interface, 458
accept method, 457
FileNotFoundException, 170, 171, 174, 176, 463, 479, 519
FileOutputStream class, 174, 479
close method, 488
FileReader class, 520–522
files
encoding property, 514
isFile method, 454
metadata, 463
methods for creating and manipulating, 459
methods for learning about, 454
obtaining file information, 455
Files tab, NetBeans IDE, 15
filesystem roots, 449
FileUtils class, 522
FileWriter class, 518–520
methods, 519
fill method, Arrays, 370
FilterInputStream class, 488, 491

FilterOutputStream class, 485
  chaining streams together, 488
  subclassing, 485, 486
  write method, 485, 486, 492
final reserved word
  constructors, 75
  extending final class, 98
  overriding final method, 101
  read-only class field, 57
  read-only fields, 57
final rethrow, catch clause, 172
finalization
  runFinalization method, 285
finalize method
  Enum class, 220
  Object class, 103, 109–110, 257
  resurrection, 264
finally clause, 172–176, 179
finally reserved word, 172
  finalize method, 110
find method, Matcher, 437, 438, 522
FindAll application, 520
findAll method, 521, 522
first method
  BreakIterator class, 410
  SortedSet interface, 340
first-class function, 148
first-in, first-out (FIFO) queue, 346
firstKey method, SortedMap, 367
flags method, Pattern, 435
flat file database, 467–472
Float class, 251–254
  methods, 252
floating-point literal, 50
Floating-point type, 46
  getFloat method, 275
  nextFloat method, 433
floating-point values
  BigDecimal class, 234
floatToIntBits method, 252
floatValue method, 252
FLOOR constant, RoundingMode enum, 238
floor method, Math, 228
flush method
  OutputStream class, 475, 476
  PrintStream class, 511
following method, BreakIterator, 410
for loops, 66–67
  breaking out of nested for loops, 72
  continuing, 72
  enhanced statement, 322
for reserved word, 66
form feed, escape sequence for, 49
Format class, 421
format method
  Format class, 421
  MessageFormat class, 427
  NumberFormat class, 235
format specifiers, 509
Formatter class, 511
formatters, 421–428
  date formatters, 424–425
  message formatters, 426–428
  number formatters, 421–424
forName method, 270, 272, 273, 277
  processing annotations, 199
forwarding methods, 114
Four of a Kind application, 20–40
  compiling, 37
  distributing, 39
  problem domain, 21
  running, 37
Four of a Kind game, 20–40
  Card enum, 25
  Card object, 26, 28
  converting pseudocode to Java code, 23–36
  deal method, 28
  Deck class, 26, 28
  DiscardPile class, 28, 29
  isEmpty method, 28
  Javadoc comments, 25
    extracting comments from code, 35
  leastDesirableCard method, 34
  modeling in pseudocode, 21–23
  object of the game, 21
  ordinal method, 34
  playing Four of a Kind, 21
  putBack method, 28
  Rank enum, 26, 34
  rank method, 26
  rankCounts array, 34
  setTopCard method, 29
  shuffle method, 28
  Suit enum, 26
  suit method, 26
  topCard method, 29
FourOfAKind class, 30
framework, collections, 315
free variable, 148
FULL constant, DateFormat, 424
functions

first-class function, 148
Future interface, 385
FutureTask class, 386

G

\G boundary matcher, 441
games
Four of a Kind, 20–40
garbage collection, 92–93
finalize method, 109
gc method, 285
PhantomReference objects, 258, 264
reachable objects, 258
References API, 257–266
root set of references, 258
SoftReference objects, 258, 261, 262
WeakHashMap class, 365
WeakReference objects, 258, 263
Gaussian value
nextGaussian method, 433
gc method, System, 285
generic methods, 212–214
generic types, 202–212
declaring, 202
declaring and using, 203–205
raw type, 203
Reference class, 259
ReferenceQueue class, 260
type parameter bounds, 205–207
type parameter scope, 208
generics, 200–214, 226
actual type arguments, 203
array type, 203
collections and type safety, 200–202
concrete parameterized type, 203
concrete type, 203
erasure, 211–212
parameterized types, 201, 202
recursive type bound, 208
reification, 211
type parameter, 203
type parameter bounds, 205–207
type parameter scope, 208
wildcards, 208–210
generified classes, 200
get method
BitSet class, 373
Calendar class, 418
Field class, 275
Future interface, 385
List interface, 325
Map interface, 351
PhantomReference class, 263
Preferences class, 430
Reference class, 259
ThreadLocal class, 309
getAbsoluteFile method, 452
getAbsolutePath method, File, 452
getAndIncrement method, AtomicLong, 397
getAnnotation method
AccessibleObject class, 276
processing annotations, 199
getAnnotations method, 270, 274
getAvailableLocales method
Collator class, 413
Locale class, 410
getBoolean method
Field class, 275
Boolean class, 248, 249
getBundle methods, ResourceBundle, 401, 402, 404
getByte method, Field, 275
getCanonicalFile method, 452
comparing two File objects, 462
getCanonicalPath method, File, 452
getCause method, Throwable, 163
getChar method, Field, 275
getClass method, Object, 103, 277
getCollationKey method, 414
getConstructors method, 270
getDateInstance methods, 424
getDateTimeInstance methods, 424
getDeclaredAnnotations method, 270, 274
getDeclaredConstructors method, 270
getDeclaredFields method, 271
getDeclaredMethods method, 271
getDeclaringClass method
Constructor class, 274
Enum class, 221, 222
getDefault method
Locale class, 399
TimeZone class, 417
getDescription method, PatternSyntaxException, 436
getDisplayName method, Calendar, 418
getDisplayNames method, Calendar, 418, 420
getDouble method, Field, 275
getEncoding method
InputStreamReader class, 515
OutputStreamWriter class, 515

getExceptionTypes method, Constructor, 275
getFD method, RandomAccessFile, 464, 465, 466
getFields method, 271
getFilePointer method, RandomAccessFile, 464
getFloat method, Field, 275
getFreeSpace method, File, 456
getID3Info method, Media, 526
getImplementationXyz methods, Package, 243
getIndex method, PatternSyntaxException, 436
getInstance factory methods, NumberFormat, 421
getInt method, Field, 275
getISOCountries method, Locale, 399
getISOLanguages method, Locale, 399
getKey method, Map, 354
getLong method, Field, 275
getMessage method
  PatternSyntaxException class, 436
  Throwable class, 163
getMethods method, 271
  processing annotations, 199
getModifiers method, Method class, 276
getName method
  Class class, 271
  Constructor class, 275
  File class, 452, 454
  Package class, 244
  Thread class, 288, 289, 290
getNextID method, 397
getNumberInstance methods, 421
getObject method, 403
getPackage method
  Class class, 271
  Package class, 244, 245
getPackages method, 244
getParameterTypes method, Constructor, 275
getParent method, File, 452
getParentFile method, 452
getPath method, File, 452, 454
getPattern method, 436
getProperty method
  Properties class, 372
  System class, 285, 286, 508
getResource method, ClassLoader, 402
getReturnType method, 276
getSharedChar method, Object, 306
getShort method, Field, 275
getSpecificationXyz methods, Package, 244
getStackTrace method, Throwable, 163
getState method, Thread, 288
getString method, 403
getStringArray method, 403
getSuperclass method, 272
getters, 83
  accessing fields, 88
getThreadGroup method, 294
getTime method
  Calendar class, 420
  Date class, 416
getTimeInstance methods, DateFormat, 424
getTimeZone method, 417, 425
getTotalSpace method, File, 456
getUsableSpace method, File, 456
getValue method, Map, 354
GMT (Greenwich Mean Time), 415
Google
  Android platform, 5
goto statement, 72
greater than operators, 54
greedy quantifier, 442, 443
GregorianCalendar class, 417
grep utility, Unix, 520
group method, Matcher, 436

H

HALF_XYZ constants, RoundingMode enum, 238, 239
handle, Java, 466
handles, 85
handling errors, 163
handling exceptions, 163, 168–172
"has a" relationship, 112
hash code, 357
  identityHashCode method, 362
hash function, 356
hash maps
  ConcurrentHashMap class, 392
hashCode method
  Boolean class, 249
  Collection interface, 319
  creating hashmap of String keys and Integer values, 359
  Date class, 416
  Double class, 252
  Enum class, 220
  File class, 462

Float class, 252
HashSet class, 335, 336
Map interface, 352
Map.Entry interface, 354
Object class, 103, 110–111
overriding to return proper hash codes for Point objects, 361
overriding when override equals(), 346
problem of not overriding, 111
Set interface, 332
HashCodeBuilder class, 360
hashing, 110
HashMap class, 145, 356–362
constructors, 358
contrasting IdentityHashMap with, 362
counting command-line arguments, 358
creating hashmap of String keys and Integer values, 359
problem of not overriding hashCode, 111
hashmaps
IdentityHashMap class, 362–364
LinkedHashMap class, 362
preventing memory leaks, 263
WeakHashMap class, 364–365
HashSet class, 333–337
equals method, 335, 336
hashCode method, 335, 336
LinkedHashSet class, 337
Hashtable class, 372, 375
hashtables
buckets, 357, 358
load factor, 358
hasMoreElements method, Enumeration, 372
hasNext method
Iterator class, 321, 322
ListIterator class, 327
hasPrevious method, ListIterator, 327
headMap method, SortedMap, 367
headSet method, SortedSet, 340, 341, 343
heap
new operator, 85, 86
helper methods, 82
hexadecimal format
integer literals, 49
toHexString method, 255
holdsLock method, Thread, 299
home directory, JDK, 7
horizontal tab
escape sequence for, 49
Huffman coding, 376

I/O *see* input/output
ID class, 397
ID3, 526
IDEs (integrated development environments), 12
Eclipse IDE, 17–20
NetBeans IDE, 13–17
identifiers, 44
identity check, 106
identityHashCode method, 362
IdentityHashMap class, 362–364
IDs
different user IDs for different threads, 309
returning unique IDs, 299
if reserved word, 62
if-else statement, 62–64
chaining if-else statements, 64
IllegalAccessException, 272, 275, 276
IllegalArgumentException, 167
add methods, collections, 321
assertions, 189
get methods, fields, 275
invoke method, Method class, 276
IllegalMonitorStateException, 303
IllegalStateException, 321, 327, 328
add method, queues, 346
entry methods, maps, 354, 355
IllegalThreadStateException, 289
image cache
SoftReference objects, 261–262
Image class, 406
ImageIcon class, 408
immutability, string objects, 278
immutable classes, 100
implementation
access control levels, 82
classes, 82
decoupling interface from, 131
handles, 85
implementation classes
collections framework, 315, 316
implementation inheritance, 97
extending classes, 98–102
"is a" relationship, 98
multiple class inheritance, 102
problem with, 112
implements reserved word, 127
implicit type conversion
coercion polymorphism, 116

import reserved word, 153
import statement, 153–154, 179
  static import statement, 161
imports, static, 160–161
incompatible class change, 500
index
  array index operator, 51
  lastIndexOf method, 279, 326
  nextIndex method, 327
  previousIndex method, 327
indexOf method
  List interface, 325
  String class, 279
IndexOutOfBoundsException, 278, 280, 283, 285, 325, 326
inequality operator, 52, 106
infinite loops, 70
  breaking out of, 70
infinity, 231
  division by zero, 56
  equals method, 253
  isInfinite method, 252
  NEGATIVE_INFINITY constant, 251
  POSITIVE_INFINITY constant, 251
infix operator, 51
information hiding, 82, 281
InheritableThreadLocal class, 310
inheritance, 97–116
  composition, 112
  extending classes, 98–102
  implementation inheritance, 97
  interface inheritance, 128
  "is a" relationship, 98
  multiple class inheritance, 102
  Object class, 103–112
  overriding methods, 100
  single inheritance, 102
Inherited meta-annotation type, 198
initCause method, Throwable, 164
initialization
  class field initialization, 48, 75
  class initialization, 77, 270
  instance field initialization, 47, 48, 75
  instance initialization, 77–78
  lazy initialization, 432
initialization order, 78–82, 102
initializing fields, 48–57
initialValue method, ThreadLocal, 309, 310
inner classes, 139
input/output
  BufferedReader class, 522
  File class, 449–462
  FileReader class, 520–522
  FileWriter class, 518–520
  I/O Redux, 530
  input stream, 473
  InputStream class, 473, 476–477
  InputStreamReader class, 515
  output stream, 473
  Output tab, NetBeans IDE, 16
  OutputStream class, 473, 475–476
  OutputStreamWriter class, 514, 515
  RandomAccessFile class, 462–473
  Reader class, 513
  reader classes, 512–513
  readers and writers, 511–529
  streams, 473–511
  System.in, 10
  System.out, 9
  Writer class, 513
  writer classes, 512–513
instance fields, 94
  accessing fields, 88
  examples of, 47
  initialization, 47, 48, 75, 82
  read-only instance fields, 57
instance initialization, 77–78, 82
instance methods, 59
  calling methods, 89
  chained instance method calls, 92
instanceof operator, 108
  runtime type identification, 122
instanceof reserved word, 54
instances
  initializing instance fields, 75
  newInstance method, 272
instantiation
  abstract reserved word, 121
  objects, 43
InstantiationException, 272
int reserved word, 45
Integer class, 255–257
  autoboxing and unboxing, 323, 324
  intValue method, 323
  valueOf method, 324
integer classes
  Byte class, 255–257
  Integer class, 255–257
  Long class, 255–257
  methods, 255
  Short class, 255–257
integer literals, 49

underscores, 50
Integer type, 46
integers
BigInteger class, 239–243
formatting numbers as, 421
getInt method, 275
nextInt method, 433
readInt method, 465
shuffling an array of, 434
writeInt method, 465
integrated development environments *see* IDEs
interface inheritance, 128, 130
interface reserved word, 125
declaring annotation type, 193
preceding with abstract, 126
preceding with public, 126
@interface syntax, annotations, 225
interfaces, 125–136
able suffix, 126
class design, 82
constant interfaces, 160
declaring, 125–126
decoupling from implementation, 131
description, 82
extending, 130–131
implementing, 127–129
interfaces within classes, 150
marker interfaces, 126, 194
methods, 126
naming conventions, 125
reasons for using, 131–136
tagging interface, 126
interfaces, list of
Annotation, 274
Appendable, 513
BlockingQueue, 392, 393, 396
Callable, 382, 385
CharSequence, 436
Cloneable, 104
Collection, 318–321
Comparable, 317
Comparator, 317
ConcurrentMap, 392
Drawable, 125, 126
Enumeration, 372, 375
Executor, 382
ExecutorService, 383
Externalizable, 506
FileFilter, 458
FilenameFilter, 457, 458
Future, 385
Iterable, 318
List, 325–328
Lock, 394
Logger, 155
Map, 351–353
Map.Entry, 354–355
Queue, 346–347
Runnable, 288
ScheduledExecutorService, 386
Serializable, 496
Set, 332
SortedMap, 367–369
SortedSet, 339–346
intern method, String, 279, 280
internal invariants, 183–184
internationalization, 398
break iterators, 409–413
calendars, 415–420
collators, 413–414
Date class, 415
dates, 415–420
formatters, 421–428
International Standards Organization (ISO), 398
locales, 398–400
resource bundles, 400–409
time zones, 415–420
Unicode, 398
Internationalization APIs, 397–428
Locale class, 398
ResourceBundle pattern, 401
interpreter, virtual machine, 4
interrupt method, Thread, 288
interrupted method, Thread, 289
InterruptedException, 260, 289, 383, 384, 385
interruption
isInterrupted method, 289
intersection character class, 440
IntputStream class, 476–477
intValue method, Integer, 255
autoboxing and unboxing, 323
InvalidClassException, 500, 503, 507
invariants, 183
class invariants, 188
control-flow invariants, 184–185
internal invariants, 183–184
InvocationTargetException, 276
invoke method, Method class, 276
invokeAll method, ExecutorService, 383

invokeAny method, ExecutorService, 383, 384
invokeLater method, EventQueue, 407, 408
IOException, 11, 164, 170, 171, 174, 176, 453, 470
"is a" relationship, 98
isAbsolute method, File, 453
isAccessible method, AccessibleObject, 276
isAlive method, Thread, 289, 293
isAnnotationPresent method
  AccessibleObject class, 276
  processing annotations, 199
isCancelled method, Future interface, 385
isCompatibleWith method, Package, 244, 246
isDaemon method, Thread, 289
isDigit method, Character, 250, 251
isDirectory method, File, 454
isDone method, Future interface, 38, 389
isEmpty method
  Collection interface, 319
  Four of a Kind game, 28
  Map interface, 352
isEnqueued method, Reference, 260
isFile method, 454
isHidden method, File, 454
isInfinite method, 252, 253
isInterrupted method, Thread, 289
isLetter method, Character, 250
isLetterOrDigit method, Character, 250
isLowerCase method, Character, 251
isNaN method, 252
ISO (International Standards Organization), 398
  getISOCountries method, 399
  getISOLanguages method, 399
ISO 639, 399
ISO/IEC 8859
  character sets and encodings, 512
isSealed method, Package, 244
isShutdown method, ExecutorService, 384
isStopped method, StoppableThread, 302
isTerminated method, ExecutorService, 384
isUpperCase method, Character, 251
isVarArgs method, Method class, 276
isWhitespace method, Character, 251
Iterable interface, 318
  hierarchy of interfaces, 316
  iterator method, 318, 321
iterating over collections, 322
iterations *see* loops
Iterator class, 372
  enhanced for loop statement, 322
  methods, 321, 322
iterator method
  Collection interface, 319
  Iterable interface, 318, 321
  SortedMap interface, 367
  SortedSet interface, 339
iterators
  listIterator methods, 326

J

JapaneseImperialCalendar class, 417
jar (Java archive creator, updater, and extractor), 7, 475
JAR (Java ARchive) files
  distributing Four of a Kind, 39
  MANIFEST.MF file, 247
  modules replacing, 159
  packages and, 159–160
  packages information, 247
  sealed packages, 243
  storing packages, 154
Java, 1–6, 41, 43
  annotations, 190–200
  assertions, 181–190
  compared with C/C++, 2
  documentation archive file, 11
  enums, 214–222
  exception and error class hierarchy, 163–164
  exceptions, 161–178
  generics, 200–214
  handle, 466
  history, 2
  inheritance, 97–116
  interfaces, 125–136
  language features, 2
  loops, 3
  nested types, 139–150
  Object class, 103–112
  packages, 151–160
  platforms, 3
  polymorphism, 116–124
  portability, 4, 46
  primitive types, 46
  security, 5
  static imports, 160–161
  stream destinations, 473
  stream sources, 473

String class, 86
java (Java application launcher), 7
running Four of a Kind, 37
Java APIs *see* APIs
Java code
converting pseudocode to, 23–36
Java compiler, 4
Java EE (Enterprise Edition), 5
Java ME (Micro Edition), 5
Java packages, 151–160
see also packages
java.awt, 406
java.io, 151
File class, 449–462
I/O, 449
input stream classes, 473
InputStream class, 473
LineNumberInputStream class, 474
LineNumberReader class, 474
output stream classes, 473
OutputStream class, 473
pathnames, 451
PrintStream class, 473, 474
PrintWriter class, 474
RandomAccessFile class, 462–473
StringBufferInputStream class, 474
StringReader class, 474
java.lang, 151, 154
CharSequence interface, 436
Class class, 270–274
ClassLoader class, 402
Comparable interface, 317
Math class, 227–233, 268, 432
Number class, 257
Package class, 243–247, 268
primitive wrapper classes, 247–257, 268
Runnable interface, 382
StrictMath class, 233–234, 268
String class, 278–281
StringBuffer class, 281–284
System class, 284–287
java.lang.ref, 268
Reference class, 259
ReferenceQueue class, 259, 260
References API, 259
java.lang.reflect, 151
AccessibleObject class, 276
Array class, 276
Constructor class, 274–275
Field class, 275
Method class, 275–276
Modifier class, 276
java.math
BigDecimal class, 234–239, 268
BigInteger class, 239–243, 268
java.net, 151
java.nio.charset
Charset class, 514, 515
CharsetDecoder class, 515
CharsetEncoder class, 514
java.text
BreakIterator class, 409
Collator class, 413
DateFormat class, 424
DateFormatSymbols class, 425
DecimalFormat class, 424
DecimalFormatSymbols class, 424
Format class, 421
NumberFormat class, 235, 421
RuleBasedCollator class, 414
SimpleDateFormat class, 425
java.util, 151
Arrays class, 434
Calendar class, 417–420
Collections class, 434
Date class, 415–416
GregorianCalendar class, 417
ListResourceBundle class, 402
Locale class, 398
MessageFormat class, 426
Properties class, 403
PropertyResourceBundle class, 402
Random class, 432
ResourceBundle class, 401
SimpleTimeZone class, 417
TimeZone class, 417
java.util.concurrent, 287, 381, 392
AbstractExecutorService class, 386
BrokenBarrierException class, 390
Callable interface, 382
CountDownLatch class, 390
Exchanger class, 390
Executor interface, 382
Executors class, 387
ExecutorService interface, 383
Future interface, 385
FutureTask class, 386
RejectedExecutionException, 382
ScheduledExecutorService interface, 386
Semaphore class, 390

java.util.concurrent.atomic, 381, 397
java.util.concurrent.locks, 381, 394
java.util.prefs, 429
java.util.regex
Matcher class, 436
Pattern class, 435
java.util.zip, 474
javax.swing, 408
Java Runtime Environment (JRE), 6
Java SE (Standard Edition), 5
Java SE Development Kit see JDK
javac (Java compiler), 7
assertions, 181
compiling Four of a Kind, 37
javadoc (Java documentation generator), 7, 35
Javadoc comments, 34–36
delimiters, 34
extracting comments from code, 35
Four of a Kind game, 25
Javadoc tags
@author, 35
@deprecated, 190, 191
@param, 35
@return, 35
javax.swing package, 408
JDBCFilterDriver class, 77
JDK (Java SE Development Kit), 6, 7
documentation archive file, 11
executing JDK tools, 7
JIT compilation, 4
Joda Time, 420
join method, Thread, 289, 293, 294
JOptionPane class
showMessageDialog method, 408
JRE (Java Runtime Environment), 6
jre directory, JDK, 7
Just In Time (JIT) compiler, 4

K

key/value pairs, maps, 351
keys
containsKey method, 351
getCollationKey method, 414
mutable keys, 362
keySet method
Map interface, 352, 353
SortedMap interface, 368, 517
keywords see reserved words

L

labeled break statement, 72
labeled continue statement, 72
language features, Java, 2
language translations
Yahoo! Babel Fish, 408
last method
BreakIterator class, 410
SortedSet interface, 340
last-in, first-out (LIFO)
queue, 346
stack, 225
lastIndexOf method
List interface, 326
String class, 279
lastKey method, SortedMap, 368
lastModified method, File, 455
late binding
subtype polymorphism, 117–119
lazy initialization, 432
leastDesirableCard method
Four of a Kind game, 34
left shift operator, 53
length method
BitSet class, 373
CharSequence interface, 436
File class, 455
RandomAccessFile class, 464
String class, 279, 281
StringBuffer class, 283
length, array, 283
less than operators, 54
lib directory, JDK, 7
lifetime, 48
isAlive method, 289, 293
local variables, 61
line
getLineInstance method, 409
separator property, 508
line feed
escape sequence for, 49
line separators, 439
line terminators, 438, 439, 508
linear congruential generator, 432
linear search, 370
LineNumberInputStream class, 474
LineNumberReader class, 474
LinkageError
forName method, 270, 273
LinkedHashMap class, 362
LinkedHashSet class, 337

LinkedList class, 330–332
fail-fast behavior, 327
nodes, 330
linking
forName method, 270
List interface, 325–328
ArrayList class, 329–330
hierarchy of interfaces, 316
LinkedList class, 330–332
methods, 325–326
range-view operations, 328
subList method, 326, 328
views, 328
list method, File, 456, 457
list resource bundle, 405–409
listFiles method, 456, 457, 458
ListIterator class, 326
methods, 326–328
navigating through lists, 328
ListResourceBundle class, 402, 405
listRoots method, File, 449
lists, 325
ArrayList class, 329–330
LinkedList class, 330–332
literals, 48
Boolean literal, 49
character literal, 49
floating-point literal, 50
integer literal, 49
string literal, 48
liveness, 397
load factor, hashtables, 358
loading
forName method, 270
local classes, 148–150
local variables, 61
accessing fields, 88
naming conventions, 61
Locale class, 398
methods, 399
locales, 398–400
getAvailableLocales method, 410
resource bundles, 400
Locale-Sensitive Services, 398
localization, 400
Lock interface, 394
lock method, 396
lockInterruptibly method, 394
tryLock method, 394
unlock method, 396
wait/notification mechanism, 394
lock method, 396
lockInterruptibly method, 394
locks, 299, 394–396
achieving synchronization, 394
deadlock, 306–308
holdsLock method, 299
ReentrantLock class, 394
log method, FileWriter, 519
log method, Math, 229
log10 method, Math, 229
Logger interface
using packages, 155
LoggerFactory abstract class, 156
logging library
using packages, 155–159
logical operators, 53
Long class, 255–257
LONG constant, DateFormat, 424
Long integer type, 46
getLong method, 275
literal representing, 49
nextLong method, 434
longValue method, 255
lookingAt method, Matcher, 437
loops, 66–69
breaking out of nested for loops, 72
continuing for loop, 72
control variables, 66
do-while loop, 69
excuting empty statement, 69
for loop, 66–67
infinite loops, 70
Java, 3
while loop, 67–68
lowercase
isLowerCase method, 251
toLowerCase method, 251

M

main method, 60
DumpArgs class, 8
FourOfAKind class, 30
initialization order, 82
Main-Class specifier
distributing Four of a Kind as JAR file, 39
MANIFEST.MF file
distributing Four of a Kind as JAR file, 39
packages information, 247
Map interface, 351–353
ConcurrentMap subinterface, 392

entrySet method, 351, 353, 354
hierarchy of interfaces, 316
keySet method, 352, 353
methods, 351–353
Map.Entry interface, 354–355
maps, 351
EnumMap class, 366
HashMap class, 356–362
IdentityHashMap class, 362–364
LinkedHashMap class, 362
SortedMap interface, 367–369
TreeMap class, 355–356
WeakHashMap class, 364–365
mark method, IntputStream, 476, 477
marker annotations, 194
marker interfaces, 126, 194
markSupported method, IntputStream, 476, 477
masking
type parameter, generics, 208
Matcher class, 436
find method, 437, 438, 522
methods, 436–437
reset method, 437, 522
matcher method, Pattern, 435, 436
matches method
Matcher class, 436
Pattern class, 435
math
StrictMath class, 233–234
Math APIs, 227–243
Math class, 227–233, 268
final reserved word, 234
methods, 227
random method, 432
StrictMath class compared, 233
utility classes, 234
max method
BigDecimal class, 236
BigInteger class, 240
Math class, 229, 232
MAX_VALUE constant, 232, 251, 255
Media class
getID3Info method, 526
MEDIUM constant, DateFormat, 424
member access operator, 53
accessing fields, 88
calling overridden methods, 101
memory leakage, 93
memory leaks, hashmaps
WeakReference objects, 263
message formatters, 426–428
MessageFormat class, 426
applyPattern method, 428
format method, 427
messages
compound messages, 426, 428
simple messages, 426
meta-annotations, 196–198
Documented type, 198
Inherited type, 198
Retention type, 196
Target type, 196
metacharacters
patterns, 434
period metacharacter, 438
regular expressions, 438
metadata
annotations, 190–200, 225
files, 463
META-INF directory
distributing Four of a Kind as JAR file, 39
packages information, 247
method calls, 61
argument passing, 91
chained instance method calls, 92
operator, 53
Method class, 275–276
processing annotations, 199
method overloading, 74
method return, 73–74
method signature, 58
method-call stack, 90
method-call statement, 89
methods, 58–75, 94
abstract methods, 120–121
abstract reserved word, 121
arguments, 58, 60
calling, 89–92
calling overridden methods, 101
class methods, 60
compound statement, 60
covariant return type, 123–124
declaring, 58–60
empty statement, 61
enhancing an enum, 216, 217
final keyword, 101
forwarding, 114
generic methods, 212–214
getDeclaredMethods method, 271
getMethods method, 271
getters, 83

helper methods, 82
implementing, 60–74
implementing interfaces, 128
instance methods, 59
integer classes, 255
interfaces, 126
local variables, 61
main method, 60
naming conventions, 58
native methods, 492
Override annotation type, 191
overriding, 100
parameters, 58, 61
postconditions, 187–188
preconditions, 185–187
recursive calls, 90
resource-access methods, 403
return type, 58
returning value from, 73
setters, 83
simple statement, 60
statements, 60
strictfp applied to, 234
when not to use assertions, 188
Micro Edition (Java ME), 5
min method
BigDecimal class, 237
BigInteger class, 240
Collections class, 371
Math class, 229, 232
MIN_VALUE constant, 232, 251, 255
minus
unary minus operator, 55
MissingResourceException, 402, 403, 405
mkdir method, File, 459
mkdirs method, File, 459
modeling Four of a Kind in pseudocode, 21–23
Modifier class, 276
modifiers
getModifiers method, 276
module reserved word, 159
modules, 159
modulus (remainder) operator, 54
monitors, 299
multicatch, 171
multilevel feedback queue scheduler, 291
multiline comment, 59
multiple implementation (class) inheritance, 102
multiplication operator, 53
multiply method
BigDecimal class, 237
BigInteger class, 241
multithreaded applications
concurrency utilities, 381
mutable keys
IdentityHashMap class, 362
mutual exclusion, 299
threads, 302

N

name method
Charset class, 517
Enum class, 221
names
forName method, 270, 272, 273
getName method, 271, 275, 288, 289, 290
setName method, 289
namespaces, 151
naming conventions
catch clause, 169
classes, 44
exceptions, 165
fields, 45
interfaces, 125
local variables, 61
methods, 58
packages, 152
NaN (Not a Number), 232, 251
division by zero, 57
equals method, 253
isNaN method, 252
native code, 4
native methods, 492
native reserved word, 492
navigator area, NetBeans IDE, 16
negate method
BigDecimal class, 237
BigInteger class, 241
negation character class, 439
negative
unary minus operator, 55
NEGATIVE_INFINITY constant, 251
testing float/double for equality with -infinity, 253
NELEM constant, 350
nested classes, 139–150, 178
anonymous classes, 146–148
interfaces within classes, 150

local classes, 148–150
nonstatic member classes, 142–145
static member classes, 139–142
nested loops
labeled break statement, 72
nested types, 139–150
NetBeans IDE, 13–17
DumpArgs project, 14
editor area, 15
Files tab, 15
navigator area, 16
New Project dialog box, 14
Output tab, 16
projects area, 15
Projects tab, 15
Services tab, 15
tasks area, 16
tutorials, 13
networks, accessing, 530
New Java Project dialog box, 18
new operator, 95
creating objects and arrays, 85–87
New Project dialog box
NetBeans IDE, 14
new reserved word, 53
NEW value, State enum, 288
newFixedThreadPool method, Executors, 387
calculating Euler's number, 389
newInstance method, 272
newline
escape sequence for, 49
hard-coding \n escape sequence, 509
line terminator, 439
next line line terminator, 439
next method
BreakIterator class, 410, 411
Iterator class, 321, 322
ListIterator class, 327, 328
nextBoolean method, Random, 433
nextBytes method, Random, 433
nextDouble method, Random, 433
nextElement method, Enumeration, 372
nextFloat method, Random, 433
nextGaussian method, Random, 433
nextIndex method, ListIterator, 327, 328
nextInt method, Random, 433, 434
nextLong method, Random, 434
nextToken method, StringTokenizer, 502
nodes, 330
nonstatic member classes, 142–145
NoSuchElementException, 321, 327, 340, 347, 367, 368, 371
notify method, Object class, 103
thread communication, 303, 306
notifyAll method, Object, 103
thread communication, 303
NotSerializableException, 496, 497
null references
using assertions, 189
NullPointerException, 167, 275, 276, 317
Number class, 257
number formatters, 421–424
customized number formatters, 424
NumberFormat class, 235, 421
factory methods, 421
format method, 235
setXyz methods, 423
NumberFormatException, 254
numbers, aligning, 423
numRecs method, PartsDB, 470, 472

O

Object class, 103–112
clone method, 103, 104–106
equals method, 103, 107–109
finalize method, 103, 109–110, 257
getClass method, 103, 277
hashCode method, 103, 110–111
notify method, 103
notifyAll method, 103
toString method, 103, 111–112
wait method, 103
object creation operator, 53
object pools, implementing, 264
object reference
accessing fields, 88
calling methods, 89
object serialization/deserialization, 496–508
custom, 501–505
default, 496–501
externalization, 506–508
ObjectInputStream class, 497, 498
defaultReadObject method, 505
readObject method, 497, 498, 500, 507, 508
object-orientation
inheritance, 97–116
interfaces, 125–136
polymorphism, 116–124
ObjectOutputStream class, 497

defaultWriteObject method, 505
writeObject method, 497, 498, 507
objects, 85–94
see also classes
AccessibleObject class, 276
accessing fields, 87–89
calling methods, 89–92
collection, 315
creating objects and arrays, 85–87
encapsulation, 43
entities, 43
garbage collection, 92–93
initialization order, 78–82, 102
instantiation, 43
phantom reachable objects, 258
primitive types in Java, 47
reachable objects, 258
representing exceptions, 162–163
softly reachable objects, 258
strongly reachable objects, 258
template for manufacturing objects, 43
unreachable objects, 258
weakly reachable objects, 258
octal format
integer literals, 50
toOctalString method, 255
of methods, EnumSet, 339
offer method
BlockingQueue interface, 396
Queue interface, 347
one-dimensional array, 47
open range/interval
SortedSet interface, 343
operands, 51
different types in operator, 56
operators, 51
addition operator, 51
array index operator, 51
assignment operator, 51
binary operator, 51
bitwise AND operator, 51
bitwise complement operator, 51
bitwise exclusive OR operator, 51
bitwise inclusive OR operator, 52
cast operator, 52
compound assignment operator, 52
conditional AND operator, 52
conditional operator, 52
conditional OR operator, 52
division operator, 52
equality operator, 52
inequality operator, 52
infix operator, 51
instanceof operator, 108
left shift operator, 53
logical AND operator, 53
logical complement operator, 53
logical exclusive OR operator, 53
logical inclusive OR operator, 53
member access operator, 53
method call operator, 53
multiplication operator, 53
new operator, 85–87
object creation operator, 53
operands of different types, 56
overflow/underflow, 56
overloading, 3
postdecrement operator, 53
postfix operator, 51
postincrement operator, 54
precedence, 55
predecrement operator, 54
prefix operator, 51
preincrement operator, 54
relational greater than operators, 54
relational less than operators, 54
relational type checking operator, 54
remainder operator, 54
signed right shift operator, 54
string concatenation operator, 51, 55
subtraction operator, 55
ternary operator, 51
unary minus operator, 55
unary operator, 51
unary plus operator, 55
unsigned right shift operator, 55
OptionalDataException, 498
or method, BitSet, 373
OR operators
bitwise exclusive OR operator, 51, 374
bitwise inclusive OR operator, 52, 374
conditional OR operator, 52
logical exclusive OR operator, 53
logical inclusive OR operator, 53
ordinal method
Enum class, 221
Four of a Kind game, 34
output
FileWriter class, 518–520
System.out, 9
Writer class, 513
writer classes, 512–513

output stream, 473
Output tab, NetBeans IDE, 16
outputList method, generics, 209
OutputStream class, 473, 475–476
  methods, 475–476
OutputStreamWriter class, 514
  getEncoding method, 515
  write method, 514
overflow/underflow, operators, 56
overflows
  strictfp reserved word, 233
overloading
  annotating method override to detect, 101
  constructors, 75
  method overloading, 74
overloading operators, 3
overloading polymorphism, 116
@Override annotation, 191
  annotating overriding method, 101
  prefixing overriding methods, 102
overriding methods
  calling overridden methods, 101
  incorrectly overriding method, 101
  inheritance, 100
  overriding final method, 101
  prefixing overriding methods, 102

P

Package class, 243–247, 268
Package Explorer tab, Eclipse, 19
package reserved word, 152
package statement, 152–153, 179
PackageInfo application, 244–247
package-private access control level, 83
package-private methods
  including in Javadoc documentation, 36
packages, 151–160
  see also Java packages
  compile-time search, 154
  getPackage method, 271
  import statement, 153–154
  JAR files, 159–160
  logging library example, 155–159
  namespaces, 151
  naming conventions, 152
  package statement, 152–153
  runtime search, 155
  sealed packages, 243
    isSealed method, 244
  searching for packages and types, 154–155
  storing, 154
  unnamed package, 153
paragraph separator line terminator, 439
@param, Javadoc tag, 35
parameterized types, generics, 201, 202
  concrete parameterized type, 203
  wildcards, 208
parameters
  getParameterTypes method, 275
  methods, 58, 61
parametric polymorphism, 116
parent class *see* superclass
parent thread, 310
parentheses
  operator precedence, 55
parseBoolean method, 249
parseDouble method, 252, 254
parseFloat method, 252, 254
parseObject methods, 421
PartsDB class, 467, 470
pass-by-value
  argument passing, 91
path, 450
pathnames, 450
  absolute pathname, 451
  compareTo method, 462
  getAbsoluteFile method, 452
  getAbsolutePath method, 452
  getCanonicalPath method, 452
  getPath method, 452
  isAbsolute method, 453
  methods for learning about stored pathname, 452
  obtaining abstract pathname information, 453
  relative pathname, 451
  separator character, 451
Pattern class, 435
  compile method, 435, 439
  line terminators, 439
  matcher method, 435, 436
  methods, 435
pattern matching
  capturing groups, 440
pattern method, 435
patterns, 426
  apostrophe in, 427
  applyPattern method, 428
  Matcher class, 436

matching text against, 434
messages, 426
metacharacters, 434
quantifiers, 442
ResourceBundle class, 401
PatternSyntaxException, 435, 436, 437
peek method, Queue, 347
percentages, formatting numbers as, 421
performance
Collections class, 372
reflection, 269
threading, 296, 301, 302
threads, 287
unlimited serialization, 497
using StringBuilder not StringBuffer, 284
period metacharacter, 438
phantom reachable objects, 258
PhantomReference class, 263–266, 268
phantom reachable objects, 258
PI constant, 227
PipedInputStream class, 481, 482
PipedOutputStream class, 481, 482
platforms
Android platform, 5
Java, 3
Java EE (Enterprise Edition), 5
Java ME (Micro Edition), 5
Java SE (Standard Edition), 5
plus
unary plus operator, 55
Point class
equals method, 107
pointers, 3
poll method
BlockingQueue interface, 396
Queue interface, 347, 349
ReferenceQueue class, 260, 265
polymorphism, 116–124
coercion polymorphism, 116
downcasting, 121–123
generics, 208
overloading polymorphism, 116
parametric polymorphism, 116
runtime type identification, 121–123
subtype polymorphism, 117
pooling
implementing object pools, 264
pop
method-call stack, 90
stack, 225
pop method, Stack, 372
portability
hard-coding \n escape sequence, 509
Java, 4
Java applications, 46
strictfp reserved word, 233, 234
positive
unary plus operator, 55
POSITIVE_INFINITY constant, 251
testing float/double for equality with +infinity, 253
possessive quantifier, 442, 443, 444
postconditions
design-by-contract development, 187–188
postdecrement operator, 53
postfix operator, 51
postincrement operator, 54
precedence, operators, 55
preceding method, BreakIterator, 411
precision, decimals, 235
precision method, BigDecimal, 237
preconditions
design-by-contract development, 185–187
predecrement operator, 54
predefined character classes, 440
preemptive scheduling, 291
preferences, 428
java.util.prefs package, 429
storing in Windows XP registry, 430, 431
system preference tree, 429
Preferences APIs, 428–431
Preferences class, 429
methods, 429–430
prefix operator, 51
preincrement operator, 54
previous method
BreakIterator class, 411
ListIterator class, 327, 328
previousIndex method, ListIterator, 327, 328
primitive types, 46
autoboxing and unboxing, 323
portability of Java applications, 46
storing in collections, 323
primitive wrapper classes, 247–257, 268
Boolean class, 248–250
Byte class, 255–257
Character class, 250–251
constants, 232
Double class, 251–254
Float class, 251–254

Integer class, 255–257
Long class, 255–257
Short class, 255–257
storing primitive types in collections, 323
superclass of, 257
print method
PrintStream class, 508
System.out, 9, 509
printf method
PrintStream class, 509
format specifiers, 510
System.out, 510
println method
PrintStream class, 508
System.out, 9, 509, 511
printStackTrace method, Throwable, 164, 168
PrintStream class, 473, 474, 508–511
checkError method, 511
flush method, 511
print method, 508
printf method, 509
format specifiers, 510
println method, 508
System.err, 508
System.out, 508
PrintWriter class, 474
priority
setPriority method, 291
priority queues, 346, 348
PriorityQueue class, 348–350
private access control level, 83
inheritance, 100
private reserved word, 83
problem domain
Four of a Kind application, 21
profiling, 132
projects area, NetBeans IDE, 15
projects directory, JDK, 7
Projects tab, NetBeans IDE, 15
properties
getProperty method, 285, 286, 372
setProperty method, 372
Properties API, 429
Properties class, 372, 375, 429
properties files, 403
property resource bundle, 403–405
PropertyResourceBundle class, 402, 403
protected access control level, 82
protected reserved word, 82
pseudocode, 21
converting to Java code, 23–36
modeling Four of a Kind in, 21–23
public access control level, 82, 83
accessing fields, 88
public methods
when not to use assertions, 188
public reserved word, 82
interfaces, 126
main method, 60
push
method-call stack, 90
stack, 225
push method, Stack, 372
put method
BlockingQueue interface, 393, 396
Map interface, 352
Preferences class, 430
putAll method, Map, 352
putBack method, Four of a Kind game, 28

Q

quantifiers, 442–444
greedy quantifier, 442, 443
possessive quantifier, 442, 443, 444
reluctant quantifier, 442, 443
zero-length matches, 443
queue data structure, 204
Queue implementation classes, 347
Queue interface, 346–347
BlockingQueue subinterface, 392
hierarchy of interfaces, 316
methods, 346–347
PriorityQueue class, 348–350
queues, 346
first-in, first-out (FIFO), 346
last-in, first-out (LIFO), 346
priority queues, 346, 348
PriorityQueue class, 348–350
Queue<E> generic type, 203–205
quote method, Pattern, 435
quotes, escape sequences for, 49

R

r mode argument
RandomAccessFile constructor, 463
race condition, threads, 298
Random class, 432–434
methods, 433–434

random method, Math class, 229, 231, 432
random number generation, 432–434, 447
  seed value, 432
RandomAccessFile class, 462–473
  close method, 464, 470
  constructors, 462–464
  creating flat file database, 467–472
  file pointer, 464
  getFD method, 464, 465, 466
  methods, 464–465
  read-prefixed methods, 466
  skipBytes method, 465, 466
  write-prefixed methods, 466
range character class, 439
range method, EnumSet, 339
range-view operations
  List interface, 328
Rank enum, Four of a Kind game, 26, 34
Rank object, Four of a Kind game
  ordinal method, 34
rank method, Four of a Kind game, 26
rankCounts array, Four of a Kind game, 34
raw type, generics, 203
reachable objects, 258
read method
  BufferedInputStream class, 492, 493
  FileInputStream class, 488, 489, 490, 491, 492
  InputStreamReader class, 515
  IntputStream class, 477
  RandomAccessFile class, 464
  Reader class, 513, 514
  System.in, 10
readChar method, RandomAccessFile, 464
Reader class, 513
  read method, 513, 514
  ready method, 514
reader classes, 512–513
  BufferedReader class, 522
  FileReader class, 520–522
  InputStreamReader class, 515
readers (and writers), 511–529
readExternal method, 506, 508
readInt method, RandomAccessFile, 465
readLine method, BufferedReader, 522
readObject method, ObjectInputStream, 497, 498, 500, 505, 507, 508
read-only fields, 57–58
ready method, Reader, 514
recursive calls, methods, 90
recursive type bound, generics, 208
ReentrantLock class, 394
refactoring, 71
Reference class, 259–260
reference classes
  PhantomReference class, 263–266
  SoftReference class, 260–263
  WeakReference class, 263
reference variable, 86
Reference<T> generic type, 259
referenced objects
  garbage collection, 92
  identity check, 106
ReferenceQueue class, 259, 260
ReferenceQueue<T> generic type, 260
References API, 257–266, 268
references, root set of, 257
reflection
  identifying class members, 272
  performance, 269
Reflection API, 269–277
  Class class, 270–274
reflexive rule
  Comparable interface, 317
region method, matchers, 437
regionEnd method, 437
regionStart method, 437
regular expressions, 434, 437–444
  boundary matchers, 441–442
  capturing groups, 440–441
  character classes, 439–440
  matching phone numbers, 444
  matching text against patterns, 434
  metacharacters, 434, 438
  predefined character classes, 440
  quantifiers, 442–444
Regular Expressions API, 434–447
reification, generics, 211
RejectedExecutionException, 382
relational greater than operator, 54
relational less than operator, 54
relational type checking operator, 54
relationships
  "has a" relationship, 112
  "is a" relationship, 98
relative pathnames, 451
reluctant quantifier, 442, 443
remainder method
  BigDecimal class, 237
  BigInteger class, 241
remainder operator, 54
remove method

Collection interface, 319, 321
Iterator class, 321
List interface, 326
ListIterator class, 327
Map interface, 352
Queue interface, 347
ReferenceQueue class, 260
ThreadLocal class, 309
removeAll method, Collection, 320, 321
renameTo method, File, 460
replace method, String, 279
ReportCreationException, 171
reserved words
getModifiers method, 276
list of reserved identifiers, 44
primitive types, 46
reserved words, list of
abstract, 121
assert, 182
break, 65, 70
case, 65
catch, 168
continue, 71
default, 65
do, 69
else, 62
enum, 215
extends, 98, 130, 205
false, 49
final, 57
finally, 110, 172
for, 66
if, 62
implements, 127
import, 153
instanceof, 54
int, 45
interface, 125
module, 159
native, 492
new, 53
package, 152
private, 83
protected, 82
public, 82
return, 73
static, 48
strictfp, 233
super, 99, 207
switch, 64
synchronized, 394
this, 76, 85
throw, 167
throws, 110, 167
true, 49
try, 110
void, 59
volatile, 302, 303
while, 67
reset method
IntputStream class, 477
Matcher class, 437, 522
resource bundles, 400–409
accessing resources, 403
candidate bundle names, 401, 402
Control class, 403
list resource bundle, 405–409
ListResourceBundle class, 402
properties files, 403
property resource bundle, 403–405
PropertyResourceBundle class, 402
resource management, automatic, 176
resource-access methods, 403
ResourceBundle class, 401
clearCache methods, 403
getBundle methods, 401, 402
setParent method, 402
resurrection, 264
implementing object pools, 264
retainAll method, Collection, 320, 321
Retention meta-annotation type, 196, 197
RetentionPolicy enum, 196
return (carriage return)
escape sequence for, 49
@return, Javadoc tag, 35
return reserved word, 73
return statement
method return, 73–74
return type
covariant return type, 123–124
getReturnType method, 276
methods, 58
reverse method
Collections class, 371
StringBuffer class, 283
right shift operators
signed, 54
unsigned, 55
rnd method, Math, 186
converting random() return value, 231
root directories
listRoots method, 449

root directory, Windows
backslash representing, 159
root set of references, 257
round method, Math, 229
BigDecimal floating-point calculations, 234
round robin scheduling, 291
ROUND_-prefixed constants
BigDecimal class, 236
rounding, decimals
setScale method, 237
RoundingMode enum, 237
RSA encryption
BigInteger class, 243
RTTI (runtime type identification), 121–123
marker/tagging interface, 126
RuleBasedCollator class, 414
run method, Runnable interface, 288, 290, 382, 385
countdown latches, 391
runFinalization method, System, 285
Runnable interface, 288, 382
Executor interface limitations, 382
run method, 288, 290, 382, 385
runnable tasks, 382
execute method, 382
submit method, 384
RUNNABLE value, State enum, 288
runtime exceptions, 165
RUNTIME retention policy, 196, 197
runtime search, 155
runtime type identification *see* RTTI
RuntimeException, 165, 179
rw/rwd/rws mode arguments
RandomAccessFile constructor, 463

S

\s predefined character class, 440
scale method, BigDecimal, 237
scale, decimals, 235
setScale method, 237
ScheduledExecutorService instance
Executors class returning, 387
ScheduledExecutorService interface, 386
schedulers, 291
multilevel feedback queue scheduler, 291
preemptive scheduling, 291
round robin scheduling, 291
scope, 48
local variables, 61
type parameter scope, 208
sealed packages, 243
isSealed method, 244
searches, binary/linear, 370
searching for packages and types, 154–155
compile-time search, 154
runtime search, 155
security
Java, 5
unlimited serialization, 496
working with, 531
seed value, 432
setSeed method, 433
seek method, RandomAccessFile, 465
select method, PartsDB, 470
selector expression, switch statement, 64
Semaphore class, 390
semaphores, synchronizers, 390
sentence
getSentenceInstance method, 409
separator character, 450, 451
separators
line terminators, 508
sequence, List interface, 325
Serializable interface, 496, 497
serialization, 496–508
custom, 501–505
default, 496–501
externalization, 506–508
NotSerializableException, 496
unlimited serialization, 496
serialver tool, 500
Services tab, NetBeans IDE, 15
servlets, Java EE, 5
Set interface, 332
add method, 332
EnumSet class, 337–339
equals method, 332
hashCode method, 332
HashSet class, 333–337
hierarchy of interfaces, 316
LinkedHashSet class, 337
TreeSet class, 332–333
set method
BitSet class, 374
Calendar class, 418
List interface, 326
ListIterator class, 328
ThreadLocal class, 309
setAccessible method, 276

setCharAt method, StringBuffer, 283
setDaemon method, Thread, 289
setDefault method, locales, 399
setDefaultUncaughtExceptionHandler method, 295
setExecutable method, File, 461
setLastModified method, File, 460
setLength method
  RandomAccessFile class, 465
  StringBuffer class, 283
setMaximumFractionDigits method, 423
setMaximumIntegerDigits method, 423
setMinimumFractionDigits method, 423
setMinimumIntegerDigits method, 423
setName method, Thread, 289
setParent method, ResourceBundle, 402
setPriority method, Thread, 291
setProperty method, Properties, 372
setReadable method, File, 461
setReadOnly method, File, 460
sets
  bitsets, 337
  EnumSet class, 337–339
  HashSet class, 333–337
  LinkedHashSet class, 337
  sorted sets, 339
  TreeSet class, 332–333
setScale method, BigDecimal, 236, 237, 239
setSeed method, Random, 433
setSharedChar method, Object, 305, 306
setters, 83
setText method, BreakIterator, 411, 412
setTime method
  Calendar class, 420
  Date class, 416
setTimeZone method, DateFormat, 425
setTopCard method, Four of a Kind game, 29
setUncaughtExceptionHandler method, 295
setValue method, Map, 355
setWritable method, File, 461
shadowing field
  accessing fields, 88
shallow copying/cloning, 104
shift operators
  left shift operator, 53
  signed right shift operator, 54
  unsigned right shift operator, 55
Short class, 255–257
SHORT constant, DateFormat, 424
Short integer type, 46, 233
  getShort method, 275
short-circuiting, 52
shortValue method, 255
showMessageDialog method, JOptionPane, 408
shuffle method
  Collections class, 434
  Four of a Kind game, 28
shutdown method, ExecutorService, 384
shutdownNow method, 384, 389
  countdown latches, 391
signature, methods, 58
signed right shift operator, 54
signum method, Math, 230
simple expressions, 48–50
simple messages, 426
simple statement, 60
SimpleDateFormat class, 425
SimpleTimeZone class, 417
sin method, Math, 230
single inheritance, 102
single quote (')
  escape sequence for, 49
single-line comment, 45
  Four of a Kind game, 29
singletonList method, Collections, 371
size method
  BitSet class, 374
  Collection interface, 320
  Map interface, 353
skip method, IntputStream, 477
skipBytes method, RandomAccessFile, 465, 466
sleep method, Thread, 289, 292, 298, 299
sockets, 473
softly reachable objects, 258
SoftReference class, 260–263, 268
  image cache, 261–262
  soft reachable objects, 258
sort method
  Arrays class, 370
  Bubble sort, 187
sorted sets, 339
  compare method, 345
  compareTo method, 345
  ordering, 345
SortedMap interface, 367–369
  hierarchy of interfaces, 316
  keySet method, 368, 517
  methods, 367–368
SortedSet interface, 339–346

closed range/interval, 343
comparator method, 340, 342
equals method, 345, 346
first method, 340
headSet method, 340, 341, 343
hierarchy of interfaces, 316
iterator method, 339
last method, 340
open range/interval, 343
subSet method, 340, 341, 343
tailSet method, 341, 343
toArray method, 339
toString method, 340
source code
annotating, 193–195
refactoring, 71
representing exceptions in, 162–166
SOURCE retention policy, 196
SPI (Service Provider Interface), 398
split method
Pattern class, 435
String class, 199, 279, 281
sqrt method, Math, 230
Stack class, 372, 375
stacks, 225
method-call stack, 90
threads, 287
standard class library, 4
Standard Edition (Java SE), 5
Standard I/O, 8
start method
Matcher class, 436
Thread class, 289, 290
startsWith method, String, 280
State enum
getState method, 288
statements, 60, 61, 94
assertions, 181–190
assignment, 61
break statement, 65, 70
case statement, 65
compound statement, 60
continue statement, 71
do-while loop, 69
empty statement, 61
for loop, 66–67
goto statement, 72
if-else statement, 62–64
import statement, 153–154
labeled break statement, 72
labeled continue statement, 72
method call statement, 61, 89
package statement, 152–153
simple statement, 60
switch statement, 64–66
while loop, 67–68
static import statement, 161
static imports, 160–161
static member classes, 139–142
static reserved word, 48
class initialization, 77
constructors, 75
main method, 60
read-only class field, 57
stop method
StoppableThread class, 301
Thread class, 296
StoppableThread class, 301–302
stopThread method, 301, 302
stream classes, 473–475
BitStreamInputStream, 492
BitStreamOutputStream, 492
BufferedInputStream, 492, 493
BufferedOutputStream, 492
ByteArrayInputStream, 477, 478
ByteArrayOutputStream, 477, 478–479
DataInputStream, 494
DataOutputStream, 494
FileInputStream, 479, 479–481
FileOutputStream, 479, 479
FilterInputStream, 488, 491
FilterOutputStream, 485
InputStream, 473, 476–477
LineNumberInputStream, 474
LineNumberReader, 474
ObjectInputStream, 497, 498
ObjectOutputStream, 497
OutputStream, 473, 475–476
PipedInputStream, 481, 482
PipedOutputStream, 481, 482
PrintStream, 473, 474, 508–511
PrintWriter, 474
StringBufferInputStream, 474
StringReader, 474
ZipOutputStream, 475
stream destinations, Java, 473
stream sources, Java, 473
stream unique identifier (SUID), 500
streams, I/O, 473–511
chaining streams, 488, 491, 493
FileReader class, 520–522
FileWriter class, 518–520

InputStreamReader class, 515
java.util.jar package, 475
java.util.zip package, 474
OutputStreamWriter class, 514
strictfp reserved word, 233
applied to methods, 234
used in class header declaration, 234
StrictMath class, 233–234, 268
String class, 278–281
charAt method, 278, 281
compareTo method, 413
intern method, 279, 280
Java, 86
length method, 279, 281
methods, 278–280
split method, 199, 279, 281
string concatenation operator, 51, 55
string literal, 48, 280
escape sequences, 49
StringBuffer class, 281–284
append methods, 282, 284
immutability of string objects, 278
methods, 281–283
using StringBuilder class instead, 284
StringBufferInputStream class, 474
StringBuilder class
using instead of StringBuffer, 284
StringReader class, 474
strings
see also toString method
string management, 277–284
immutability of string objects, 278
StringTokenizer class, 501, 503
countTokens method, 502
nextToken method, 502
problematic deserialization, 502
strongly reachable objects, 258
Stub annotation type
adding elements to, 194
annotating Deck class, 197
declaring, 193
meta annotations, 197
meta-annotations, 196
processing annotations, 198
retention policies affecting, 197
StubFinder application
processing annotations, 198–199
subclasses
filter output streams, 485
initialization order, 102
"is a" relationship, 98
subList method, 326, 328
subMap method, 368
submit method, ExecutorService, 384, 386
subSequence method, 436
Subset class, 142
subSet method, 340, 341, 343
substring method
String class, 280
StringBuffer class, 283
subtract method
BigDecimal class, 237
BigInteger class, 241
subtraction character class, 440
subtraction operator, 55, 56
subtype polymorphism, 117
upcasting and late binding, 117–119
Suit enum, Four of a Kind game, 26
suit method, Four of a Kind game, 26
super reserved word
calling overridden methods, 101
finalize method, 109
inheritance, 99
type parameter bounds, 207
superclass
fragile base class problem, 114
getSuperclass method, 272
initialization order, 102
"is a" relationship, 98
Object class, 103–112
SuppressWarnings annotation type, 191, 193
declaring and using generic types, 205
meta-annotations, 196
Swing, 407, 408
JOptionPane class, 408
switch reserved word, 64
switch statement, 64–66
break reserved word, 65
case reserved word, 65
default reserved word, 65
selector expression, 64
symmetric rule
Comparable interface, 317
sync method, FileDescriptor, 466
synchronization, locks, 299, 394
synchronization, threads, 296–311
attempting to stop threads, 300
communication between threads, 300, 302
generating unique filenames, 299
long and double types, 300

monitors, 299
mutual exclusion, 299, 302
synchronized reserved word, 394, 397
synchronizedSet method, 371
synchronizers, 390–392, 447
countdown latches, 390
cyclic barriers, 390
exchangers, 390
semaphores, 390
System class, 284–287
arraycopy method, 285, 286
currentTimeMillis method, 285, 286, 415
getProperty method, 285, 286, 508
methods, 284–285
system preference tree, 429
System.err, 10
PrintStream class, 508
System.in
read method, 10
System.out, 9
print method, 509
printf method, 510
println method, 509, 511
PrintStream class, 508
systemNodeForPackage method, 430
systemRoot method, Preferences, 430

T

tab, horizontal
escape sequence for, 49
tagging interface, 126
tailSet method, SortedSet, 341, 343
take method, BlockingQueue, 393, 396
tan method, Math, 230
Target meta-annotation type, 196, 197
tasks, 382
callable tasks, 382, 385
decoupling submission from execution, 382
Executor interface limitations, 382
invokeAll method, 383
invokeAny method, 383, 384
isTerminated method, 384
RejectedExecutionException, 382
runnable task, 382
shutdown method, 384
shutdownNow method, 384
submit method, 384
tasks area, NetBeans IDE, 16
TERMINATED value, State enum, 288
text
detecting logical boundaries within, 409
matching against patterns, 434
Thai Buddhist calendar, 417
this reserved word, 85
accessing fields, 88
constructors, 76
Thread class, 288–289
currentThread method, 288, 289, 290
deprecated methods, 296
getName method, 288, 289, 290
getThreadGroup method, 294
isAlive method, 289, 293
join method, 289, 293, 294
methods, 288–289
setDefaultUncaughtExceptionHandler method, 295
setPriority method, 291
setUncaughtExceptionHandler method, 295
sleep method, 289, 292, 298, 299
start method, 289, 290
stop method, 296
yield method, 296
thread pools, 387
newFixedThreadPool method, 387
ThreadFactory instance
Executors class returning, 387
ThreadGroup class, 294
threading
performance, 296, 301, 302
Threading API, 287–311
concurrency utilities, 381
Runnable interface, 288
ThreadLocal class, 308, 309
initialValue method, 309, 310
methods, 309
thread-local variables, 308, 310
threads, 287
attempting to stop threads, 300
child thread, 310
countdown latches, 390
cyclic barriers, 390
daemon thread, 289
deadlock, 306–308
different user IDs for, 309
event-dispatching thread, 407
exchangers, 390
lock method, 396
Lock objects, 394
mutual exclusion, 302

notify method, 103
notifyAll method, 103
parent thread, 310
performance, 287
race condition, 298
returning unique identifiers, 397
semaphores, 390
stacks, 287
StoppableThread class, 301
stopThread method, 301, 302
strongly reachable objects, 258
submitting callable task to executor, 386
synchronization, 296–311, 390
triggering coordinated start, 390
user thread, 289
volatile thread communication, 302
wait method, 103
throw reserved word, 167
throw statement, 167, 179
Throwable class, 163–164
methods, 163–164
printStackTrace method, 164, 168
throwing exceptions, 163, 166–168
final rethrow, 172
rethrowing exceptions, 172
throws clause, 167, 168, 179
throws Exception clause
processing annotations, 199
throws reserved word, 167
finalize method, 110
time
currentTimeMillis method, 285, 286, 415
getTime method, 416
GMT (Greenwich Mean Time), 415
Joda Time, 420
setTime method, 416
Unix epoch, 425
UTC (Coordinated Universal Time), 415
time zones
daylight saving time, 417
internationalization, 415
SimpleTimeZone class, 417
TIMED_WAITING value, State enum, 288
timeout, 383, 384, 385
TimeoutException, 384, 385
TimeUnit enum, 385
TimeZone class, 417
getDefault method, 417
getTimeZone method, 417, 425
title, packages
getImplementationTitle method, 243
getSpecificationTitle method, 244
toAlignedBinaryString, Integer, 256
toArray method
Collection interface, 320
SortedMap interface, 367
SortedSet interface, 339
toBinaryString method, Integer, 255, 256
toByteArray method, 478, 479
toCharArray method, 280
toDegrees method
Math class, 230
StrictMath class, 233
toHexString method, Integer, 255
tokens, enums, 218
toLowerCase method
Character class, 251
String class, 280
toOctalString method, Integer, 255
Toolkit class, 406
topCard method, Four of a Kind game, 29
top-level classes, 139
partitioning of, 151, 179
toRadians method
Math class, 230
StrictMath class, 233
toString method
BigDecimal class, 237, 239
BigInteger class, 241
BitSet class, 374
Boolean class, 249
CharSequence interface, 436
Date class, 416, 417
Enum class, 216, 218, 220
File class, 453
Integer class, 256
Object class, 103, 111–112
Pattern class, 435
SortedMap interface, 367
SortedSet interface, 340
StringBuffer class, 283
touch program, Unix, 526
toUpperCase method
Character class, 251
String class, 280
transitive rule
Comparable interface, 317
translations
Yahoo! Babel Fish, 408
TreeMap class, 355–356
trees
system preference tree, 429

user preference tree, 429
TreeSet class, 332–333
sorted sets, 339
trim method, String, 280
true reserved word, 49
try reserved word
finalize method, 110
handling exceptions, 168
try statement, 168, 169
tryLock method, 394
two's complement format, 239
two-dimensional array, 47
type checking operators
relational type checking operator, 54
type conversion
coercion polymorphism, 116
type parameter, generics, 203
bounds, 205–207
erasure, 211
masking, 208
recursive type bound, 208
scope, 208
type safety
enums, 216
generics, 200–202
wildcards, 209
problem with enumerated type, 215
types
abstract types, 133
compile-time search, 154
covariant return type, 123–124
declaring classes, 44
fields, 45
generic types, 202–212
nested types, 139–150
primitive types, 46
runtime search, 155
runtime type identification, 121–123
searching for packages and, 154–155

U

unary operator, 51
unary plus operator, 55
unboxing, 323–324
unchecked exceptions, 165, 168, 179
underflows
strictfp reserved word, 233
underscores
integer literals, 50
Unicode, 44
Character class, 250
character sets and encodings, 512
escape sequences, 49
internationalization, 398
UTF (Unicode Transformation Format), 512
Unicode 0, 46
union character class, 439
Unix epoch, 415, 425
formatting, 425
unlimited serialization, 496
unlock method, 396
unmodifiableMap method, 371
unnamed package, 153
UNNECESSARY constant, RoundingMode enum, 238
unreachable objects
garbage collection, 258
unreferenced objects
garbage collection, 92
unsigned right shift operator, 55
UnsupportedOperationException, 320
UP constant, RoundingMode enum, 238
upcasting
subtype polymorphism, 117–119
update method, PartsDB, 470
uppercase
isUpperCase method, 251
toUpperCase method, 251
user preference tree, 429
user thread, 289
userNodeForPackage method, 430
userRoot method, 430
users
different user IDs for different threads, 309
UTC (Coordinated Universal Time), 415, 425
UTF (Unicode Transformation Format), 512
utilities
concurrency utilities, 381–397
utility classes, 234
Arrays class, 369–371
Collections class, 371–372
collections framework, 315, 369–372

V

valid method, FileDescriptor, 466
value classes *see* primitive wrapper classes
value method
processing annotations, 199

valueOf method
Boolean class, 249
Enum class, 221, 222
Integer class, 324
values
containsValue method, 351
values method
Enum class, 218, 219, 221
Map interface, 353
SortedMap interface, 368
variables
atomic variables, 397
declaring in classes, 45
free variable, 148
isVarArgs method, 276
local variables, 61
loop-control variables, 66
reference variable, 86
thread-local variable, 308
Vector class, 372, 375
elements method, 372
vendor, packages
getImplementationVendor method, 243
getSpecificationVendor method, 244
version, packages
getImplementationVersion method, 243
getSpecificationVersion method, 244
views
List interface, 328
range-view operations, 328
virtual machine, 3
bytecode verifier, 4
classloader, 4
Dalvik virtual machine, 5
finalize method, 109
interpreter, 4
javac loading, 7
void reserved word, 59
volatile reserved word, 302, 303, 484

W

\w predefined character class, 440
wait method, Object class, 103
thread communication, 303, 306
WAITING value, State enum, 288
warnings
deprecation warnings, 192, 193
SuppressWarnings annotation type, 193
WeakHashMap class, 364–365
weakly reachable objects, 258
WeakReference class, 263, 268
weak reachable objects, 258
web browser
viewing Javadoc in, 35
while loop, 67–68
while reserved word, 67
do-while loop, 69
whitespace
toWhitespace method, 251
wildcards, generics, 208–210
actual type arguments, 203
windowing toolkits
Swing, 407, 408
word processing
detecting logical boundaries within text, 409
words
getWordInstance method, 409
workbench, Eclipse, 19
wrapped exception (cause), 164
wrapper class, 115
primitive wrapper classes, 247–257
storing primitive types in collections, 323
wrapping exceptions, 171
write method
BufferedOutputStream class, 492, 493
FilterOutputStream class, 485, 486, 492
OutputStream class, 475, 476
OutputStreamWriter class, 514
RandomAccessFile class, 465
Writer class, 513
writeChars method, RandomAccessFile, 465
writeExternal method, 506, 507
writeInt method, RandomAccessFile, 465
writeObject method, ObjectOutputStream, 497, 498, 505, 507
Writer class, 513
writer classes, 512–513
FileWriter class, 518–520
OutputStreamWriter class, 514
PrintWriter class, 474
writers (and readers), 511–529

X

Xlets, Java ME, 5
XML documents
parsing and creating, 530
xor method, BitSet, 374

Y

Yahoo! Babel Fish, 408
yield method, Thread, 296

Z

\Z boundary matcher, 441
zero, division by, 56
zero-length matches, 442
    boundary matchers, 442
    quantifiers, 443
zip files
    java.util.zip package, 474
    ZipOutputStream class, 475

Companion eBook Available

Learn Java for Android Development

Friesen

Apress

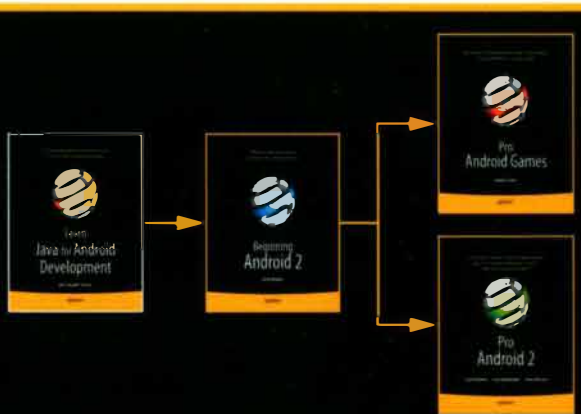
156

The Android platform is hot – and with more than 70,000 apps already available, the development market is sizzling. *Learn Java for Android Development* teaches you most of the essential Java language features, APIs, and tools that you need to know before you can become an effective Android developer.

This book explains nearly every pre-Java 7 language feature (native methods are not covered), and most of those that are new in Java 7 (such as switch on string, multicatch, and final rethrow), illustrating them with relevant code examples. Starting with classes and objects, you progress to object-oriented features (such as interfaces), and advanced features organized into nested types, packages, static imports, exceptions, assertions, annotations, generics, enums, and miscellaneous categories. Continuing, you investigate various foundational APIs, including Mathematics, Packages, Primitive Wrapper Classes, References, Reflection, String Management, System, Threading, Collections, Concurrency, Internationalization, Preferences, Random, Regular Expressions, and I/O.

To round off the book's coverage of Java programming, you will learn how to use the JDK's javac (compiler), java (application launcher), javadoc (Java documentation generator), and jar (Java archive creator, updater, and extractor) tools. There is also an introduction to the NetBeans and Eclipse integrated development environments.

Each of the book's ten chapters provides an exercise section that helps to reinforce your understanding of the chapter's material. (Answers to the approximately 300 exercises are provided). Although topics such as networking and JDBC are not covered (for brevity), you will be able to freely download (over the next few months) additional PDF-based chapters on these topics (and more) from my javajeff.mb.ca website.



COMPANION eBOOK
SEE LAST PAGE FOR DETAILS ON $10 eBOOK VERSION

Apress®
SOURCE CODE ONLINE
www.apress.com

Shelve in
Programming Languages/Java and
Mobile Computing

157

User level:
Beginner

ISBN 978-1-4302-3156-1
9 781430 231561