Visit informit.com/thejavaseries for a complete list of available publications.

Since 1996, when Addison-Wesley published the first edition of *The Java Programming Language* by Ken Arnold and James Gosling, this series has been the place to go for complete, expert, and definitive information on Java technology. The books in this series provide the detailed information developers need to build effective, robust, and portable applications and are an indispensable resource for anyone using the Java platform.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>xxiii</td>
</tr>
<tr>
<td>About the Authors</td>
<td>xxvii</td>
</tr>
<tr>
<td><strong>Chapter 1</strong> Getting Started</td>
<td>1</td>
</tr>
<tr>
<td>The Java Technology Phenomenon</td>
<td>1</td>
</tr>
<tr>
<td>The Java Programming Language</td>
<td>2</td>
</tr>
<tr>
<td>The Java Platform</td>
<td>2</td>
</tr>
<tr>
<td>What Can Java Technology Do?</td>
<td>4</td>
</tr>
<tr>
<td>How Will Java Technology Change My Life?</td>
<td>4</td>
</tr>
<tr>
<td>The “Hello World!” Application</td>
<td>5</td>
</tr>
<tr>
<td>“Hello World!” for the NetBeans IDE</td>
<td>6</td>
</tr>
<tr>
<td>“Hello World!” for Microsoft Windows</td>
<td>15</td>
</tr>
<tr>
<td>“Hello World!” for Solaris and Linux</td>
<td>20</td>
</tr>
<tr>
<td>A Closer Look at the “Hello World!” Application</td>
<td>23</td>
</tr>
<tr>
<td>Source Code Comments</td>
<td>24</td>
</tr>
<tr>
<td>The HelloWorldApp Class Definition</td>
<td>25</td>
</tr>
<tr>
<td>The main Method</td>
<td>25</td>
</tr>
<tr>
<td>Common Problems (and Their Solutions)</td>
<td>27</td>
</tr>
<tr>
<td>Compiler Problems</td>
<td>27</td>
</tr>
<tr>
<td>Runtime Problems</td>
<td>29</td>
</tr>
<tr>
<td>Questions and Exercises: Getting Started</td>
<td>31</td>
</tr>
</tbody>
</table>
## Chapter 4  Classes and Objects

**Classes**
- Declaring Classes 89
- Declaring Member Variables 90
- Defining Methods 92
- Providing Constructors for Your Classes 94
- Passing Information to a Method or a Constructor 95

**Objects**
- Creating Objects 100
- Using Objects 104

**More on Classes**
- Returning a Value from a Method 107
- Using the this Keyword 109
- Controlling Access to Members of a Class 110
- Understanding Class Members 112
- Initializing Fields 116
- Summary of Creating and Using Classes and Objects 118
- Questions and Exercises: Classes 119
- Questions and Exercises: Objects 120

**Nested Classes**
- Why Use Nested Classes? 122
- Static Nested Classes 122
- Inner Classes 123
- Shadowing 123
- Serialization 124
- Inner Class Example 125
- Local and Anonymous Classes 127
- Modifiers 127
- Local Classes 127
- Anonymous Classes 131
- Lambda Expressions 136
- When to Use Nested Classes, Local Classes, Anonymous Classes, and Lambda Expressions 155
- Questions and Exercises: Nested Classes 156

**Enum Types**
- Questions and Exercises: Enum Types 161

## Chapter 5  Annotations

**Annotations**
- Annotations Basics 164
- The Format of an Annotation 164
<table>
<thead>
<tr>
<th>Where Annotations Can Be Used</th>
<th>165</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaring an Annotation Type</td>
<td>165</td>
</tr>
<tr>
<td>Predefined Annotation Types</td>
<td>167</td>
</tr>
<tr>
<td>Annotation Types Used by the Java Language</td>
<td>167</td>
</tr>
<tr>
<td>Annotations That Apply to Other Annotations</td>
<td>169</td>
</tr>
<tr>
<td>Type Annotations and Pluggable Type Systems</td>
<td>170</td>
</tr>
<tr>
<td>Repeating Annotations</td>
<td>171</td>
</tr>
<tr>
<td>Step 1: Declare a Repeatable Annotation Type</td>
<td>172</td>
</tr>
<tr>
<td>Step 2: Declare the Containing Annotation Type</td>
<td>172</td>
</tr>
<tr>
<td>Retrieving Annotations</td>
<td>173</td>
</tr>
<tr>
<td>Design Considerations</td>
<td>173</td>
</tr>
<tr>
<td>Questions and Exercises: Annotations</td>
<td>173</td>
</tr>
<tr>
<td>Questions</td>
<td>173</td>
</tr>
<tr>
<td>Exercise</td>
<td>174</td>
</tr>
<tr>
<td>Answers</td>
<td>174</td>
</tr>
</tbody>
</table>

**Chapter 6** Interfaces and Inheritance

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>175</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces in Java</td>
<td>176</td>
</tr>
<tr>
<td>Interfaces as APIs</td>
<td>177</td>
</tr>
<tr>
<td>Defining an Interface</td>
<td>177</td>
</tr>
<tr>
<td>Implementing an Interface</td>
<td>178</td>
</tr>
<tr>
<td>Using an Interface as a Type</td>
<td>180</td>
</tr>
<tr>
<td>Evolving Interfaces</td>
<td>181</td>
</tr>
<tr>
<td>Default Methods</td>
<td>182</td>
</tr>
<tr>
<td>Summary of Interfaces</td>
<td>192</td>
</tr>
<tr>
<td>Questions and Exercises: Interfaces</td>
<td>193</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inheritance</th>
<th>193</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Java Platform Class Hierarchy</td>
<td>194</td>
</tr>
<tr>
<td>An Example of Inheritance</td>
<td>195</td>
</tr>
<tr>
<td>What You Can Do in a Subclass</td>
<td>196</td>
</tr>
<tr>
<td>Private Members in a Superclass</td>
<td>196</td>
</tr>
<tr>
<td>Casting Objects</td>
<td>197</td>
</tr>
<tr>
<td>Multiple Inheritance of State, Implementation, and Type</td>
<td>198</td>
</tr>
<tr>
<td>Overriding and Hiding Methods</td>
<td>199</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>203</td>
</tr>
<tr>
<td>Hiding Fields</td>
<td>206</td>
</tr>
<tr>
<td>Using the Keyword super</td>
<td>206</td>
</tr>
<tr>
<td>Object as a Superclass</td>
<td>208</td>
</tr>
<tr>
<td>Writing Final Classes and Methods</td>
<td>212</td>
</tr>
<tr>
<td>Abstract Methods and Classes</td>
<td>212</td>
</tr>
</tbody>
</table>
Chapter 7  Generics  219
   Why Use Generics?  220
   Generic Types  220
      A Simple Box Class  220
      A Generic Version of the Box Class  221
      Type Parameter Naming Conventions  221
      Invoking and Instantiating a Generic Type  222
      The Diamond  223
      Multiple Type Parameters  223
      Parameterized Types  224
      Raw Types  224
   Generic Methods  226
   Bounded Type Parameters  227
      Multiple Bounds  228
      Generic Methods and Bounded Type Parameters  229
   Generics, Inheritance, and Subtypes  229
      Generic Classes and Subtyping  230
   Type Inference  232
      Type Inference and Generic Methods  232
      Type Inference and Instantiation of Generic Classes  233
      Type Inference and Generic Constructors of Generic and Nongeneric Classes  234
      Target Types  235
   Wildcards  236
      Upper-Bounded Wildcards  236
      Unbounded Wildcards  237
      Lower-Bounded Wildcards  238
      Wildcards and Subtyping  239
      Wildcard Capture and Helper Methods  240
      Guidelines for Wildcard Use  243
   Type Erasure  244
      Erasure of Generic Types  245
      Erasure of Generic Methods  246
      Effects of Type Erasure and Bridge Methods  247
      Nonreifiable Types and Varargs Methods  249
   Restrictions on Generics  252
      Cannot Instantiate Generic Types with Primitive Types  252
      Cannot Create Instances of Type Parameters  253
Contents

The Path Class 362
File Operations 370
Checking a File or Directory 374
Deleting a File or Directory 375
Copying a File or Directory 376
Moving a File or Directory 377
Managing Metadata (File and File Store Attributes) 378
Reading, Writing, and Creating Files 386
Random Access Files 393
Creating and Reading Directories 395
Links, Symbolic or Otherwise 399
Walking the File Tree 401
Finding Files 407
Watching a Directory for Changes 410
Other Useful Methods 416
Legacy File I/O Code 418
Summary 421
Questions and Exercises: Basic I/O 422
  Questions 422
  Exercises 422
  Answers 422

Chapter 12 Collections 423
Introduction to Collections 424
  What Is a Collections Framework? 424
  Benefits of the Java Collections Framework 425
Interfaces 426
  The Collection Interface 428
  Traversing Collections 429
  Collection Interface Bulk Operations 432
  Collection Interface Array Operations 432
  The Set Interface 433
  The List Interface 438
  The Queue Interface 446
  The Deque Interface 448
  The Map Interface 449
  Object Ordering 458
  The SortedSet Interface 464
  The SortedMap Interface 467
  Summary of Interfaces 469
  Questions and Exercises: Interfaces 470
Synchronization 527
  Thread Interference 527
  Memory Consistency Errors 528
  Synchronized Methods 529
  Intrinsic Locks and Synchronization 531
  Atomic Access 533
Liveness 533
  Deadlock 534
  Starvation and Livelock 535
Guarded Blocks 535
Immutable Objects 539
  A Synchronized Class Example 540
  A Strategy for Defining Immutable Objects 541
High-Level Concurrency Objects 543
  Lock Objects 544
  Executors 546
  Concurrent Collections 552
  Atomic Variables 553
  Concurrent Random Numbers 554
Questions and Exercises: Concurrency 555
  Question 555
  Exercises 555
  Answers 556

Chapter 14 Regular Expressions 557
Introduction 558
  What Are Regular Expressions? 558
  How Are Regular Expressions Represented in This Package? 558
Test Harness 559
String Literals 560
  Metacharacters 561
Character Classes 562
  Simple Classes 562
Predefined Character Classes 566
Quantifiers 568
  Zero-Length Matches 569
  Capturing Groups and Character Classes with Quantifiers 572
  Differences among Greedy, Reluctant, and Possessive Quantifiers 573
Capturing Groups 574
  Numbering 574
  Backreferences 575
Contents

Boundary Matchers 576
Methods of the Pattern Class 578
  Creating a Pattern with Flags 578
  Embedded Flag Expressions 580
  Using the matches(String,CharSequence) Method 580
  Using the split(String) Method 581
  Other Utility Methods 582
  Pattern Method Equivalents in java.lang.String 582
Methods of the Matcher Class 583
  Index Methods 583
  Study Methods 584
  Replacement Methods 584
  Using the start and end Methods 585
  Using the matches and lookingAt Methods 586
  Using replaceFirst(String) and replaceAll(String) 587
  Using appendReplacement(StringBuffer,String) and appendTail(StringBuffer) 588
  Matcher Method Equivalents in java.lang.String 589
Methods of the PatternSyntaxException Class 589
Unicode Support 591
  Matching a Specific Code Point 591
  Unicode Character Properties 591
Questions and Exercises: Regular Expressions 592
  Questions 592
  Exercise 593
  Answers 593

Chapter 15 The Platform Environment 595
Configuration Utilities 595
  Properties 596
  Command-Line Arguments 600
  Environment Variables 601
  Other Configuration Utilities 602
System Utilities 603
  Command-Line I/O Objects 603
  System Properties 604
  The Security Manager 607
  Miscellaneous Methods in System 608
PATH and CLASSPATH 609
  Update the PATH Environment Variable (Microsoft Windows) 609
  Update the PATH Variable (Solaris, Linux, and OS X) 611
  Checking the CLASSPATH Variable (All Platforms) 612
Questions and Exercises: The Platform Environment 613
  Question 613
  Exercise 614
  Answers 614

Chapter 16  Packaging Programs in JAR Files 615
  Using JAR Files: The Basics 616
    Creating a JAR File 616
    Viewing the Contents of a JAR File 620
    Extracting the Contents of a JAR File 622
    Updating a JAR File 623
    Running JAR-Packaged Software 625
  Working with Manifest Files: The Basics 627
    Understanding the Default Manifest 627
    Modifying a Manifest File 628
    Setting an Application’s Entry Point 629
    Adding Classes to the JAR File’s Class Path 630
    Setting Package Version Information 631
    Sealing Packages within a JAR File 633
    Enhancing Security with Manifest Attributes 634
  Signing and Verifying JAR Files 635
    Understanding Signing and Verification 636
    Signing JAR Files 639
    Verifying Signed JAR Files 641
  Using JAR-Related APIs 642
    An Example: The JarRunner Application 643
    The JarClassLoader Class 643
    The JarRunner Class 646
  Questions and Exercises: Packaging Programs in JAR Files 648
    Questions 648
    Answers 648

Chapter 17  Java Web Start 649
  Additional References 650
  Developing a Java Web Start Application 650
    Creating the Top JPanel Class 651
    Creating the Application 652
    Benefits of Separating Core Functionality from the Final Deployment Mechanism 652
    Retrieving Resources 653
  Deploying a Java Web Start Application 653
    Setting Up a Web Server 656
Displaying a Customized Loading Progress Indicator 656
Developing a Customized Loading Progress Indicator 657
Specifying a Customized Loading Progress Indicator for a Java Web Start Application 659
Running a Java Web Start Application 660
Running a Java Web Start Application from a Browser 660
Running a Java Web Start Application from the Java Cache Viewer 660
Running a Java Web Start Application from the Desktop 661
Java Web Start and Security 661
Dynamic Downloading of HTTPS Certificates 662
Common Java Web Start Problems 662
“My Browser Shows the Java Network Launch Protocol (JNLP) File for My Application as Plain Text” 663
“When I Try to Launch My JNLP File, I Get the Following Error” 663
Questions and Exercises: Java Web Start 663
Questions 663
Exercises 664
Answers 664
Chapter 18 Applets 665
Getting Started with Applets 666
Defining an Applet Subclass 666
Methods for Milestones 667
Life Cycle of an Applet 668
Applet’s Execution Environment 670
Developing an Applet 670
Deploying an Applet 673
Doing More with Applets 677
Finding and Loading Data Files 677
Defining and Using Applet Parameters 678
Displaying Short Status Strings 681
Displaying Documents in the Browser 682
Invoking JavaScript Code from an Applet 683
Invoking Applet Methods from JavaScript Code 686
Handling Initialization Status with Event Handlers 689
Manipulating DOM of Applet’s Web Page 691
Displaying a Customized Loading Progress Indicator 693
Writing Diagnostics to Standard Output and Error Streams 698
Developing Draggable Applets 698
Communicating with Other Applets 701
Working with a Server-Side Application 703
What Applets Can and Cannot Do 705
Solving Common Applet Problems 707
 “My Applet Does Not Display” 707
 “The Java Console Log Displays java.lang.ClassNotFoundException” 708
 “I Was Able to Build the Code Once, but Now the Build Fails Even Though There Are No Compilation Errors” 708
 “When I Try to Load a Web Page That Has an Applet, My Browser Redirects Me to www.java.com without Any Warning” 708
 “I Fixed Some Bugs and Rebuilt My Applet’s Source Code. When I Reload the Applet’s Web Page, My Fixes Are Not Showing Up” 708
Questions and Exercises: Applets 708
 Questions 708
 Exercises 709
 Answers 709

Chapter 19 Doing More with Java Rich Internet Applications 711
Setting Trusted Arguments and Secure Properties 711
 System Properties 713
 JNLP API 714
 Accessing the Client Using the JNLP API 715
Cookies 719
 Types of Cookies 719
 Cookie Support in RIAs 719
 Accessing Cookies 720
 Customizing the Loading Experience 722
Security in Rich Internet Applications 722
 Guidelines for Securing RIAs 724
 Follow Secure Coding Guidelines 724
 Test with the Latest Version of the JRE 724
 Include Manifest Attributes 725
 Use a Signed JNLP File 725
 Sign and Time Stamp JAR Files 725
 Use the HTTPS Protocol 726
 Avoid Local RIAs 726
 Questions and Exercises: Doing More with Rich Internet Applications 726
 Questions 726
Exercise 726
Answers 727

Chapter 20 Deployment in Depth
User Acceptance of RIAs 729
Deployment Toolkit 731
  Location of Deployment Toolkit Script 731
  Deploying an Applet 732
  Deploying a Java Web Start Application 735
  Checking the Client JRE Software Version 738
Java Network Launch Protocol 739
  Structure of the JNLP File 739
Deployment Best Practices 748
  Reducing the Download Time 748
  Avoiding Unnecessary Update Checks 749
  Ensuring the Presence of the JRE Software 751
Questions and Exercises: Deployment in Depth 753
  Questions 753
  Exercise 753
  Answers 753

Chapter 21 Date-Time
Date-Time Overview 756
Date-Time Design Principles 756
  Clear 756
  Fluent 757
  Immutable 757
  Extensible 757
The Date-Time Packages 757
Method Naming Conventions 758
Standard Calendar 759
  Overview 759
DayOfWeek and Month Enums 760
  DayOfWeek 760
  Month 762
Date Classes 762
  LocalDate 763
  YearMonth 763
  MonthDay 764
  Year 764
Date and Time Classes 764
  LocalDateTime 764
This page intentionally left blank
Preface

Since the acquisition of Sun Microsystems by Oracle Corporation in early 2010, it has been an exciting time for the Java language. As evidenced by the activities of the Java Community Process program, the Java language continues to evolve. The publication of this sixth edition of The Java® Tutorial reflects version 8 of the Java Platform Standard Edition (Java SE) and references the Application Programming Interface (API) of that release.

This edition introduces new features added to the platform since the publication of the fifth edition (under release 7):

- Lambda expressions enable you to treat functionality as a method argument or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly. See the new section in Chapter 4, “Lambda Expressions.”
- Type annotations can be used in conjunction with pluggable type systems for improved type checking, and repeating annotations enable the application of the same annotation to a declaration or type use. See the new sections in Chapter 5, “Type Annotations and Pluggable Type Systems” and “Repeating Annotations.”
- Default methods are methods in an interface that have an implementation. They enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces. See the new section in Chapter 6, “Default Methods.”
Aggregate operations enable you to perform functional-style operations on streams of elements—in particular, bulk operations on collections such as sequential or parallel map-reduce transformations. See the new section in Chapter 12, “Aggregate Operations.”

Improvements have been added that focus on limiting attackers from using malicious applets and rich Internet applications (RIAs). See the following new and updated sections:

- Chapter 16, “Packaging Programs in JAR Files”
- Chapter 20, “Deployment Best Practices”

Date-Time APIs enable you to represent dates and times and manipulate date and time values. They support the International Organization for Standardization (ISO) calendar system as well as other commonly used global calendars. See the new Chapter 21.

If you plan to take one of the Java SE 8 certification exams, this book can help. The appendix, “Preparation for Java Programming Language Certification,” lists the three exams that are available, detailing the items covered by each exam, cross-referenced to places in the book where you can find more information about each topic. Note that this is one source, among others, that you will want to use to prepare for your exam. Check the online tutorial for the latest certification objectives and cross-references to sections of the tutorial.

All of the material has been thoroughly reviewed by members of Oracle Java engineering to ensure that the information is accurate and up to date. This book is based on the online tutorial hosted on Oracle Corporation’s web site at the following URL:

http://docs.oracle.com/javase/tutorial/

The information in this book, often referred to as “the core tutorial,” is required by most beginning to intermediate programmers. Once you have mastered this material, you can explore the rest of the Java platform documentation on the web site. If you are interested in developing sophisticated RIAs, check out JavaFX, the Java graphical user interface (GUI) toolkit, which comes with the Java SE Development Kit (JDK). To learn more, see Chapter 22, “Introduction to JavaFX.”

As always, our goal is to create an easy-to-read, practical programmers’ guide to help you learn how to use the rich environment provided by Java to build applications, applets, and components. Go forth and program!
Who Should Read This Book?

This book is geared toward both novice and experienced programmers:

- **New programmers** can benefit most from reading the book from beginning to end, including the step-by-step instructions for compiling and running your first program in Chapter 1, “Getting Started.”
- **Programmers experienced with procedural languages** such as C may want to start with the material on object-oriented concepts and features of the Java programming language.
- **Experienced programmers** may want to jump feet first into the more advanced topics, such as generics, concurrency, or deployment.

This book contains information to address the learning needs of programmers with various levels of experience.

How to Use This Book

This book is designed so you can read it straight through or skip around from topic to topic. The information is presented in a logical order, and forward references are avoided wherever possible.

The examples in this book are compiled against the JDK 8 release. **You need to download this release (or later) in order to compile and run most examples.**

Some material referenced in this book is available online—for example, the downloadable examples, the solutions to the questions and exercises, the JDK 8 guides, and the API specification.

You will see footnotes like the following:

8/docs/api/java/lang/Class.html

and

tutorial/java/generics/examples/BoxDemo.java

The Java documentation home on the Oracle web site is at the following location:

http://docs.oracle.com/javase/

To locate the footnoted files online, prepend the URL for the Java documentation home:

http://docs.oracle.com/javase/8/docs/api/java/lang/Class.html
http://docs.oracle.com/javase/tutorial/java/generics/examples/BoxDemo.java

The Java Tutorials are also available in two eBook formats:

- mobi eBook files for Kindle
- ePub eBook files for iPad, Nook, and other devices that support the ePub format

Each eBook contains a single trail that is equivalent to several related chapters in this book. You can download the eBooks via the link “In Book Form” on the home page for the Java Tutorials:

http://docs.oracle.com/javase/tutorial/index.html

We welcome feedback on this edition. To contact us, please see the tutorial feedback page:

http://docs.oracle.com/javase/feedback.html

Acknowledgments

This book would not be what it is without the Oracle Java engineering team who tirelessly reviews the technical content of our writing. For this edition of the book, we especially want to thank Alan Bateman, Alex Buckley, Stephen Colebourne, Joe Darcy, Jeff Dinkins, Mike Duigou, Brian Goetz, Andy Herrick, Stuart Marks, Thomas Ng, Roger Riggs, Leif Samuelsson, and Daniel Smith.

Illustrators Jordan Douglas and Dawn Tyler created our professional graphics quickly and efficiently.

Editors Janet Blowney, Deborah Owens, and Susan Shepard provided careful and thorough copyedits of our JDK 8 work.

Thanks for the support of our team: Devika Gollapudi, Ram Goyal, and Alexey Zhebel.

Last but not least, thanks for the support of our management: Sowmya Kannan, Sophia Mikulinsky, Alan Sommerer, and Barbara Ramsey.
About the Authors

Raymond Gallardo is a senior technical writer at Oracle Corporation. His previous engagements include college instructor, technical writer for IBM, and bicycle courier. He obtained his BSc in computer science and English from the University of Toronto and MA in creative writing from the City College of New York.

Scott Hommel is a senior technical writer at Oracle Corporation, where he documents Java SE. For the past fifteen years, he has written tutorials, technical articles, and core release documentation for Java SE and related technologies.

Sowmya Kannan wears many hats on the Java SE documentation team, including planning, writing, communicating with developer audiences, and tinkering with production tools. She has more than fifteen years of experience in the design, development, and documentation of the Java platform, Java-based middleware, and web applications.

Joni Gordon is a principal technical writer at Oracle Corporation. She has contributed to the documentation for Java SE and JavaFX. She has been a technical writer for more than fifteen years and has a background in enterprise application development.

Sharon Biocca Zakhour was previously a principal technical writer on staff at Oracle Corporation and formerly at Sun Microsystems. She has contributed to Java SE documentation for more than twelve years, including The Java™ Tutorial, Fourth Edition, and The JFC Swing Tutorial, Second Edition. She graduated from UC Berkeley with a BA in computer science and has worked as a programmer, developer support engineer, and technical writer for thirty years.
This page intentionally left blank
This page intentionally left blank
You've already learned that objects store their state in fields. However, the Java programming language uses the term *variable* as well. The first section of this chapter discusses this relationship, plus variable naming rules and conventions, basic data types (primitive types, character strings, and arrays), default values, and literals.

The second section describes the operators of the Java programming language. It presents the most commonly used operators first and the less commonly used operators last. Each discussion includes code samples that you can compile and run.

Operators may be used for building expressions, which compute values; expressions are the core components of statements, and statements may be grouped into blocks. The third section discusses expressions, statements, and blocks using example code that you've already seen.

The final section describes the control flow statements supported by the Java programming language. It covers the decision-making, looping, and branching statements that enable your programs to conditionally execute particular blocks of code.

Note that each section contains its own questions and exercises to test your understanding.
Variables

As you learned in the previous chapter, an object stores its state in fields:

```java
int cadence = 0;
int speed = 0;
int gear = 1;
```

In Chapter 2, the section “What Is an Object?” introduced you to fields, but you probably have still a few questions, such as, What are the rules and conventions for naming a field? Besides `int`, what other data types are there? Do fields have to be initialized when they are declared? Are fields assigned a default value if they are not explicitly initialized? We’ll explore the answers to such questions in this chapter, but before we do, there are a few technical distinctions you must first become aware of. In the Java programming language, the terms field and variable are both used; this is a common source of confusion among new developers because both often seem to refer to the same thing. The Java programming language defines the following kinds of variables:

- **Instance variables (nonstatic fields).** Technically speaking, objects store their individual states in “nonstatic fields”—that is, fields declared without the static keyword. Nonstatic fields are also known as instance variables because their values are unique to each instance of a class (to each object, in other words); for example, the `currentSpeed` of one bicycle is independent of the `currentSpeed` of another.

- **Class variables (static fields).** A class variable is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. For example, a field defining the number of gears for a particular kind of bicycle could be marked as static since conceptually the same number of gears will apply to all instances. The code `static int numGears = 6;` would create such a static field. Additionally, the keyword `final` could be added to indicate that the number of gears will never change.

- **Local variables.** Similar to how an object stores its state in fields, a method will often store its temporary state in local variables. The syntax for declaring a local variable is similar to declaring a field (e.g., `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared— which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
- **Parameters.** You’ve already seen examples of parameters, both in the Bicycle class and in the main method of the “Hello World!” application. Recall that the signature for the main method is `public static void main(String[] args)`. Here, the `args` variable is the parameter to this method. The important thing to remember is that parameters are always classified as *variables*, not *fields*. This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you’ll learn about later in the chapter.

That said, the remainder of the chapters use the following general guidelines when discussing fields and variables. If we are talking about *fields in general* (excluding local variables and parameters), we may simply use the term *fields*. If the discussion applies to *all of the above*, we may simply use the term *variables*. If the context calls for a distinction, we will use specific terms (such as *static field* or *local variables*) as appropriate. You may also occasionally see the term *member* used as well. A type’s fields, methods, and nested types are collectively called its *members*.

**Naming**

Every programming language has its own set of rules and conventions for the kinds of names that you’re allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarized as follows:

- Variable names are case sensitive. A variable’s name can be any legal identifier—an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign ($), or the underscore character (_). The convention, however, is to always begin your variable names with a letter, not $ or _. Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where autogenerated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it’s technically legal to begin your variable’s name with _, this practice is discouraged. White space is not permitted.

- Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases, it will also make your code self-documenting; fields named cadence, speed, and gear, for example, are much more intuitive than abbreviated versions, such as
s, c, and g. Also keep in mind that the name you choose must not be a keyword or reserved word.

- If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word. The names gearRatio and currentGear are prime examples of this convention. If your variable stores a constant value, such as static final int NUM_GEARS = 6, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

### Primitive Data Types

The Java programming language is statically typed, which means that all variables must first be declared before they can be used. This involves stating the variable’s type and name, as you’ve already seen:

```java
int gear = 1;
```

Doing so tells your program that a field named `gear` exists, holds numerical data, and has an initial value of 1. A variable’s data type determines the values it may contain plus the operations that may be performed on it. In addition to `int`, the Java programming language supports seven other primitive data types. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are as follows:

1. The `byte` data type is an 8-bit signed two’s complement integer. It has a minimum value of –128 and a maximum value of 127 (inclusive). The `byte` data type can be useful for saving memory in large arrays, where the memory savings actually matters. It can also be used in place of `int` where its limits help clarify your code; the fact that a variable’s range is limited can serve as a form of documentation.

2. The `short` data type is a 16-bit signed two’s complement integer. It has a minimum value of –32,768 and a maximum value of 32,767 (inclusive). As with `byte`, the same guidelines apply: you can use a `short` to save memory in large arrays in situations where the memory savings actually matter.

3. By default, the `int` data type is a 32-bit signed two’s complement integer, which has a minimum value of –2\(^{31}\) and a maximum value of \(2^{31} – 1\). In Java SE 8 and later, you can use the `int` data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of \(2^{32} – 1\). The `Integer` class also supports unsigned 32-bit integers. Static methods like
compareUnsigned and divideUnsigned have been added to the Integer class to support arithmetic operations for unsigned integers.  

4. The long data type is a 64-bit two's complement integer. The signed long has a minimum value of \(-2^{63}\) and a maximum value of \(2^{63} - 1\). In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of \(2^{64} - 1\). Use this data type when you need a range of values wider than those provided by the int data type. The Long class also contains methods like compareUnsigned and divideUnsigned to support arithmetic operations for unsigned long values.  

5. The float data type is a single-precision 32-bit IEEE 754 floating-point value. Its range of values is beyond the scope of this discussion but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. As with the recommendations for the byte and short data types, use a float (instead of double) value if you need to save memory in large arrays of floating-point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead. Chapter 9 covers BigDecimal and other useful classes provided by the Java platform.  

6. The double data type is a double-precision 64-bit IEEE 754 floating-point value. Its range of values is beyond the scope of this discussion but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned previously, this data type should never be used for precise values, such as currency.  

7. The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its “size” isn’t something that’s precisely defined.  

8. The char data type is a single 16-bit Unicode character. It has a minimum value of \('\u0000'\) (or 0) and a maximum value of \('\uffff'\) (or 65,535 inclusive).  

In addition to the eight primitive data types, the Java programming language also provides special support for character strings through the java.lang.String
class. Enclosing your character string within double quotes will automatically create a new String object—for example, `String s = "this is a string";`. String objects are immutable, which means that, once created, their values cannot be changed. The String class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such. You'll learn more about the String class in Chapter 9.

**Default Values**

It's not always necessary to assign a value when a field is declared. Fields that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or `null`, depending on the data type. Relying on such default values, however, is generally considered bad programming style. Table 3.1 summarizes the default values for the above data types.

Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

**Literals**

You may have noticed that the `new` keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A literal is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown here, it's possible to assign a literal to a variable of a primitive type:

```java
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

**Integer Literals**

An integer literal is of type `long` if it ends with the letter `L` or `l`; otherwise, it is of type `int`. It is recommended that you use the uppercase letter `L` because the lowercase letter `l` is hard to distinguish from the digit `1`.

Values of the integral types `byte`, `short`, `int`, and `long` can be created from `int` literals. Values of type `long` that exceed the range of `int` can be created from `long` literals. Integer literals can be expressed by these number systems:

---

Variables

- **Decimal.** Base 10, whose digits consist of the numbers 0 through 9 (This is the number system you use every day.)
- **Hexadecimal.** Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F
- **Binary.** Base 2, whose digits consists of the numbers 0 and 1

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix 0x indicates hexadecimal and 0b indicates binary:

```java
// The number 26, in decimal
int decVal = 26;
// The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

**Floating-Point Literals**

A floating-point literal is of type `float` if it ends with the letter F or f; otherwise, its type is `double` and it can optionally end with the letter D or d. The floating-point types (`float` and `double`) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal), and D or d (64-bit double literal, which is the default and by convention is omitted).

```java
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;
```

---

**Table 3.1** Default Values for Data Types

<table>
<thead>
<tr>
<th>Data type</th>
<th>Default value (for fields)</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>0L</td>
</tr>
<tr>
<td>float</td>
<td>0.0f</td>
</tr>
<tr>
<td>double</td>
<td>0.0d</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>char</td>
<td>'\u0000'</td>
</tr>
<tr>
<td>String (or any object)</td>
<td>null</td>
</tr>
</tbody>
</table>
Character and String Literals

Literals of types char and String may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a Unicode escape, such as '\u0108' (for a capital C with circumflex, Ĉ) or "S\u00ED Se\u00F1or" (for Sí Señor in Spanish). Always use 'single quotes' for char literals and "double quotes" for String literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in char or String literals.

The Java programming language also supports a few special escape sequences for char and String literals: \b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), " (double quote), ’ (single quote), and \ (backslash).

There’s also a special null literal that can be used as a value for any reference type. You may assign null to any variable except variables of primitive types. There’s little you can do with a null value beyond testing for its presence. Therefore, null is often used in programs as a marker to indicate that some object is unavailable.

Finally, there’s also a special kind of literal called a class literal, formed by taking a type name and appending .class (e.g., String.class). This refers to the object (of type Class) that represents the type itself.

Using Underline Characters in Numeric Literals

Any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma or a space as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```java
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:
Variables

- At the beginning or end of a number
- Adjacent to a decimal point in a floating-point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

The following examples demonstrate valid and invalid underscore placements (which are bold) in numeric literals:

```java
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;
// OK (decimal literal)
int x3 = 5_______2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;
// OK (hexadecimal literal)
int x6 = 0x52_;
// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

Arrays

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. You have seen an example of arrays already, in the main method of the “Hello World!” application. This section discusses arrays in greater detail.

Each item in an array is called an element, and each element is accessed by its numerical index. As shown in the preceding illustration, numbering begins with 0. The ninth element, for example, would therefore be accessed at index 8.
The following program, `ArrayDemo`, creates an array of integers, puts some values in the array, and prints each value to standard output:

```java
class ArrayDemo {
    public static void main(String[] args) {
        // declares an array of integers
        int[] anArray;

        // allocates memory for 10 integers
        anArray = new int[10];

        // initialize first element
        anArray[0] = 100;
        // initialize second element
        anArray[1] = 200;
        // and so forth
        anArray[2] = 300;
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Element at index 0: " + anArray[0]);
        System.out.println("Element at index 1: " + anArray[1]);
        System.out.println("Element at index 2: " + anArray[2]);
        System.out.println("Element at index 3: " + anArray[3]);
        System.out.println("Element at index 4: " + anArray[4]);
        System.out.println("Element at index 5: " + anArray[5]);
        System.out.println("Element at index 6: " + anArray[6]);
        System.out.println("Element at index 7: " + anArray[7]);
        System.out.println("Element at index 8: " + anArray[8]);
        System.out.println("Element at index 9: " + anArray[9]);
    }
}
```
Here is the output from this program:

Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000

In a real-world programming situation, you would probably use one of the supported looping constructs to iterate through each element of the array, rather than write each line individually as in the preceding example. However, the example clearly illustrates the array syntax. You will learn about the various looping constructs (for, while, and do-while) later in the “Control Flow” section.

Declaring a Variable to Refer to an Array

The preceding program declares an array (named anArray) with the following code:

```java
// declares an array of integers
int[] anArray;
```

Like declarations for variables of other types, an array declaration has two components: the array’s type and the array’s name. An array’s type is written as `type[]`, where `type` is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array’s name can be anything you want, provided that it follows the rules and conventions as previously discussed in the “Naming” section. As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type. Similarly, you can declare arrays of other types:

```java
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

You can also place the brackets after the array’s name:

```java
// this form is discouraged
float anArrayOfFloats[];
```
However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

**Creating, Initializing, and Accessing an Array**

One way to create an array is with the `new` operator. The next statement in the `ArrayDemo` program allocates an array with enough memory for ten integer elements and assigns the array to the `anArray` variable:

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

If this statement is missing, then the compiler prints an error like the following and compilation fails:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

The next few lines assign values to each element of the array:

```java
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // and so forth
```

Each array element is accessed by its numerical index:

```java
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```java
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

Here the length of the array is determined by the number of values provided between braces and separated by commas.

You can also declare an array of arrays (also known as a *multidimensional array*) by using two or more sets of brackets, such as `String[][]` names. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following `MultiDimArrayDemo` program:
The following code prints the array’s size to standard output:

```java
System.out.println(anArray.length);
```

### Copying Arrays

The `System` class has an `arraycopy()` method that you can use to efficiently copy data from one array into another:

```java
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

The two `Object` arguments specify the array to copy from and the array to copy to. The three `int` arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, `ArrayCopyDemo`, declares an array of `char` elements, spelling the word *decaffeinated*. It uses the `System.arraycopy()` method to copy a subsequence of array components into a second array:

```java
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```
Here is output from this program:

caffein

### Array Manipulations

Arrays are a powerful and useful concept in programming. Java SE provides methods to perform some of the most common manipulations related to arrays. For instance, the ArrayCopyDemo example uses the arraycopy() method of the System class instead of manually iterating through the elements of the source array and placing each one into the destination array. This is performed behind the scenes, enabling the developer to use just one line of code to call the method.

For your convenience, Java SE provides several methods for performing array manipulations (common tasks such as copying, sorting, and searching arrays) in the java.util.Arrays class. For instance, the previous example can be modified to use the copyOfRange() method of the java.util.Arrays class, as you can see in the ArrayCopyOfDemo example. The difference is that using the copyOfRange() method does not require you to create the destination array before calling the method because the destination array is returned by the method:

```java
class ArrayCopyOfDemo {
    public static void main(String[] args) {
        char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd'};
        char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);
        System.out.println(new String(copyTo));
    }
}
```

As you can see, the output from this program is the same (caffein), although it requires fewer lines of code.

Some other useful operations provided by methods in the java.util.Arrays class are as follows:

- Search an array for a specific value to get the index at which it is placed (the binarySearch() method).
- Compare two arrays to determine if they are equal or not (the equals() method).
- Fill an array to place a specific value at each index (the fill() method).

---

7. [8/docs/api/java/util/Arrays.html](http://8/docs/api/java/util/Arrays.html)
- Sort an array into ascending order. This can be done either sequentially, using the `sort()` method, or concurrently, using the `parallelSort()` method introduced in Java SE 8. Parallel sorting of large arrays on multiprocessor systems is faster than sequential array sorting.

**Summary of Variables**

The Java programming language uses both *fields* and *variables* as part of its terminology. Instance variables (nonstatic fields) are unique to each instance of a class. Class variables (static fields) are fields declared with the `static` modifier; there is exactly one copy of a class variable, regardless of how many times the class has been instantiated. Local variables store temporary state inside a method. Parameters are variables that provide extra information to a method; both local variables and parameters are always classified as *variables* (not *fields*). When naming your fields or variables, there are rules and conventions that you should (or must) follow.

The eight primitive data types are `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`. The `java.lang.String` class represents character strings. The compiler will assign a reasonable default value for fields of these types; for local variables, a default value is never assigned. A literal is the source code representation of a fixed value. An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

**Questions and Exercises: Variables**

**Questions**

1. The term *instance variable* is another name for ___.
2. The term *class variable* is another name for ___.
3. A local variable stores temporary state; it is declared inside a ___.
4. A variable declared within the opening and closing parenthesis of a method signature is called a ___.
5. What are the eight primitive data types supported by the Java programming language?
6. Character strings are represented by the class ___.
7. An ___ is a container object that holds a fixed number of values of a single type.

---

8. 8/docs/api/java/lang/String.html
Exercises

1. Create a small program that defines some fields. Try creating some illegal field names and see what kind of error the compiler produces. Use the naming rules and conventions as a guide.

2. In the program you created in Exercise 1, try leaving the fields uninitialized and print out their values. Try the same with a local variable and see what kind of compiler errors you can produce. Becoming familiar with common compiler errors will make it easier to recognize bugs in your code.

Answers

You can find answers to these questions and exercises at http://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/answers_variables.html.

Operators

Now that you’ve learned how to declare and initialize variables, you probably want to know how to do something with them. Learning the operators of the Java programming language is a good place to start. Operators are special symbols that perform specific operations on one, two, or three operands and then return a result.

As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in Table 3.2 are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first.

All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

In general-purpose programming, certain operators tend to appear more frequently than others; for example, the assignment operator (\(=\)) is far more common than the unsigned right shift operator (\(>>\)). With that in mind, the following discussion focuses first on the operators that you’re most likely to use on a regular basis and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you’ve just learned.
Assignment, Arithmetic, and Unary Operators

The Simple Assignment Operator

One of the most common operators that you’ll encounter is the simple assignment operator, =. You saw this operator in the Bicycle class; it assigns the value on its right to the operand on its left:

```java
int cadence = 0;
int speed = 0;
int gear = 1;
```

This operator can also be used on objects to assign object references, as discussed in Chapter 4, “Creating Objects.”

The Arithmetic Operators

The Java programming language provides operators that perform addition, subtraction, multiplication, and division. There’s a good chance you’ll recognize them by their counterparts in basic mathematics. The only symbol that might look new to you is %, which divides one operand by another and returns the remainder as its result.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postfix</td>
<td>expr ++ expr --</td>
</tr>
<tr>
<td>unary</td>
<td>++expr --expr +expr -expr ~ !</td>
</tr>
<tr>
<td>multiplicative</td>
<td>* / %</td>
</tr>
<tr>
<td>additive</td>
<td>+ -</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</td>
</tr>
<tr>
<td>relational</td>
<td>&lt; &gt; &lt;= &gt;= instanceof</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>bitwise AND</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitwise exclusive OR</td>
<td>^</td>
</tr>
<tr>
<td>bitwise inclusive OR</td>
<td></td>
</tr>
<tr>
<td>logical AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>logical OR</td>
<td></td>
</tr>
<tr>
<td>ternary</td>
<td>? :</td>
</tr>
<tr>
<td>assignment</td>
<td>= += -= *= /= %= &amp;= ^=</td>
</tr>
</tbody>
</table>
The following program, `ArithmeticDemo`, tests the arithmetic operators:

```java
class ArithmeticDemo {
    public static void main (String[] args) {
        int result = 1 + 2;
        // result is now 3
        System.out.println("1 + 2 = " + result);
        int original_result = result;

        result = result - 1;
        // result is now 2
        System.out.println(original_result + " - 1 = " + result);
        original_result = result;

        result = result * 2;
        // result is now 4
        System.out.println(original_result + " * 2 = " + result);
        original_result = result;

        result = result / 2;
        // result is now 2
        System.out.println(original_result + " / 2 = " + result);
        original_result = result;

        result = result + 8;
        // result is now 10
        System.out.println(original_result + " + 8 = " + result);
        original_result = result;

        result = result % 7;
        // result is now 3
        System.out.println(original_result + " % 7 = " + result);
    }
}
```

This program prints the following:

```
1 + 2 = 3
3 - 1 = 2
2 * 2 = 4
4 / 2 = 2
2 + 8 = 10
10 % 7 = 3
```

### Table 3.3 Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Additive operator (also used for String concatenation)</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction operator</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication operator</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division operator</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Remainder operator</td>
</tr>
</tbody>
</table>

This program tests the arithmetic operators with different values and demonstrates how they manipulate the `result` variable.
You can also combine the arithmetic operators with the simple assignment operator to create compound assignments. For example, x+=1; and x=x+1; both increment the value of x by 1.

The + operator can also be used for concatenating (joining) two strings together, as shown in the following ConcatDemo program:

```java
class ConcatDemo {
    public static void main(String[] args){
        String firstString = "This is";
        String secondString = " a concatenated string.";
        String thirdString = firstString + secondString;
        System.out.println(thirdString);
    }
}
```

By the end of this program, the variable thirdString contains "This is a concatenated string.", which gets printed to standard output.

### The Unary Operators

The unary operators require only one operand; they perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

The following program, UnaryDemo, tests the unary operators:

```java
class UnaryDemo {
    public static void main(String[] args) {
        int result = +1;
        // result is now 1
        System.out.println(result);

        result--;
        // result is now 0
        System.out.println(result);

        result++;
        // result is now 1
        System.out.println(result);

        result = -result;
        // result is now -1
        System.out.println(result);

        boolean success = false;
        // false
        System.out.println(success);

        System.out.println(!success);
    }
}
```
The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code `result++;` and `++result;` will both end in `result` being incremented by one. The only difference is that the prefix version (`++result`) evaluates to the incremented value, whereas the postfix version (`result++`) evaluates to the original value. If you are just performing a simple increment/decrement operation, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one you choose may make a significant difference.

The following program, `PrePostDemo`, illustrates the prefix/postfix unary increment operator:

```java
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++; // prints 4
        System.out.println(i);
        ++i; // prints 5
        System.out.println(i);
        // prints 6
        System.out.println(++i);
        // prints 6
        System.out.println(i++);
        // prints 7
        System.out.println(i);
    }
}
```

### Equality, Relational, and Conditional Operators

#### The Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use `==`, not `=`, when testing if two primitive values are equal:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Unary plus operator; indicates positive value (numbers are positive without this, however)</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus operator; negates an expression</td>
</tr>
<tr>
<td>++</td>
<td>Increment operator; increments a value by 1</td>
</tr>
<tr>
<td>--</td>
<td>Decrement operator; decrements a value by 1</td>
</tr>
<tr>
<td>!</td>
<td>Logical complement operator; inverts the value of a boolean</td>
</tr>
</tbody>
</table>
== equal to
!= not equal to
> greater than
>= greater than or equal to
< less than
<= less than or equal to

The following program, ComparisonDemo, tests the comparison operators:

class ComparisonDemo {
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        if (value1 == value2)
            System.out.println("value1 == value2");
        if (value1 != value2)
            System.out.println("value1 != value2");
        if (value1 > value2)
            System.out.println("value1 > value2");
        if (value1 < value2)
            System.out.println("value1 < value2");
        if (value1 <= value2)
            System.out.println("value1 <= value2");
    }
}

Here is the output:

value1 != value2
value1 < value2
value1 <= value2

The Conditional Operators
The && and || operators perform Conditional-AND and Conditional-OR operations on two boolean expressions. These operators exhibit short-circuiting behavior, which means that the second operand is evaluated only if needed:

&& Conditional-AND
|| Conditional-OR

The following program, ConditionalDemo1, tests these operators:

class ConditionalDemo1 {
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        if ((value1 == 1) && (value2 == 2))
            System.out.println("value1 is 1 AND value2 is 2");
        if ((value1 == 1) || (value2 == 1))
Another conditional operator is `?:`, which can be thought of as shorthand for an if-then-else statement (discussed in the “Control Flow Statements” section of this chapter). This operator is also known as the **ternary operator** because it uses three operands. In the following example, this operator should be read as follows: “If \texttt{someCondition} is true, assign the value of \texttt{value1} to \texttt{result}. Otherwise, assign the value of \texttt{value2} to \texttt{result}.”

The following program, **ConditionalDemo2**, tests the `?:` operator:

```java
class ConditionalDemo2 {
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        int result;
        boolean someCondition = true;
        result = someCondition ? value1 : value2;
        System.out.println(result);
    }
}
```

Because \texttt{someCondition} is true, this program prints 1 to the screen. Use the `?:` operator instead of an if-then-else statement if it makes your code more readable (e.g., when the expressions are compact and without side effects, such as in assignments).

### The Type Comparison Operator `instanceof`

The `instanceof` operator compares an object to a specified type. You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

The following program, **InstanceofDemo**, defines a parent class (named \texttt{Parent}), a simple interface (named \texttt{MyInterface}), and a child class (named \texttt{Child}) that inherits from the parent and implements the interface.

```java
class InstanceofDemo {
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        Parent obj2 = new Child();

        System.out.println("obj1 instanceof Parent: "
        + (obj1 instanceof Parent));
        System.out.println("obj1 instanceof Child: "
        + (obj1 instanceof Child));
    }
}
```
System.out.println("obj1 instanceof MyInterface: "+ (obj1 instanceof MyInterface));  
System.out.println("obj2 instanceof Parent: "+ (obj2 instanceof Parent));  
System.out.println("obj2 instanceof Child: "+ (obj2 instanceof Child));  
System.out.println("obj2 instanceof MyInterface: "+ (obj2 instanceof MyInterface));  
}

class Parent {}
class Child extends Parent implements MyInterface {}
interface MyInterface {}

Here is the output:

obj1 instanceof Parent: true  
obj1 instanceof Child: false  
obj1 instanceof MyInterface: false  
obj2 instanceof Parent: true  
obj2 instanceof Child: true  
obj2 instanceof MyInterface: true

When using the instanceof operator, keep in mind that null is not an instance of anything.

**Bitwise and Bit Shift Operators**

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used. Therefore their coverage is brief; the intent is to simply make you aware that these operators exist.

The unary bitwise complement operator (~) inverts a bit pattern; it can be applied to any of the integral types, making every 0 a 1 and every 1 a 0. For example, a byte contains 8 bits; applying this operator to a value whose bit pattern is 00000000 would change its pattern to 11111111.

The signed left shift operator (<<) shifts a bit pattern to the left, and the signed right shift operator (>>) shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift is given by the right-hand operand. The unsigned right shift operator (>>>)) shifts a zero into the leftmost position, while the leftmost position after >> depends on sign extension.

The bitwise & operator performs a bitwise AND operation. The bitwise ^ operator performs a bitwise exclusive OR operation. The bitwise | operator performs a bitwise inclusive OR operation.

The following program, BitDemo, uses the bitwise AND operator to print the number 2 to standard output:
class BitDemo {
    public static void main(String[] args) {
        int bitmask = 0x000F;
        int val = 0x2222;
        // prints "2"
        System.out.println(val & bitmask);
    }
}

Summary of Operators

The following quick reference summarizes the operators supported by the Java programming language.

**Simple Assignment Operator**

= Simple assignment operator

**Arithmetic Operators**

+ Additive operator (also used for String concatenation)
- Subtraction operator
* Multiplication operator
/ Division operator
% Remainder operator

**Unary Operators**

+ Unary plus operator; indicates positive value, although numbers can be positive without this
- Unary minus operator; negates an expression
++ Increment operator; increments a value by 1
-- Decrement operator; decrements a value by 1
! Logical complement operator; inverts the value of a boolean

**Equality and Relational Operators**

== Equal to
!= Not equal to
> Greater than
>= Greater than or equal to
< Less than
<= Less than or equal to
Conditional Operators

&&    Conditional AND
||    Conditional OR
?:    Ternary (shorthand for if-then-else statement)

Type Comparison Operator

instanceof    Compares an object to a specified type

Bitwise and Bit Shift Operators

~    Unary bitwise complement
<<    Signed left shift
>>    Signed right shift
>>>    Unsigned right shift
&    Bitwise AND
^    Bitwise exclusive OR
|    Bitwise inclusive OR

Questions and Exercises: Operators

Questions

1. Consider the following code snippet:
   ```java
   arrayOfInts[j] > arrayOfInts[j+1]
   ```
   Which operators does the code contain?

2. Consider the following code snippet.
   ```java
   int i = 10;
   int n = i++%5;
   ```
   a. What are the values of i and n after the code is executed?
   b. What are the final values of i and n if instead of using the postfix increment operator (i++), you use the prefix version (++i)?

3. To invert the value of a boolean, which operator would you use?

4. Which operator is used to compare two values, = or == ?

5. Explain the following code sample: `result = someCondition ? value1 : value2;`

Exercises

1. Change the following program to use compound assignments:
Chapter 3  ■  Language Basics

```java
class ArithmeticDemo {
    public static void main (String[] args){
        int result = 1 + 2; // result is now 3
        System.out.println(result);
        result = result - 1; // result is now 2
        System.out.println(result);
        result = result * 2; // result is now 4
        System.out.println(result);
        result = result / 2; // result is now 2
        System.out.println(result);
        result = result + 8; // result is now 10
        result = result % 7; // result is now 3
        System.out.println(result);
    }
}
```

2. In the following program, explain why the value 6 is printed twice in a row:

```java
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        System.out.println(i); // "4"
        ++i;
        System.out.println(i); // "5"
        System.out.println(++i); // "6"
        System.out.println(i++); // "6"
        System.out.println(i); // "7"
    }
}
```

Answers

You can find answers to these questions and exercises at http://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/answers_operators.html.

Expressions, Statements, and Blocks

Now that you understand variables and operators, it’s time to learn about expressions, statements, and blocks. Operators may be used in building expressions, which compute values. Expressions are the core components of statements, and statements may be grouped into blocks.

Expressions

An expression is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates
to a single value. You’ve already seen examples of expressions, illustrated in bold as follows:

```java
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);

int result = 1 + 2; // result is now 3
if (value1 == value2)
    System.out.println("value1 == value2");
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression `cadence = 0` returns an `int` because the assignment operator returns a value of the same data type as its left-hand operand; in this case, `cadence` is an `int`. As you can see from the other expressions, an expression can return other types of values as well, such as `boolean` or `String`.

The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here’s an example of a compound expression:

```
1 * 2 * 3
```

In this particular example, the order in which the expression is evaluated is unimportant because the result of multiplication is independent of order; the outcome is always the same, regardless of the order of the numbers being multiplied. However, this is not true for all expressions. For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

```
x + y / 100 // ambiguous
```

You can specify exactly how an expression will be evaluated using balanced parentheses: ( and ). For example, to make the previous expression unambiguous, you could write the following:

```
(x + y) / 100 // unambiguous, recommended
```

If you don’t explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than the addition operator. Therefore the following two statements are equivalent:
\[ x + y / 100 \]
\[ x + (y / 100) \] // unambiguous, recommended

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first. This practice makes code easier to read and maintain.

**Statements**

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (\;):

- Assignment expressions
- Any use of ++ or --
- Method invocations
- Object creation expressions

Such statements are called *expression statements*. Here are some examples of expression statements:

```
// assignment statement
aValue = 8933.234;
// increment statement
aValue++;  
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Bicycle myBike = new Bicycle();
```

In addition to expression statements, there are two other kinds of statements: *declaration statements* and *control flow statements*. A *declaration statement* declares a variable. You’ve seen many examples of declaration statements already:

```
// declaration statement
double aValue = 8933.234;
```

Finally, *control flow statements* regulate the order in which statements get executed. You’ll learn about control flow statements in the next section, “Control Flow Statements.”
Expressions, Statements, and Blocks

Blocks

A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, BlockDemo, illustrates the use of blocks:

```java
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // begin block 1
            System.out.println("Condition is true.");
        } // end block one
        else { // begin block 2
            System.out.println("Condition is false.");
        } // end block 2
    }
}
```

Questions and Exercises: Expressions, Statements, and Blocks

Questions

1. Operators may be used in building ___, which compute values.
2. Expressions are the core components of ___.
3. Statements may be grouped into ___.
4. The following code snippet is an example of a ___ expression:
   
   ```java
   1 * 2 * 3
   ```

5. Statements are roughly equivalent to sentences in natural languages, but instead of ending with a period, a statement ends with a ___.
6. A block is a group of zero or more statements between balanced ___ and can be used anywhere a single statement is allowed.

Exercise

1. Identify the following kinds of expression statements:
   - aValue = 8933.234;
   - aValue++;
   - System.out.println("Hello World!");
   - Bicycle myBike = new Bicycle();
Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.

The if-then and if-then-else Statements

The if-then Statement

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code only if a particular test evaluates to true. For example, the Bicycle class could allow the brakes to decrease the bicycle’s speed only if the bicycle is already in motion. One possible implementation of the applyBrakes method could be as follows:

```java
void applyBrakes() {
    // the "if" clause: bicycle must be moving
    if (isMoving){
        // the "then" clause: decrease current speed
        currentSpeed--;
    }
}
```

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.

In addition, the opening and closing braces are optional, provided that the “then” clause contains only one statement:

```java
void applyBrakes() {
    // same as above, but without braces
    if (isMoving)
        currentSpeed--;
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the “then” clause,
a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you’ll just get the wrong results.

The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an “if” clause evaluates to false. You could use an if-then-else statement in the applyBrakes method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```java
void applyBrakes() {
    if (isMoving) {
        currentSpeed--;  
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

The following program, IfElseDemo, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on:

```java
class IfElseDemo {
    public static void main(String[] args) {
        int testscore = 76;
        char grade;
        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

The output from the program is as follows:

```
Grade = C
```

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60. However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.
The switch Statement

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with enumerated types (discussed in Chapter 4, “Enum Types”), the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer (discussed in Chapter 9).

The following code example, SwitchDemo, declares an int named month whose value represents a month. The code displays the name of the month, based on the value of month, using the switch statement:

```java
public class SwitchDemo {
    public static void main(String[] args) {
        int month = 8;
        String monthString;
        switch (month) {
            case 1: monthString = "January"; break;
            case 2: monthString = "February"; break;
            case 3: monthString = "March"; break;
            case 4: monthString = "April"; break;
            case 5: monthString = "May"; break;
            case 6: monthString = "June"; break;
            case 7: monthString = "July"; break;
            case 8: monthString = "August"; break;
            case 9: monthString = "September"; break;
            case 10: monthString = "October"; break;
            case 11: monthString = "November"; break;
            case 12: monthString = "December"; break;
            default: monthString = "Invalid month"; break;
        }
        System.out.println(monthString);
    }
}
```

In this case, August is printed to standard output.

The body of a switch statement is known as a switch block. A statement in the switch block can be labeled with one or more case or default labels. The switch
statement evaluates its expression and then executes all statements that follow the matching case label.

You could also display the name of the month with if-then-else statements:

```java
int month = 8;
if (month == 1) {
   System.out.println("January");
} else if (month == 2) {
   System.out.println("February");
}
// ... and so on
```

The choice between if-then-else statements or a switch statement depends on readability and the expression that the statement is testing. An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

Another point of interest is the break statement. Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block. The break statements are necessary because without them, statements in switch blocks fall through: All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered. The program SwitchDemoFallThrough shows statements in a switch block that fall through; it displays the month corresponding to the integer month and the months that follow in the year:

```java
public class SwitchDemoFallThrough {
   public static void main(String[] args) {
      java.util.ArrayList<String> futureMonths =
            new java.util.ArrayList<String>();

      int month = 8;

      switch (month) {
         case 1: futureMonths.add("January");
         case 2: futureMonths.add("February");
         case 3: futureMonths.add("March");
         case 4: futureMonths.add("April");
         case 5: futureMonths.add("May");
         case 6: futureMonths.add("June");
         case 7: futureMonths.add("July");
         case 8: futureMonths.add("August");
         case 9: futureMonths.add("September");
         case 10: futureMonths.add("October");
         case 11: futureMonths.add("November");
         case 12: futureMonths.add("December");
            break;
      default: break;
   }
```
if (futureMonths.isEmpty()) {
    System.out.println("Invalid month number");
} else {
    for (String monthName : futureMonths) {
        System.out.println(monthName);
    }
}

This is the output from the code:

August
September
October
November
December

Technically, the final break is not required because flow falls out of the switch statement. Using a break is recommended so that modifying the code is easier and less error prone. The default section handles all values that are not explicitly handled by one of the case sections.

The following code example, SwitchDemo2, shows how a statement can have multiple case labels. The code example calculates the number of days in a particular month:

class SwitchDemo2 {
    public static void main(String[] args) {
        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1: case 3: case 5: case 7: case 8: case 10: case 12:
                numDays = 31;
                break;
            case 4: case 6: case 9: case 11:
                numDays = 30;
                break;
            case 2:
                if (((year % 4 == 0) && !(year % 100 == 0))
                    || (year % 400 == 0))
                    numDays = 29;
                else
                    numDays = 28;
                break;
            default:
                System.out.println("Invalid month.");
        }
    }
}
break;
        }
        System.out.println("Number of Days = " + numDays);
        }
    }

This is the output from the code:

    Number of Days = 29

**Using Strings in switch Statements**

You can use a String object in the switch statement's expression. The following code example, StringSwitchDemo, displays the number of the month based on the value of the String named month:

```java
public class StringSwitchDemo {
    public static int getMonthNumber(String month) {
        int monthNumber = 0;
        if (month == null) {
            return monthNumber;
        }

        switch (month.toLowerCase()) {
            case "january":
                monthNumber = 1;
                break;
            case "february":
                monthNumber = 2;
                break;
            case "march":
                monthNumber = 3;
                break;
            case "april":
                monthNumber = 4;
                break;
            case "may":
                monthNumber = 5;
                break;
            case "june":
                monthNumber = 6;
                break;
            case "july":
                monthNumber = 7;
                break;
            case "august":
                monthNumber = 8;
                break;
            case "september":
                monthNumber = 9;
                break;
            default:
                return monthNumber;
        }
    }  
```
case "october":
    monthNumber = 10;
    break;
case "november":
    monthNumber = 11;
    break;
case "december":
    monthNumber = 12;
    break;
default:
    monthNumber = 0;
    break;
}
return monthNumber;
}

public static void main(String[] args) {
    String month = "August";
    int returnedMonthNumber =
            StringSwitchDemo.getMonthNumber(month);
    if (returnedMonthNumber == 0) {
        System.out.println("Invalid month");
    } else {
        System.out.println(returnedMonthNumber);
    }
}

The output from this code is 8.

The String in the switch expression is compared with the expressions associated with each case label, as if the String.equals method was being used. In order for the StringSwitchDemo example to accept any month regardless of case, month is converted to lowercase (with the toLowerCase method) and all the strings associated with the case labels are in lowercase.

**Note**

This example checks if the expression in the switch statement is null. Ensure that the expression in any switch statement is not null to prevent a NullPointerException from being thrown.

9. 8/docs/api/java/lang/String.html#equals-java.lang.Object-
10. 8/docs/api/java/lang/String.html#toLowerCase--
The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as follows:

```java
while (expression) {
    statement(s)
}
```

The while statement evaluates `expression`, which must return a boolean value. If the expression evaluates to true, the while statement executes the `statement(s)` in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished via the following `WhileDemo` program:

```java
class WhileDemo {
    public static void main(String[] args) {
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: "+ count);
            count++;
        }
    }
}
```

You can implement an infinite loop using the while statement as follows:

```java
while (true) {
    // your code goes here
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

```java
do {
    statement(s)
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following `DoWhileDemo` program:

```java
class DoWhileDemo {
    public static void main(String[] args) {
        int count = 1;
        do {
```
The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the for loop because of the way it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```java
for (initialization; termination; increment) {
    statement(s)
}
```

When using this version of the for statement, keep the following in mind:

- The initialization expression initializes the loop; it’s executed once as the loop begins.
- When the termination expression evaluates to false, the loop terminates.
- The increment expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment or decrement a value.

The following program, ForDemo, uses the general form of the for statement to print the numbers 1 through 10 to standard output:

```java
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: " + i);
        }
    }
}
```

Here is the output of this program:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```
Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the `for` statement, so it can be used in the termination and increment expressions as well. If the variable that controls a `for` statement is not needed outside the loop, it’s best to declare the variable in the initialization expression. The names i, j, and k are often used to control `for` loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the `for` loop are optional; an infinite loop can be created as follows:

```java
// infinite loop
for ( ; ; ) {
    // your code goes here
}
```

The `for` statement also has another form designed for iteration through collections and arrays. This form is sometimes referred to as the `enhanced for` statement and can be used to make your loops more compact and easier to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```java
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, `EnhancedForDemo`, uses the enhanced `for` to loop through the array:

```java
class EnhancedForDemo {
    public static void main(String[] args){
        int[] numbers =
            {1,2,3,4,5,6,7,8,9,10};
        for (int item : numbers) {
            System.out.println("Count is: "+item);
        }
    }
}
```

In this example, the variable `item` holds the current value from the `numbers` array. The output from this program is the same as before:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```
We recommend using this form of the `for` statement instead of the general form whenever possible.

### Branching Statements

#### The break Statement

The `break` statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the `switch` statement. You can also use an unlabeled `break` to terminate a `for`, `while`, or `do-while` loop, as shown in the following `BreakDemo` program:

```java
class BreakDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = {
            32, 87, 3, 589,
            12, 1076, 2000,
            8, 622, 127
       };
        int searchfor = 12;
        int i;
        boolean foundIt = false;
        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }
        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

This program searches for the number 12 in an array. The `break` statement, shown in boldface, terminates the `for` loop when that value is found. Control flow then transfers to the statement after the `for` loop. This program's output is as follows:

```
Found 12 at index 4
```

An unlabeled `break` statement terminates the innermost `switch`, `for`, `while`, or `do-while` statement, but a labeled `break` terminates an outer statement. The following program, `BreakWithLabelDemo`, is similar to the previous program but uses nested `for` loops to search for a value in a two-dimensional array. When the value is found, a labeled `break` terminates the outer `for` loop (labeled `search`):
class BreakWithLabelDemo {
    public static void main(String[] args) {
        int[][] arrayOfInts = {
            {32, 87, 3, 589},
            {12, 1076, 2000, 8},
            {622, 127, 77, 955}
        }
        int searchfor = 12;
        int i;
        int j = 0;
        boolean foundIt = false;

        search:
        for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length; j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at " + i + ", " + j);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}

This is the output of the program:

    Found 12 at 1, 0

The break statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

The continue Statement
The continue statement skips the current iteration of a for, while, or do-while loop. The unlabeled form skips to the end of the innermost loop’s body and evaluates the boolean expression that controls the loop. The following program, ContinueDemo, steps through a String, counting the occurrences of the letter p. If the current character is not a p, the continue statement skips the rest of the loop and proceeds to the next character. If it is a p, the program increments the letter count:

class ContinueDemo {
    public static void main(String[] args) {

This is the output of the program:

    Found 12 at 1, 0

The break statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.
String searchMe = "peter piper picked a " + "peck of pickled peppers";
int max = searchMe.length();
int numPs = 0;

for (int i = 0; i < max; i++) {
    // interested only in p's
    if (searchMe.charAt(i) != 'p')
        continue;

    // process p's
    numPs++;
}

System.out.println("Found " + numPs + " p's in the string.");

Here is the output of this program:

    Found 9 p's in the string.

To see this effect more clearly, try removing the continue statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

A labeled continue statement skips the current iteration of an outer loop marked with the given label. The following example program, ContinueWithLabelDemo, uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and another to iterate over the string being searched. The following program, ContinueWithLabelDemo, uses the labeled form of the continue statement to skip an iteration in the outer loop:

class ContinueWithLabelDemo {
    public static void main(String[] args) {

        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() - 
                  substring.length();

        test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++) != substring.charAt(k++)) {
                    continue test;
                }
            }
        }
    }
}
Control Flow Statements

```java
foundIt = true;
break test;
}
System.out.println(foundIt ? "Found it" : "Didn't find it");
```

Here is the output from this program:

```plaintext
Found it
```

**The return Statement**

The last of the branching statements is the `return` statement. The `return` statement exits from the current method, and control flow returns to where the method was invoked. The `return` statement has two forms: one that returns a value and another that doesn’t. To return a value, simply put the value (or an expression that calculates the value) after the `return` keyword:

```java
return ++count;
```

The data type of the returned value must match the type of the method’s declared return value. When a method is declared `void`, use the form of `return` that doesn’t return a value:

```java
return;
```

Chapter 4 covers everything you need to know about writing methods.

**Summary of Control Flow Statements**

The `if-then` statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code only if a particular test evaluates to `true`. The `if-then-else` statement provides a secondary path of execution when an “if” clause evaluates to `false`. Unlike `if-then` and `if-then-else`, the `switch` statement allows for any number of possible execution paths. The `while` and `do-while` statements continually execute a block of statements while a particular condition is true. The difference between `do-while` and `while` is that `do-while` evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the `do` block are always executed at least once. The `for` statement provides a compact way to iterate over a range of values. It has two forms, one of which was designed for looping through collections and arrays.
Questions and Exercises: Control Flow Statements

Questions

1. The most basic control flow statement supported by the Java programming language is the ___ statement.
2. The ___ statement allows for any number of possible execution paths.
3. The ___ statement is similar to the while statement but evaluates its expression at the ___ of the loop.
4. How do you write an infinite loop using the for statement?
5. How do you write an infinite loop using the while statement?

Exercises

1. Consider the following code snippet:
   ```java
   if (aNumber >= 0)
       if (aNumber == 0)
           System.out.println("first string");
       else System.out.println("second string");
   System.out.println("third string");
   ```
   a. What output do you think the code will produce if aNumber is 3?
   b. Write a test program containing the previous code snippet; make aNumber 3. What is the output of the program? Is it what you predicted? Explain why the output is what it is; in other words, what is the control flow for the code snippet?
   c. Using only spaces and line breaks, reformat the code snippet to make the control flow easier to understand.
   d. Use braces, { and }, to further clarify the code.

Answers

You can find answers to these questions and exercises at http://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/answers_flow.html.
This page intentionally left blank
Index

Symbols
- (minus sign)
  operator, 59–62, 279
  in regular expressions, 562–64
-- operator, 59, 61, 62
- (underscore)
  in constant names, 114
  in numeric literals, 50–51
  in package names, 263
  in predefined character classes, 566
  in variable names, 45–46
, (comma)
  in numbers, 50, 276, 278
  in regular expressions, 572
; (semicolon)
  in class paths, 212, 269, 610
  declaring abstract methods, 159
  listing enum types, 159
  in statements, 28, 70, 319
  terminating method signatures, 176, 178
: (colon), in class paths, 269
! operator, 59
/ separator, 644, 645
!= operator, 59, 63, 66
? (question mark)
  in regular expressions, 236, 372–73
?: operator, 64, 67
/ (forward slash)
  file name separator, 267, 360, 417, 621
  operator, 59–60, 279–80
// in comments, 12, 25
/* in comments, 24
/** in comments, 24
/= operator, 59
. (dot)
  in class paths, 364, 612
  in JAR file commands, 620, 630
  in method invocations, 87, 104–7
  in numbers, 278, 287, 299
  in regular expressions, 561, 568
  in variable names, 105
… (ellipses), 96–97
^ (caret)
  operator, 59, 66–67
  in regular expressions, 563, 576–77, 579
^= operator, 59
~ operator, 59, 65
' (single quote), escape sequence for, 50, 288
" (double quote)
  escape sequence for, 50, 288
  in literals, 288
() (parentheses)
  in declarations, 92, 106, 317
  in expressions, 69, 797
  in generics, 222
  in interfaces, 133
  in regular expressions, 573–74
[] (square brackets)
  in arrays, 53–54
  in regular expressions, 373, 568, 592
{} (braces)
  in blocks, 71, 72, 117, 127
  in declarations, 89–90, 90–92, 127–28
  in lambda expressions, 147
  in methods, 44
  in regular expressions, 373, 571–72
@ (at)
  in annotations, 164, 166
  in Javadoc, 167
$ (dollar sign)
  in DecimalFormat patterns, 278
  in variable names, 45
* (asterisk)
  in import statements, 264
  operator, 59
  in regular expressions, 372, 407
*/ in comments, 24
* = operator, 59
\ (backslash)
  in escape sequences, 50, 288–89, 360, 566
  file name separator, 267, 417
  in regular expressions, 373, 561
& (ampersand) operator, 59, 66–67
&& operator, 59, 63, 66–67
&= operator, 59
# (pound sign)
  in DecimalFormat patterns, 278
  in regular expressions, 561
% (percent sign)
  format specifier, 275, 350
  operator, 59, 279–80
%= operator, 59
+ (plus sign)
  operator, 59, 66–67, 279–80
  in regular expressions, 561
++ operator, 59, 61–62, 66–67
+= operator, 59
< operator, 59, 62–63
<= operator, 59, 66–67
< = operator, 59, 62–63, 66–67
<> (angle brackets), 141, 221, 223, 224–27, 232
= operator, 59
-= operator, 59
== operator, 59, 62–63, 66–67
> operator, 59, 62–63, 66–67
>= operator, 59, 62–63, 66–67
>> operator, 59, 66–67
>>= operator, 59
>>> operator, 59
>>> = operator, 59
-> (arrow token), 147, 487
| (vertical bar)
  in exception handling, 316
  operator, 59, 66–67
  in regular expressions, 579
|= operator, 59
|| operator, 59, 63, 66–67

A
abs method, 280
abstract classes, 212–14
  example, 214–15
  as an implementation of a service, 603
  implementations, 512
  methods, 212–13
  numeric wrapper classes, 272
  versus interfaces, 213–14
Abstract Window Toolkit (AWT), 265
  AWT Event Dispatcher, 652, 672
AbstractMap class, 214
access control list (ACL), 380
access modifiers
  classes and, 90–95, 111
  constants and, 178
  default, 110–11
  fields and, 196
  interfaces and, 177–78
  levels of, 110–12
  methods and, 92, 196, 206
  package-private, 111
  private keyword, 89, 110–11, 196–97
  protected, 111–12
  public, 89, 110–11, 196–97
AccessControlException, 608
accessor methods, 290, 295, 296, 381, 382, 467
accumulator function, 476–77
acos method, 282–83
add method, 429, 434, 442, 446
addAll method
  in the Collection interface, 428–29, 430, 432, 452
  in the List interface, 438, 439
  in the Map interface, 452
addFirst method, 448–49, 494, 498
addLast method, 448–49, 494, 498
aggregate operations, xxiv, 145–46, 450, 471–72
  bulk operations vs., 430–31
  iterators vs., 474
  side effects of, 484–87
  traversing collections with, 429–30
algorithms (collections), 423–25, 500–509
  in the Collections class, 508–9
  composition, 509
  finding extreme values with, 509
  generic, 220, 229
  listing data, 445
  polymorphism, defined, 424
  routine data manipulation, 508
  searching data, 508–9
  shuffling data, 508
Index

- sorting data, 505–8
- work stealing, 549
- ampersand. See &
- Anagrams example, 456, 507
- angle brackets. See <>
- annotations, 163–74
- cardinality of type, 173
- container, 172
- declaring, 165–67, 172
- design considerations, 173
- elements and, 164
- formatting of, 164–65
- legacy code, 173
- meta-annotations, 169–70
- predefined, 165, 167–70
- repeating, 164–65, 171–73
- retrieving, 173
- type. See type annotations
- used by the Java language, 167–68
- where to use, 165
- anonymous classes, 131–36
- declaring and accessing, 131–34
- examples of, 134–36
- GUI applications and, 134
- specifying search criteria code in, 140
- syntax of, 132–33
- when to use, 155
- APIs (Application Programming Interfaces), 138
- array-based versus collection-based, 502
- compatibility of, 513–15
- design of, 515–17
- interfaces as, 177
- JAR-related, 642–648
- Java core, 4, 34, 40
- logging, 328
- Reflection API, 173
- append method, 253, 302–4
- appendReplacement method, 584–85, 588
- appendTail method, 584, 588
- Applet class, 665, 667, 672, 680–81, 684
- applet tag, 626, 673–76
- deploying with, 676
- JAR files and, 626
- JNLP and, 676, 734–35, 748
- manually coding, 676
- AppletContext interface, 677, 682–83, 702, 706
- applets, 665–709
- API of, 677
- background color of, 733
- common problems, 707–8
- communicating with other applets, 701–3
- core functionality versus deployment mechanism, 673–74
- debugging, 698
- defining and using applet parameters, 678–80
- deploying, 673–76, 733–35
- developing, 670–73
- directories of, 673–75, 678
- displaying documents, 682–83
- displaying short status strings, 681
- draggable, 698–701
- event handling and, 689–90
- execution environment of, 670
- finding and loading data files, 677
- GUIs in, 671
- JavaScript functions and, 670, 676–77, 701–3
- leaving and returning to web pages, 669
- life cycle of, 668–69
- loading, 669
- milestones, 667–68
- packing in JAR files, 673–75
- parameters in, 668–670
- qualified names, 659, 697
- quitting the browser, 669
- reloading, 669
- sandbox, 706–7
- security and, 596, 634, 677
- server-side applications, 703–5
- signed, 31, 654
- threads in, 670
- appletviewer application, 608
- Application-Library-Allowable-Codebase attribute, 635
- Application-Name attribute, 634
- applications. See rich Internet applications (RIAs)
- archive attribute, 708, 733
- args variable, 45, 274–75, 601, 646–47
- arguments
  - arbitrary number of, 96–97
  - command-line. See command-line arguments
  - glob, 372–73
  - number of, 95
  - primitive data types, 96, 98
  - reference data types, 98
  - versus parameters, 95
- arithmetic operators, 59–60, 61, 66
- ArithmeticDemo example, 60, 68
- ArrayBlockingQueue class, 497
- arraycopy method, 55–56, 608, 648
- ArrayCopyDemo example, 55–56
- ArrayDemo example, 52–54, 55
- ArrayDeque class, 498
- arrays, 51–57
  - assigning values to, 54
  - of characters, 288–89
  - comparing, 56
arrays (continued)
copying, 55–56
creating, 54–55
filling, 56
List view of, 502
looping through, 85
multidimensional, 54–55
searching, 56
sorting, 57
arrow token. See ->
asin method, 282
as a convenience implementation, 502
upward compatibility and, 513
writing a custom implementation, 511
asList method, 237–38, 440
assert statement, 120
assignments
checking with assert, 120
compound, 61, 67
conditional operators and, 63–64
asterisk. See *
at. See @
at prefix, 758
atan method, 282–83
atomic file operations, 372, 377, 392, 496, 500
access, 533
actions, 533
methods, 496
synchronization, 500–501, 533, 553–54
variables, 553–54
ATOMIC_MOVE enum, 372, 377
AtomicCounter example, 554
AtomicInteger class, 272, 554
Attributes class, 645
autoboxing, 223, 253, 271–72, 283–84, 285, 288
AutoCloseable interface, 317, 320
autoflush, 346
buffers, 345–46
bugs. See errors
byte data type, 46
data streams and, 390
default value of, 49
switch statement and, 74
byte streams, 340–42
buffered, 346
character streams and, 342–43
classes, 348–49
closing, 341–42
I/O streams and, 352, 354–55
standard streams and, 352
using, 341
when not to use, 342
bytecodes, 2, 5, 7, 16, 18, 20
in the HelloWorld example, 13, 16, 18, 20,
29–30
type erasure and, 244–45
byteValue method, 273, 286

C
CA. See Signer Certificate Authority (CA)
keystore
Calendar class, 358
call stack
exception handling, 310–11, 315, 321, 323
propagating errors up, 332–33
Callable objects, 547–48
Caller-Allowable-Codebase attribute, 635, 686
capturing groups, 572–73
cascading style sheets (CSSs), 793
catch blocks, 315–16, 321, 336, 370, 800
Catch or Specify Requirement, 309–12, 330, 799
by passing, 312
cd command, 18, 21–22, 29–30
ceil method, 280
char data type, 47
character and string literals, 50, 288–90
  converting to strings, 288–90
  in data streams, 354–55
  default value of, 48
  escape sequences in, 50, 288–90
  generic methods and bounded type
  parameters, 229
  getting by index, 295
  translating individual tokens, 347–48
  wrapper class. See Character class
Character class, 287–89, 306
  implementing Comparable, 459
  restrictions on generics, 252–56
  switch statement and, 74
  useful methods in, 289
  as a wrapper class, 287
classes, 36–38, 88–89, 118–19. See also
  inheritance; nested classes
  abstract, 212–16
  access modifiers and, 90–91
  adapter, 517
  base or parent. See superclasses
  child, derived, or extended. See subclasses
  constructors for, 94–95
  declaring, 89–90, 127–28
  final, 212
  hierarchy of, 194
  inner, 130–31. See also inner classes
  instantiating, 101
  interfaces implemented by, 90, 178
  local, 127–31, 139–40, 155
  methods and, 92–94
  naming, 92
  numbers, 272–74
  passing information, 95–99
  static initialization blocks in, 117
  variables (static fields), 44, 57
  wrapper, 272, 283–87, 306, 797
ClassNotFoundException, 357, 646, 648, 708
CLASSPATH system variable, 29–30, 268–69, 595,
  609, 611–13, 706
ClipboardService interface, 715
Clock class, 783
close method, 319–20, 347, 370
Cloneable interface, 209, 213–14
CloneNotSupportedException, 208–9
close method, 319–20, 347, 370
Closeable interface, 313, 317, 320, 370
Cmd command, 17
code
  case sensitivity in, 12, 17, 22, 45
  error handling, 331–32
  error-prone, 76, 107, 171, 335
  platform-independent, 4, 31
  readability of, 31, 64, 94, 122, 155, 267, 311,
  329, 346
Codebase attribute, manifest file, 634, 724
codebase attribute, JNLP file, 733, 734, 737
CollationKey class, 459
collect method, 476–80
Collection interface, 428–29, 432
  array operations, 432–33
  backward compatibility and, 514
  bulk operations, 432
  implementations of, 502
Collection interface (continued)
views, 452–54
wrappers for, 499–500
collections, 423–517
concurrent, 552–53
hierarchy of, 265
internal delegation and, 474
older APIs and, 225, 432–33, 512–13
ordered, 427, 438, 469, 495
read-only access to, 490
synchronized, 490, 500–501
traversing, 429–30
Collections class, 224, 230, 439, 445, 504, 508–9
backward compatibility and, 224–25, 230–31
methods in, 499–500
polymorphic algorithms in, 424
Collectors class, 224, 230, 439, 445, 504, 508–9
backward compatibility and, 224–25, 230–31
methods in, 499–500
colon. See :
combiner function, 477
comma. See ,
command-line arguments, 292–93, 600–601
analogies to applet parameters, 678
echoing, 600
numeric, 601
test harnesses and, 559–60
URLs and, 646–47
comments, 24
annotations and, 165–67
Pattern class methods, 578, 580
Comparable interface, 369, 458, 462
Comparator interface, 189–92, 461–64, 497, 506–7, 509, 796, 798
compare method, 462
compareTo method, 462
compareToIgnoreCase method, 154, 301
ComparisonDemo example, 63
comparisons
between classes, 459
of numbers, 59–62
of object, 64–65
compatibility, 513–15
backward, 514–15
binary, 183, 187
cross-platform, 794
upward, 513–14
compile method, 559, 578
compilers, 124, 198, 234
information for, 163
ComputeResult class, 307
concat method, 291
ConcatDemo example, 61
concurrency, 519–56
collections, 552–53
high-level objects, 543–55
random numbers, 554
ConcurrentHashMap implementation, 214, 496, 553
ConcurrentMap interface, 482, 491, 496, 552–53
ConcurrentNavigableMap interface, 553
ConcurrentSkipListMap interface, 553
conditional operators, 62–64, 67, 797
ConditionalDemo1 example, 63
ConditionalDemo2 example, 64
constants, 115
compile-time, 115
data streams and, 354–55
embedded flag expressions, 580
empty, 503–4
enum types and, 157–61
importing, 265–66
interfaces and, 176, 177–78
naming, 45–46, 157–58
numbers and, 278–79, 286
for upper and lower bounds, 238–39, 272
variables, 130–31
constructors, 87–89, 95–99
calling, 108–10
chaining, 208
conversion, 428–29
declaring, 89, 94–95
default, 95
for enum types, 159
generic, 234
inheritance and, 194, 207–8
methods and, 95, 212
no-argument, 95, 104, 110, 207, 783
synchronization and, 530
Consumer interface, 142–43, 146
containers. See collections
contains method, 296, 297, 428, 599
containsAll method, 429, 430, 430–32, 454, 494
containsKey method, 449, 451, 599
containsValue method, 214, 449, 451
continue statements, 83–84
ContinueDemo example, 83
ContinueWithLabelDemo example, 84
control flow statements, 72–86
branching, 82–85
decision-making, 72
controlling access. See access modifiers
converters, 275–76
Index

Cookie Applet Example, 722
cookies
  accessing, 719–22
  kinds of, 719
  rich Internet applications (RIAs) and, 719–20, 722
copy method, 243, 376
CopyBytes example, 341–42, 343
CopyCharacters example, 343–44, 345
CopyLines example, 344–45
copyOfRange method, 56
CopyOnWriteArrayList implementation, 494
CopyOnWriteArraySet implementation, 493–94
core collection interfaces, 423, 426–28, 469–70, 500–501, 513. See also by individual type
  compatibility of, 513–14
  hierarchy of, 265
  implementations of, 499–502
cos method, 266, 279
Countdown example, 447
Counter example, 527
CreateObjectDemo example, 99–100, 105–6
createTempFile method, 393, 421
currentTimeMillis method, 609
customized loading screens
  in applets, 722
  in Java Web Start applications, 656–61
D
data encapsulation, 33, 35
data types, 46–51, 95–96, 341–42. See also by individual type
  reference, 95–96
  returned by expressions, 68–69, 91–92
  switch statement and, 74
DataInput interface, 355–57
DataInputStream class, 355–56
DataOutput interface, 355, 356, 357
DataOutputStream class, 355, 356
DataStreams example, 355–57
dates, 153, 621. See also Date-Time package
Date-Time package, xxiv, 755–91
  basic representations of time, 759–60
  calendar systems, xxiv, 755–56, 759, 788
  clarity in, 756
  clocks, 783–84
date-time classes, 764–70
design principles, 756–57
duration, 781
epochs, 759, 770
  extensibility of, 757
  fluent interface of, 757
  formatting 773–74
  human vs. machine time, 770
  immutability of, 757
  Instant class and, 770–71
  legacy date-time code, 787–90
  method naming conventions, 758
  non-ISO date conversions, 784–87
  packages, 757–58, 774–80
  parsing 772–73
  period, 782–83
  temporal-based classes, 760–61
time zone and offset classes, 766–70
DateTimeFormatter class, 772
DayOfWeek enum, 760–62
Deadlock example, 534
deadlocks, 491, 533–35, 544
Deal example, 444–45
decimal number system, 48, 49, 273
DecimalFormat class, 272, 277–78
declaration statements, 70, 317
declarations. See by individual type
decode method, 274
default keyword, 185
default methods, xxiii, 182–92
  binary compatibility, 183
  defining implementations for, 182–84
  defining new methods as, 182
  extending interfaces that contain, 185–86
  integrating into existing libraries, 187–92
DelayQueue class, 497
delete method, 304, 375, 419
deleteOnExit method, 420
deployment, 729–53
  applets, 673–77
  best practices, 748
  Java Web Start applications, 653–56
Deployment Toolkit, 653–56, 673–76
@Deprecated annotation type, 167–68
@deprecated Javadoc tag, 167
Deque interface, 448–49
  basic operations, 498
  concurrent implementations, 499
  implementations of, 498–99
  methods, 448–49
destroy method, 668
diamond, 223–24, 233–34, 428
dir command, 18, 29
directories
  changing, 17–19, 27
  checking, 374–75
  copying, 376–77
  creating, 20–22, 395–96
deleting, 375
directories (continued)
delimiters, 360
error messages involving, 29
filtering, 398
moving, 377–78
packages, 267–68
root, 363, 395
temporary, 396–97
verifying the existence of, 375
watching for changes, 410–16
documentation, 165–66
source code comments, 24
@Documented annotation type, 167, 169
dollar sign. See $
dot. See .
double data type, 47
double quote. See "
doubleValue method, 237, 273
do-while statements, 53, 72, 79, 82–83, 85, 798
DoWhileDemo example, 79
DownloadService interface, 657, 664, 693, 694, 715
DownloadServiceListener interface, 657, 693–94, 715
Duration object, 780–81
DynamicTreeDemo applet, 654, 674
E
E constant, 279
in scientific notation, 49
as type parameter naming convention, 221
Echo example, 600
element method, 438
elements (in collections), 427–28
adding, 428–29, 432, 438–39
checking, 431
counting, 433–34, 447
cursor position and, 441–442
not duplicated, 433–434
null, 432
ordering, 427, 433–434
removing, 429–432, 435–436
searching, 435–436
sequence of, 145
swapping, 439–440
ellipsis. See . . .
emacs text editor, 20
emptyList method, 235, 503
emptyMap method, 503
emptySet method, 503
EmptyStackException, 325
encapsulation, 33, 35, 91, 122, 155, 796, 799
end method, 583
endsWith method, 301, 369
EnhancedForDemo example, 81
ensureCapacity method, 303, 501
Entry-Point attribute, 635
tentrySet method, 449, 453–55, 512
enum keyword, 157
enum types, 87, 157–61, 493
constructors for, 159
naming, 157–58
enumerated types. See enum types
Enumeration collection, 259, 513, 514, 515, 599
compatibility and, 513–15
EnumMap implementation, 495
EnumSet implementation, 493
EnumTest example, 158
Env example, 602
environment, 595–614
properties of, 596–99
restricted, 661
environment variables, 601–2, 609–13
CLASSPATH, 29, 30, 609
common problems with, 29
passing to new processes, 602
PATH, 609–12
platform dependency issues, 602
querying, 601–2
EnvMap example, 601–2
EOFException, 335, 356
epochs, 759, 770
equality operators. See comparisons
equals method, 56, 208, 210–11, 460, 463, 496
equalsIgnoreCase method, 301
Error class, 326
error messages, 313–15
legacy file I/O code, 418
Microsoft Windows, 27, 29–30
Solaris and Linux, 27–28, 30–31
unchecked, 225–26
using to check assignments, 120
wildcard capture and, 241
errors
compiler, 28, 58, 101, 105, 107, 201, 229, 241,
255, 264, 330
compile-time, 48, 95, 172, 197, 203, 207, 220,
228, 239, 244, 253–56, 284
grouping and differentiating types, 334–35
memory consistency, 527, 528–29, 530–31, 533,
543, 553
propagating in, 332–34
runtime, 29–31
semantic, 29
syntax, 28
escape sequences, 50, 288–89
in regular expressions, 578, 582
in Unicode, 50, 591
EventHandler interface, 134–35, 146
exception classes, 309, 324–29
creating, 328–29
grouping errors, 334–35
hierarchy, 329
PatternSyntaxException class, 589–91
exception handlers, 45, 310–17
associating with try blocks, 314–15
catching more than one exception type, 316
catching multiple exceptions, 319
constructing, 320–23
exceptions, 309–37
advantages of, 329–30
catching, 313–23
chained, 326–28
checked, 312
class hierarchy of, 329
creating exception classes, 328
e external. See errors
in file operations, 370–71
kinds of, 311–12
logging, 328, 698
specifying by method, 323–24
suppressed, 319–20
throwing, 324–30
unchecked, 329–30
exclamation sign. See !
Executor interface, 546–48, 549
ExecutorService interface, 546–49
exit method, 609
exp method, 281
ExponentialDemo example, 281
exponents, 266
expression statements, 70–71
expressions, 68–70
ExtendedService interface, 664, 715
extends keyword, 38, 227, 236, 238
extensions, 603, 613, 615, 625, 638, 673, 723

F
F or f in 32-bit float literals, 47
fields, 35–38, 206, 796. See also variables
declaring, 117
default values of, 48
final, 530
hiding, 206
inherited, 196
initializing, 116–18, 796
members versus, 45
nonstatic, 44, 57
private, 196
qualified names, 98
referencing, 104–5
shadowing, 97, 123–24
static, 44, 112–14, 216, 254, 531, 578, 645
static final. See constants
synchronization and, 527, 531
FIFO (first-in, first out), 427, 446, 469, 490, 497
File class, 359
file descriptors, 211
file operations, 370–74
atomic, 372
catching exceptions, 370–71
method chaining, 372
releasing system resources, 370
varargs in, 371–72
file paths, 359–62
checking symbolic links, 375–76
comparing, 369
converting, 366–67
creating, 363
creating a path between two, 368
joining two, 367
relative versus absolute, 360–61
removing redundancies from, 364–66
retrieving information about, 363–64
symbolic links and, 361–62
FileInputStream class, 341, 343, 356, 598, 606
Filename class, 298
FilenameDemo example, 299
FileNotFoundException, 312, 334–35
FileOpenService interface, 715–17
FileOutputStream class, 341, 344, 355, 598, 608
FileReader class, 311–12, 317–18, 343–48, 608
files
accessibility of, 376
basic attributes, 381
checking, 374–75
copying, 376–77
creating, 389–90, 392–93
deleting, 375
DOS attributes, 378
file stores, 418
finding, 407–8
I/O and, 359–420
moving, 377–78
POSIX file permissions, 383–84
random access, 339, 359, 390, 393, 420
files (continued)
reading, 389–90
setting ownership, 384
temporary, 393
time stamps, 382
user-defined attributes, 385
verifying the existence of, 375
writing, 389–90

FileSaveService interface, 715–18
FileSystem class, 373, 386, 407, 412, 422
FileVisitor interface, 401–5
FileWriter class, 313–14, 316, 320–23, 343–45, 608
fill method, 56
final
catch parameter, 316
class, 212
class variable, 44, 115
constants, 115
effectively, 129, 133, 149-150
immutable objects, 541, 542
method, 118, 212
final modifier, 115, 178
finalize method, 208, 210–11
finally block, 309, 316–17
find method, 584
FindDups example, 435–37, 470
FindDups2 example, 437
first method, 448
float data type, 47
default value of, 48
floatValue method, 273, 293
floor method, 280
flush method, 346
for statement, 80–82, 85
enhanced, 81, 395, 418, 470, 488, 798
nested, 82
skipping the current iteration, 84–85
terminating, 80
ForDemo example, 80–81
for-each construct, 159, 429, 430, 434, 435, 472–73, 499
fork/join framework, 480, 546, 549–50, 552
form feed, 50
Format example, 352
format method, 274–75, 278, 350, 758, 772
format specifiers, 275, 350–51, 644, 772
format strings, 275, 292, 349–50
Formatter class, 350
formatting
numeric print output, 274–78
stream objects, 346, 349–52
forward slash. See /
frequency method, 451–52
from method, 758
functional interface, 141–42
@FunctionalInterface annotation type, 169
functions. See methods
Future object, 547
FXML scripting language, 793
G
garbage collection, 5, 106–7, 119, 120, 208, 210, 496, 540, 797
empty references and, 208
immutable objects and, 539–40
memory leaks and, 5
weak references and, 496
generic methods, 226–27
generic objects, 219–58
bounded type parameters and, 227
erasure of, 244–46
instantiating, 222, 233–34
invoking, 222
subtype and, 230–31
type inference and, 232–35
generic types, 141–42, 220–26, 239, 245, 249, 252, 426
get method, 211, 313–14, 363, 758
getAbsolutePath method, 421
getApplet method, 683, 702
getCanonicalPath method, 421
getCause method, 327
getChars method, 291
getClass method, 208, 211, 216, 227
getCodeBase method, 678, 704
getDescription method, 589–90
getEnv method, 601–2
getFields method, 211
getFirst method, 449, 494, 498
getHost method, 704
getImage method, 678
getIndex method, 589–90
getInterfaces method, 211
getLast method, 449, 494, 498
getMainAttributes method, 644–45
getMainClassName method, 644–45
getLocalizedMessage method, 371, 589, 590
getName method, 365
getParameter method, 680–81
getParent method, 364–66
getPattern method, 589
getProperty method, 599, 605, 711–13
ggetResource method, 653

for-each construct, 159, 429, 430, 434, 435, 472–73, 499
fork/join framework, 480, 546, 549–50, 552
form feed, 50
Format example, 352
format method, 274–75, 278, 350, 758, 772
format specifiers, 275, 350–51, 644, 772
format strings, 275, 292, 349–50
Formatter class, 350
formatting
numeric print output, 274–78
stream objects, 346, 349–52
forward slash. See /
frequency method, 451–52
from method, 758
functional interface, 141–42
@FunctionalInterface annotation type, 169
functions. See methods
Future object, 547
FXML scripting language, 793
G
garbage collection, 5, 106–7, 119, 120, 208, 210, 496, 540, 797
empty references and, 208
immutable objects and, 539–40
memory leaks and, 5
weak references and, 496
generic methods, 226–27
generic objects, 219–58
bounded type parameters and, 227
erasure of, 244–46
instantiating, 222, 233–34
invoking, 222
subtype and, 230–31
type inference and, 232–35
generic types, 141–42, 220–26, 239, 245, 249, 252, 426
get method, 211, 313–14, 363, 758
getAbsolutePath method, 421
getApplet method, 683, 702
getCanonicalPath method, 421
getCause method, 327
getChars method, 291
getClass method, 208, 211, 216, 227
getCodeBase method, 678, 704
getDescription method, 589–90
getEnv method, 601–2
getFields method, 211
getFirst method, 449, 494, 498
getHost method, 704
getImage method, 678
getIndex method, 589–90
getInterfaces method, 211
getLast method, 449, 494, 498
getMainAttributes method, 644–45
getMainClassName method, 644–45
getLocalizedMessage method, 371, 589, 590
getName method, 365
getParameter method, 680–81
getParent method, 364–66
getPattern method, 589
getProperty method, 599, 605, 711–13
ggetResource method, 653
Index

getSecurityManager method, 607
getsimpleName method, 211
getStackTrace method, 327
getSuperclass method, 211
getValue method, 645
globbing, 372–74
    filtering a directory listing, 398
    finding files, 407–10
groupCount method, 575
groupingBy operation, 479, 482
groupingByConcurrent operation, 482
guarded blocks, 535–39
H
hard links, 359, 399
HashMap implementation, 214, 234, 451–52, 491, 495–96, 504
HashSet implementation, 433–37, 450, 454–55, 490–93
    compatibility and, 513–14
    concurrency through ConcurrentHashMap, 496
    synchronization and, 490–91
hasNext method, 431, 441
headSet method, 464–67
heap pollution, 250
HelloRunnable example, 521–22
HelloThread example, 522
HelloWorld, 6–24, 656–66
    applet, 665–66
    JavaFX example, 146
    for Microsoft Windows, 15–20
    for Solaris and Linux, 20–23
    for the NetBeans IDE, 6–15
hexadecimal number system, 49, 51, 211, 273–74, 286, 351, 591
HTML. See also web browsers; web pages
    generated code, 732, 736
    specification, 676
HTTP requests, 548, 719
HTTPS certificates, 662, 726
I
IDE projects, 7–9, 14–15
identity element, 476, 480
IdentityHashMap implementation, 495–96
IfElseDemo example, 73
if-then statements, 72–73, 85, 797
if-then-else statements, 73, 85, 797
IllegalAccessException, 646
IllegalStateException, 446
immutable objects, 539–43
    defining, 541–43
immutable singleton set, 503
ImmutableRGB example, 503
implementations, 489–505
    abstract, 490, 510–13
    adapter, 510–11
    anonymous, 499
    concurrent, 489, 491, 498–99
    convenience, 489, 502–4
    custom, 509–13
    documenting, 426
    general purpose, 489, 490, 491, 498, 502, 504, 510, 553
    multiple inheritance of, 198
    special purpose, 489, 493, 504
    wrapper, 424, 499–502
    writing, 510–13
implements keyword, 39, 89, 178
import statement, 264–66, 776
indexOf method, 299, 438–39, 442
indexOfSubList method, 445–46
information hiding, 36
inheritance, 38–39, 193–217, 797, 799
    example, 195–96
    multiple, 159, 198–99, 214
@Inherited annotation type, 170
init method, 667, 671–73, 690
initCause method, 327
initializer blocks, 117–18
inner classes, 87, 122, 123–27
    accessing members of, 128–29
    anonymous, 123
    compatibility issues, 125
    controlling access to, 122
    example, 125–27
    instantiating, 123
    local, 127–31
    serialization of, 124–25
InputStream class, 343–44
InputStreamReader class, 344, 352–53, 389, 718
insert method, 302
instance members, 87, 117–18, 122, 130
instance methods
    interface methods vs., 200–202
instance variables, 44, 57, 87, 112–19, 122, 698
instanceof operator, 59, 64–67, 198, 250, 254, 405, 459, 692
InstanceofDemo example, 64
instances, 36, 44, 101–2, 112–13
class members and, 112–16
inner classes and, 123
testing, 64–65
Instant class, 228, 273–74
interfaces, 39–40, 176–93
abstract classes and, 213–14, 799
as APIs, 177
body, 178
collection. See core collection interfaces
defining, 177–78
evolving, 181–82
functional, 141–42
implementing, 178–80
as a type, 180–81
International Organization for Standardization (ISO) calendar system, xxiv, 755–56, 759
internationalization, 342–43
Internet domain names, 262–63, 267
Internet Explorer. See web browsers
interoperability, 513–17
API design, 515–17
compatibility, 513–15
with legacy code, 418–21, 420–21
interprocess communication (IPC) resources, 520
interrupt mechanism, 423–24
interrupt status, 525–25
interrupted method, 524–25
InterruptedException, 447, 523–26, 536–39, 545, 555
intValue method, 228, 273, 284
invokeClass method, 646–48
I/O, 339–422
atomic, 372
binary, 354–55
buffered, 345–46
channel, 390–92, 393–94
closing, 210–11, 341–42
from the command line, 352–57
command-line objects, 603
exceptions, 334–35, 370–71
interoperability, 418–19
line oriented, 344–45
memory mapped, 386
method chaining, 372, 802
NIO.2, 339–422
of objects, 357
of primitive data type values, 354–55
random access, 393–94
scanning and formatting, 246–352
streams, 339–40, 603. See streams, I/O
IOException, 312, 402
IOException, 313–23
is prefix, 758
isAfter method, 771
isAnnotation method, 211
isBefore method, 771
isEmpty method, 428–39, 434, 447–48, 451, 786
isInterface method, 211
isInterrupted method, 524
isLetter method, 289
isLowerCase method, 289
isLowerCase method, 289
ISO-86013. See International Organization for Standardization (ISO) calendar system
IsoField class, 775
isUpperCase method, 289
isWhitespace method, 289, 346
iterator method, 125, 244–45, 369, 397, 431, 434, 438, 441, 501
iterator object, 431
iterators, 428–34
aggregate operations vs., 474
fail-fast, 490
J
JApplet class, 653, 665–69, 671–73, 677, 678, 680–81, 698, 712
JApplet getCodeBase method, 678
JApplet getDocumentBase method, 678
JAR tool, 616, 618–23, 628, 630
setting entry points, 630
JarClassLoader class, 643–48
JarRunner example, 643–46
jarsigner tool, 722
JarURLConnection class, 642–45
Java 2D, 4
Java Application Programming Interface (API), 3–5, 34, 40–41, 793. See also APIs
(Application Programming Interfaces)
hierarchy of packages, 265
legacy, 224–25
raw types and, 224–25
runtime exceptions and, 312, 326, 330
Java Archive (JAR) files, 615–48. See also security
adding classes class path, 630–31
applets packaged in, 626
as applications, 626–27
benefits of, 615–16
creating, 616–20
extracting contents of, 622–23
manifest files, 627–35, 654, 674, 686, 724
paths in, 621
running JAR packaged software, 625–27
sealing, 634
signing, 639–41, 654, 722, 725–26
time stamping and, 654, 675, 725–26
uncompressed, 619
upgrading, 623–25
using, 616–27
using JAR-related APIs, 642–48
verifying, 635–42
viewing contents of, 620–22
Java Archive Tool. See JAR tool
Java Cache Viewer, 660–61
Java Collections Framework. See collections
Java Database Connectivity (JDBC) API, 4
Java HotSpot virtual machine, 2
Java Interactive Data Language (IDL) API, 4
java launcher tool, 2, 4, 626
Java Naming and Directory Interface (JNDI) API, 4
Java Network Launching Protocol. See JNLP files
Java platform, 2–4, 595–14
API specification. See Java Application Programming Interface (API)
command-line arguments, 600–601
configuration utilities, 595–603
environment variables, 601–3
language, 2–4
properties, 596–600, 604–7
supported encodings on. See Unicode encoding system utilities, 603–9
Java Plug-In software, 670, 693, 700–702, 734, 739, 749, 750
Java Programming Language Certification, 795–805
Java SE 8 Upgrade Exam, 801–5
Programmer Level I Exam, 795–801
Programmer Level II Exam, 801–5
Java Remote Invocation (RMI), 4
Java Remote Method Invocation over Internet Inter-ORB Protocol (Java RMI-IIOP), 4
Java SE Development Kit 8. See JDK 8 (Java SE Development Kit 8)
Java SE Runtime Environment. See JRE (Java SE Runtime Environment)
Java Virtual Machine (Java VM), 2–3, 268–70
Java Web Start applications, 650–56
changing the launch button of, 737
common problems, 662–63
deploying, 653–56, 735–38
developing without codebase attribute, 737–38
developing, 650–53
displaying customized loading progress indicator, 656–60
Java Cache Viewer, 660–61
retrieving resources, 653
running, 660–61
security and, 661–62
separating core functionality from final deployment mechanism, 652–53
setting up web servers for, 656
signed, 31, 654
JavaBeans, 5
javac compiler, 2, 4, 7
case sensitivity in, 12
javadoc tool, 24
JavaFX, xxiv, 4, 134, 793–94
HelloWorld.java example, 146
Scene Builder, 793
java.io package, 274, 339, 386, 389, 421
java.lang.Character API, 288
JavaScript
applets and, 670, 673–89
Deployment Toolkit scripts, 653, 673–74
interpreter, 670, 689, 731
java.time, 757. See Date-Time package
java.time.chrono package, 757
java.time.format package, 757
java.time.temporal package, 757–58, 774
java.time.zone package, 758
java.util.Arrays class, 56, 552
java.util.concurrent.atomic package, 553–54
java.util.concurrent.locks package, 544
java.util.function, 141
java.util.jar package, 642–43, 645
java.util.regex package, 551, 557–58, 581–88, 590, 592
javax.jnlp package, 650, 657, 716–17, 719
javax.swing.JApplet class, 666, 671–73, 677, 680, 712
JButton, 651, 672, 700
JDialog, 667
JDK 8 (Java SE Development Kit 8), xxiii–xxiv, 4, 6
adding to platform list, 9
aggregate operations. See aggregate operations concurrency in, 552
concurrent random numbers, 554–55
default manifest, 627
default methods. See default methods
JDK 8 (Java SE Development Kit 8) (continued)
directory structure, 609–13
generics and, 224
high-level concurrency objects and, 543
JAR tool in, 616
lambda expressions. See lambda expressions
local classes. See local classes
repeating annotations and, 165. See also
annotations; type annotations
target typing and, 236
ThreadLocalRandom, 543, 555
TransferQueue implementation, 498
JFrame class, 651
JNLP files, 739–48
API, 650, 706, 711, 714–17, 739
common errors, 662–63
commonly used elements and attributes, 741–47
deployment options with jnlp_href, 733
embedding in Applet tag, 734–35
encoding, 740
rich Internet applications (RIAs) and, 714–19
security and, 634
signed, 725
structure of, 740–47
join method, 525
JPanel class, 651–52, 671–72
JProgressBar object, 657, 658, 694, 695
JRE (Java SE Runtime Environment), 604
checking client version of, 738–39
ensuring the presence of, 751–53
JSR 310 Date and Time API. See Date-Time package
K
keys method, 599
keySet method, 453–56
keywords, 90. See also by individual type
L
lambda expressions, xxiii, 136–55
aggregate operations that accept, 145–46
as anonymous methods, 147
as arguments, 190
classification function, 479
errors, 149
example use, 137
generic types and, 141–42, 144–45
GUI applications and, 146–47
interference and, 484–85
in pipelines and streams, 478
shadowing and, 148
specifying search criteria code in, 141
stateful, 486–87
stream operations and, 486
syntax of, 147
using throughout an application, 142–44, 155–56
last method, 605, 646
lastIndexOf method, 296–99, 438–39
lastIndexOfSubList method, 446
laziness, 485–86
length method, 290, 302
line feed, 50, 344
line terminators, 344, 364, 567, 578
link awareness, 374
LinkedBlockingDeque class, 499
LinkedBlockingQueue class, 497
LinkedHashMap implementation, 450, 451, 491, 495
LinkedHashSet implementation, 433–34, 450, 492, 493
LinkedList implementation, 498
links
hard, 359, 399
symbolic, 361, 364, 367, 374–77, 379, 381, 399–405, 408, 419, 422
Linux. See Solaris/Linux
List interface, 468–46
algorithms, 445–46
collection operations, 375, 386, 394, 553
implementations, 493–94
iterators, 440–43
method, 513–14
positional access and search operations, 439–40
range view operations, 443–45
listIterator method, 438, 440–42, 512
ListOfNumbers example, 313–14, 320, 322, 323, 337
listRoots method, 421
lists, 138
cursor positions in, 441–42
iterating backward, 441
literals, 43, 48–51, 291, 557, 560–61, 566
classifier and string, 50
class, 50
floating point, 49
integer, 48–49
using underscore characters, 50–51
LiveConnect Specification, 683–84, 686
local classes, 127–31
specifying search criteria code in, 139–40
when to use, 155
LocalDate class, 763
Index

LocalDateTime class, 765–66
locales, 342, 347–48
LocalTime class, 764–65
lockInterruptibly method, 544
locks, 531–32, 544
deadlocks, 491, 533–35, 544
intrinsic, 531, 536
livelocks, 527, 533, 535
starvation, 527, 533, 535
synchronization and, 531–32
logarithms, 178, 279, 281, 286
logical operators, 58–68
Long class, 273, 286, 292–93, 459
long data type, 47
default value of, 48
longValue method, 273
lookingAt method, 584, 586–87
loops, 80–84, 798
infinite, 79–81, 86, 411
nested, 84
test harness, 559–60, 570
ls command, 22, 23, 238, 254, 407

M
main method, 24–26
manifest files, 616, 619, 627–35
default, 627
digest entries, 637–638
fetching attributes, 645
modifying, 628–629
setting application entry point, 629–630
setting package version information, 631–633
signature block files, 638, 641
signature files, 637–38
Map interface, 214, 449–58, 469, 482, 486, 491, 496, 504, 507, 803
basic operations, 451–52
bulk operations, 452
implementations of, 495–96
viewing as a Collection, 452–54
mapper element, 480
Matcher class, 557–58, 583, 585, 587–88
MatcherDemo example, 585–86
MatchesLooking example, 586–87
Math class, 266, 279, 281–82, 286
MAX_VALUE constant, 286, 404
members, 45
controlling access to, 110–11, 799
memory
allocating access to, 110–11, 799
allocating sufficient, 101
consistency errors, 528–29
error-handling, 331–32
garbage collection, 106–7
leaks, 5
locations, 12
saving in large arrays, 46–47
metadata, 378–86
method references, 152–55
to a constructor, 154–55
to an instance method of a particular object, 154
to an instance method of an arbitrary object of a particular type, 154
in pipelines and streams, 478
to a static method, 154
method signatures, 92
in interface declarations, 177–78
in method declarations, 92
overloaded methods and, 92
type erasure and, 248–49
methods, 34–36. See also by individual type
abstract, 184–86, 212–16
access modifiers and, 90, 95, 213, 798
accessor, 290, 295–96, 381–82, 467
applet milestone, 667–68
atomic, 496
bridge, 245–49
chaining, 372
class, 114, 118, 122, 273, 286, 292, 294
default, 182–92
defining, 92–94
final, 212
generic, 226–27
hiding, 199–203
instance, 199, 200–201
interface, 200–201
naming, 93, 758
overriding, 199–203
package-private, 111
qualified names, 119
returning a class or interface, 108–9
returning values from, 107–8
static, 186–87
synchronized, 529–31
wildcards, 240–43
Microsoft Windows
access control list (ACL), 380
CLASSPATH in, 612
common errors, 27–28
environment variables on, 602
file name separators on, 267
HelloWorld, 15–19
log files, 698
Microsoft Windows (continued)
PATH in, 362–70
path separators on, 296
root directories on, 359–60
system file stores, 417–18
MIME types, 380, 385, 417, 422, 656, 663
minus prefix, 758, 771
MIN_VALUE constant, 272, 286, 463
modifiers. See access modifiers
modularity, 36
monitor locks. See locks, intrinsic
Month enum, 760, 762
MonthDay class, 764
Mozilla Firefox add-ons, 732, 736
MultiDimArrayDemo example, 54, 55
multimaps, 456
multiple inheritance, 198–99
multisets, 510

N
NameSort example, 461
nanoTime method, 609
nCopies method, 502–3
NegativeArraySizeException, 326
nested classes, 121–57
controlling access and, 196
importing, 264–65
inheritance and, 195
inner. See inner classes
nonstatic, 121–22, 156
static, 122–23, 156
when to use, 156
NetBeans IDE, 1, 5–10, 14–15, 31, 410, 663, 796
HelloWorld application, 6–15
new keyword, 48, 101, 132, 222, 289
newCachedThreadPool method, 549
newFixedThreadPool method, 548
newline. See line terminators
newSingleThreadExecutor method, 549
next method, 431
nextIndex method, 441–42
NIO.2, 339–422
NonNull module, 171
NoSuchElementException, 446–47, 474
NoSuchMethodError, 30–31
Notepad demo, 656, 660, 736–38
Notepad text editor, 16–17
notifyAll method, 208, 536–38
now method, 758, 783
null parameter, 756–57
null value, 50, 342, 413
NullPointerException, 78, 171, 312, 330, 442,
459–60, 756, 800
Number class, 271, 273, 286, 601
number systems, 48–49
converting between, 273
decimal, 47, 49
hexadecimal, 49, 211, 273–74, 286, 351, 591
octal, 273–74
NumberFormatException, 526, 601
numbers, 271–87
converting between strings and, 292–95
formatting, 274–78
random, 283

O
Object class, 175, 208–16, 237, 243
as a superclass, 208–12
object ordering, 458–64
object references, 59, 87, 105–7, 113–14, 122,
209–10, 248, 342, 358, 688, 796, 799
ObjectInput interface, 357
ObjectInputStream class, 357
object-oriented programming, 33–41, 203
ObjectOutputStream class, 357
ObjectOutput interface, 357
ObjectOutputStream class, 357
objects, 34–36, 99–107, 118–19
calling methods, 105–6
casting, 197–98
creating, 100–104
declaring variables to refer to, 101
hash codes of, 208, 211, 460
immutable, 539–43
initializing, 102–4
lock, 544–46
retrieving fields, 104–5
ObjectStreams example, 357
octal number system, 273–74
of prefix, 758
offer method, 446
offerFirst method, 448–49
offerLast method, 448–49
OffsetDateTime class, 767, 769–70
operation element, 480
operators, 58–68. See also by individual type
assignment, 59
precedence of, 59, 69, 797
prefix/postfix, 59, 62
OutputStream class, 340–41, 343–45, 718
OutputStreamWriter class, 344, 718
@override annotation class, 168, 199

P
package members, 263
package importing, 264
Index

package referring to, 263–64
package using, 263–67
package-private, 111
package statements, 261–62, 264, 269, 679
packages, 4, 33, 34, 40, 259–70, 796
  apparent hierarchies of, 265
  creating, 261–62
  importing, 264–65
  name ambiguities, 265–66
  naming, 262–63
  qualified names, 262–67
  using package members, 263–67
pages. See web pages
Panel class, 651
parallelism, 480–81
parallelSort method, 57, 552
parameterized types, 224
  assigning raw types, 224–25
  backward compatibility and, 224
  bounded, 227–28
  casting, 254
  generic, 252
  heap pollution and, 250, 251
  primitive, 252–53
  restrictions, 252–56
  type erosion and, 244–45
  type inference and, 233–34
  varargs methods and, 249–50
parameters, 45
  naming, 97
  types, 96
parentheses. See ()
parse methods, 758, 772
parseXXX methods, 294, 601
PassPrimitiveByValue class, 98
Password example, 353–54
passwords, 353, 603, 639–40, 641
Path class, 362–70
PATH variable, 27, 29, 610–13
Pattern class, 557–59, 575, 578–79, 581. See also regular expressions
  PatternSyntaxException, 589–91
peek method, 447, 449
percent sign. See %
Period class, 780
Perl, 557, 558, 578, 591
permissions
  Permissions attribute, 31, 634, 724
  sandbox, 654, 674
  PI constant, 266
pipelines, 145, 472–74. See also streams
collections vs., 473
  components of, 473
downstream collectors, 479
  ordering, 483–84
  parallel execution of, 481
  terminal operations, 473, 475. See also reduction operations
Planet class, 159–60
pluggable type systems, xxiii, 170–71. See also type annotations
plus method, 758, 771
poll method, 410, 416, 446–47
polymorphism, 203–6, 245, 249, 799
 pound sign. See #
pow method, 95, 281
precision query, 778
Predicate interface, 141–42
Preferences API, 603
PrePostDemo example, 62, 68
primitive data types, 46–51. See also by individual type; numbers
print method, 349–50
printf method, 97, 274–76
println method, 349–50
PriorityBlockingQueue class, 497
PriorityQueue implementation, 490, 497
problems. See errors
ProcessBuilder object, 520, 602
processElements method, 145–46
processes, 520
  lightweight. See threads
Producer example, 538
ProducerConsumerExample example, 539
programs. See applications
properties
  managing, 597
  saving, 599–600
  setting, 599–600
  system, 605–7, 713–14
PropertiesTest example, 606
propertyName method, 599
protected modifier, 111
public modifier, 91, 111, 178, 185, 187
put method, 512
putAll method, 449, 452, 454
pwd command, 22, 30
Q
qualified names
  in applets, 659, 697
  for fields, 97
  for instance variables, 118
  for methods, 118
  for packages, 262–63
quantifiers, 573
question mark. See ?
Queue implementations, 427, 446–47, 490, 496–97
Queue interface, 446–48, 469, 491, 494, 497, 504
queues, 446–48
  bounded, 446
  priority, 427
QuoteClientApplet applet, 704–5
quoteReplacement method, 585
QuoteServer applet, 704–5
R
  radians, 282–83
random access files, 339, 390, 393–95
random method, 283
random numbers, 283, 286, 543, 554–55
RandomAccessFile class, 337, 420
raw types, 224–25, 234, 248, 249, 250
readDouble method, 355–56
readInt method, 355–56
readObject method, 357–59
readPassword method, 353–54
readUTF method, 355–56
Receiver applet, 702–3
  reduction operations, 474–80
  concurrent, 481–82
  multilevel reduction in streams, 479
  mutable, 484
RegexTestHarness example, 559, 579–80
RegexTestHarness2 example, 589–90
regionMatches method, 300–301
RegionMatchesDemo example, 300
regular expressions, 557–93
  backreferences in, 574–75
  boundary matchers in, 576–77
  capturing groups in, 572–76
  character classes in, 557–58, 562–67
  greedy quantifiers, 569, 573–74
  intersections, 558–65
  Matcher class, 583–89
  metacharacters in, 561–62
  negation, 563
  Pattern class, 578–83
  PatternSyntaxException class, 589–91
  possessive quantifiers, 573
  quantifiers in, 568–74
  ranges, 563–64
  reluctant quantifiers, 573
  string literals, 560–62
  subtraction, 565–66
  test harness, 559–60
  Unicode support, 591–92
  zero-length matches in, 569–72
Relatable interface, 178–80
relational operators. See comparisons
remove method, 429, 431, 434, 469, 502
removeAll method, 430, 432–37, 454–56, 503
removeDups method, 434
removeEldestEntry method, 495
removeFirst method, 448, 494, 498
removeFirstOccurrence method, 449
removeLast method, 449, 498
removeLastOccurrence method, 449 [*]
renameTo method, 420
@Repeatable annotation type, 170, 172
replace method, 496
replaceAll method, 298, 445, 584, 587–89
ReplaceDemo example, 587
ReplaceDemo2 example, 587–88
REPLACE_EXISTING enum, 376–78
replaceFirst method, 298, 585, 587–89
reserved words. See keywords
retainAll method, 436–37, 454, 455
@Retention annotation type, 169
return statements, 85, 107, 147, 151, 464
return types, 92–94
  constructors, 102
  covariant, 109, 199
rich Internet applications (RIAs), xxiv, 711–27
cookies and, 719–22
customizing the loading experience in, 722
ten entry points, 635
  local, 726
  security in, 722–26
  setting secure properties, 711–14
  setting trusted arguments, 711–14
  signing, 730–31
  system properties, 713–14
testing, 724
  user acceptance of, 729–31
  using the JNLP API, 714–19
Root example, 349–50
Root2 example, 350
run method, 522, 523
runtime, 2, 26
  checks at, 197
  errors, 29–31
  examining annotations at, 163
RuntimeException, 309, 312, 326, 329–30
S
  Safelock example, 544–46
  @SafeVarargs annotation type, 169
sandbox, 635, 661–62, 722–25, 729–31, 748
applets, 705–7, 713–14
permissions, 654, 674, 677, 691, 703
Scanner class, 346–48, 457
scanning, 345–46
ScanSum example, 348
ScanXan example, 346–47
ScheduledExecutorService interface, 546, 548, 549
ScheduledThreadPoolExecutor class, 549
security, xxiv
applets and, 596, 677–78
coding guidelines, 724
digitally signed files, 31, 637
JAR files, 31, 615, 635–42
Java Control Panel settings, 31, 730
Java versions and, 724, 730
Java Web Start applications and, 661–62
keystores and, 31, 639
legacy file I/O code and, 418–19
managers, 607–8, 707
manifest attributes and, 634–35
password entry and, 353
public and private keys, 636, 639
rich Internet applications (RIAs) and, 722–24
sandbox, 706–7
time stamping and, 654, 675, 725–26
TOCTTOU, 375
violations, 608
web browsers and, 697
SecurityException, 172, 607, 608
SecurityManager class, 607–8
semicolon. See ;
Sender applet, 702–3
sequences. See collections, ordered
Serializable interface, 214, 232, 357, 490
serialization, 152, 214
servers, 664, 678. See web servers
ServiceLoader class, 603
Set implementations, 492–93
Set interface, 433–37, 469, 504
array operations of, 437
basic operations of, 434–35
bulk operations of, 436–37
set method, 553, 757–58
setDefaultHostnameVerifier method, 662
setDefaultSSLSocketFactory method, 662
setLastModified method, 420–21
setLayout method, 379
setLength method, 302, 303
setProperties method, 605–7
setProperty method, 599–600
shadowing, 123–24
fields, 123–24
lambda expressions and, 148
local classes and, 129
Short class, 74, 273, 285, 293, 459
short data type, 46
default value of, 46
switch statement and, 74
shortValue method, 273, 286
ShowDocument applet, 682
showDocument method, 682–83, 706
showStatus method, 681
Shuffle example, 440, 444
signature block files, 638
signature files, 637–38
Signer Certificate Authority (CA) keystore, 31
Simple applet, 668
SimpleThreads example, 522, 525–26
single quote. See ’
singleton method, 432, 455, 489, 503
size method, 511–12
slash. See /
sleep method, 522–23
SleepMessages example, 523–24
Smalltalk’s collection hierarchy, 425
Socket class, 40
sockets, 40, 520
software. See applications
Solaris/Linux
HelloWorld, 20–23
paths in, 360–67
updating PATH variable, 611–12
sort method, 56, 189–90, 448, 506
SortedMap interface, 428, 467–69, 490, 495, 500,
502, 516
Sorted comparison to SortedSet, 468
Sorted map operations, 468
Sorted standard conversion constructor in, 468
SortedSet interface, 426, 428, 464–67, 468–69,
490, 492, 500
Sorted comparison to SortedMap, 469
Sorted endpoint operations in, 467
Sorted range-view operations, 465–67
Sorted set operations, 465
Sorted standard conversion constructors, 434
split method, 550, 581, 583
SplitDemo example, 581
SplitDemo2 example, 581–82
sqrt method, 97, 281, 349–50
square brackets. See [ ]
square root. See []
SSLsocketFactory class, 662
Index

Stack interface, 448
stack trace, 248, 312, 327–28, 335
StackOfInts class, 263
standard error, 352, 698
standard input, 352–53
start method, 586, 667, 691, 699, 709
startsWith method, 301, 369
statements, 70. See also by individual type
synchronized, 529, 531–532
static import statement, 266–67
static initialization blocks, 117
static keyword, 130–31, 186–87
static modifier, 44, 57, 112, 114–15
stop method, 667
Stream.collect method, 476–80
Stream.reduce method, 475–76
streams, 145, 473
parallel, 481–84
pipelines and, 472–74
streams, I/O, 339–59. See also by individual type
buffered, 345–46
byte, 340–42
close, 341–42
creating a file using, 389
data, 354–57
flushing, 346
object, 357–59
reading a file using, 389
unbuffered, 345
string builders, 302–6
StringBuilder class, 271, 302–7
StringDemo example, 290, 294, 305
StringIndexOutOfBoundsException, 299
StringBuilder method, 599
strings, 288–308
capacity of, 302
comparing portions of, 300, 301
concatenating, 291–92
converting between numbers and, 292–95
creating, 289
creating format strings, 292
length, 302
manipulating characters in, 295–300
replacing characters in, 296
searching within, 296
StringBuilder example, 77–78
subclasses, 38–39, 88–90, 111, 118, 194, 787
abstract methods and, 212–16
access levels and, 111
capabilities of, 196
constructors, 207–8
creating, 38–39
final methods and, 117–18
inheritance and, 38–39
polymorphism in, 203–4
returning, 108
subList method, 438, 443–46, 466
submit method, 547
subSequence method, 55, 297, 363, 575, 583–86
subSet method, 264, 436, 464, 466, 670, 759
substring method, 295–96, 299
subtyping, 203–31, 239, 249
super keyword, 202, 207, 238
superclasses, 33, 168, 175, 198–99, 209, 216, 334
accessing, 206–7
choosing, 329
constructors for, 95, 207
declaring, 89–90
inheritance and, 38–39, 193–94, 216
Object class, 208–12
private members in, 196–97
supplier argument, 477
@SuppressWarnings annotation type, 165, 167–68, 226, 252, 416
swap method, 439–40, 445, 508, 788
Swing, 4, 210, 794. See graphical user interfaces (GUIs)
switch block, 74–75
switch statements, 74–79
SwitchDemo example, 74
SwitchDemo2 example, 76–77
SwitchDemoFallThrough example, 75–76
symbolic links, 361–62, 364, 374–77, 379, 381, 399–405, 408, 419, 422
synchronization, 527–33
atomic access, 533
intrinsic locks and, 531
reentrant, 532
synchronized class example, 540–41
synchronized keyword, 529–30
synchronizedCollection method, 500–501
SynchronizedCounter example, 529–30, 554
synchronizedList method, 486–87, 494
synchronizedMap method, 500–501
SynchronizedRGB example, 540–42
synchronizedSet method, 500
synchronizedSortedMap method, 500
synchronizedSortedSet method, 500
SynchronousQueue class, 497
synthetic constructs, 124–25
System class, 26, 55–56, 597, 603–5
System.console, 353, 559, 590
System.err, 349, 352–53
System.in, 352–53
System.out, 274, 348, 352
T
tab, 289
tailSet method, 464, 467
@Target annotation type, 170
target typing, 150–51, 235–36
Temporal interface, 774–75
TemporalAccessor interface, 774–75
TemporalAdjuster interface, 776–80
  custom, 779–80
  predefined, 778–79
TemporalQuery method, 778–80
TemporalUnit interface, 774
ternary operators, 59, 64, 67, 797
test harness, 559–60
TestFormat example, 276–77
TextField class, 135
this keyword, 87, 98, 109–11, 799
Thread class, 522
thread pools, 411, 546–49
ThreadLocalRandom class, 543, 554–55
ThreadPoolExecutor class, 549
threads, 520–21
  in applets, 670
  contention, 481
defining, 521–22
  guarded blocks and, 535–39
  interference, 485–86, 527
  interrupts, 523–25
  joins, 525
  locks and, 534–35
  multithreaded applications, 272
  pausing, 522–23
  starting, 523–24
  synchronization of, 527–33
  thread objects, 521–27
  thread pools, 548–49
  thread safe, 306
throw statement, 324–25, 335
Throwable class, 255, 315, 325–26, 335
throws keyword, 324
TicTacToe example, 618–26
time. See Date-Time package
Time Zone Database (TZDB), 756
to prefix, 758
toArray method, 429–33, 465, 468, 497, 512,
  514–15
TOCTTOU, 375
toDegrees method, 282–83
tokens, 346–47
toLowerCase method, 77–78, 289, 297, 306
toRadians method, 282–83
toString method, 211–12
ToStringDemo example, 294–95
toupperCase method, 289, 297, 306
TransferQueue implementation, 498
TreeMap implementation, 450, 451, 468, 490–91,
  495, 553
TreeSet implementation, 433–35, 450, 463, 465,
  490–93
TrigonometricDemo example, 282
trigonometry, 272, 282–83, 286
trim method, 297, 470
troubleshooting. See errors
Trusted-Library attribute, 635
Trusted-Only attribute, 635
try blocks, 314–16, 318–22, 336, 370, 685, 800
tryLock method, 544–45
try-with-resources statement, 313, 317–20,
  370, 397
type annotations, xxiii, 165. See also annotations
  pluggable type systems and, 170–71
type erasure, 244–52, 256
  bridge methods and, 247–49
effects of, 247–49
type parameters, 98, 141, 221–30, 233–35, 237,
  245–46, 249, 253–57
type variables, 221–22, 228
types. See also by individual type
  multiple inheritance of, 198–99
  nonreifiable, 227–27
  parameterized. See parameterized types
  raw, 224–26
  supertypes with common ancestors, 201
type-checking, 170–71
TZDB. See Time Zone Database (TZDB)
U
unary operators, 59, 61–62, 66, 797
UnaryDemo example, 61
unboxing, 253, 271–72, 283–86, 288
underscore. See _
Unicode encoding
  character properties, 591–92
  regular expressions and, 591–92
  UTF-8, 356, 617, 628, 740
  UTF-16, 50, 740
UNIX. See Solaris/Linux
unset CLASSPATH, 30, 269
UnsupportedOperationException, 382, 385, 400, 426, 501–2
URLClassLoader implementation, 642–44, 646
useDelimiter method, 347
UTF. See Unicode encoding
V
valueOf method, 286, 293, 294
ValueOfDemo example, 287, 293–94
values method, 159
varargs, 96
   potential vulnerabilities of, 250–52
   preventing warnings from, 252
variables, 44–57. See also by individual type;
   fields
   atomic, 553–54
   class. See fields, static
   constant, 130–31
   environment, 601–2, 609–13
   instance. See instance variables
   local, 42, 44, 133–34, 148–49
   naming conventions, 45–46, 90, 92
   referring to objects, 101
Vector collection, 490, 494, 514
   compared to ArrayList, 494
vertical bar. See |
vi text editor, 20
volatile keyword, 533, 553
W
wait method, 208
warning messages, 225
   deprecation, 168
   security, 686
   suppressing, 686
   unchecked, 168
WarningDemo example, 225
watch keys, 413–14
WatchService API, 410–16
WeakHashMap implementation, 495–96
web browsers
   displaying documents in, 682–83
   frames in, 682–83
   security in, 677–78
web pages
   HTML frames, 682–83
   invoking applets, 674–76, 704–5
   Java applications, 654–56. See Java Web Start applications
web servers
   applets and, 31, 453–54
   JNLP errors in, 663
   placing applications on, 649, 656
   setting up, 656, 731–32
   testing, 31, 749–50
while statement, 79–80, 85–86
WhileDemo example, 79
white space
   allowing, 577–78
   character construct for, 567
   cleaning up, 749
   disallowed, 45
   leading, 576–77
   tokens and, 308, 347–48
   trailing, 296–97
wildcards, 236–44
   capture and, 240–43
   guidelines for using, 243–44
   helper methods and, 240–43
   lower-bounded, 238–39
   subtyping and, 239–40
   unbounded, 237–38
   upper-bounded, 236–37
Windows. See Microsoft Windows
with method, 758
wrappers, 344, 480, 491, 500–502
   checked interface, 502
   implementations of, 489–90, 499–502
   synchronization, 480–81, 489, 491, 500–501
   unmodifiable, 502
write method, 349, 386–88
writeDouble method, 355–56
writeInt method, 355–56
writeObject method, 358–59
writer method, 353
writeUTF method, 355–56
Y
Year class, 764
YearMonth class, 763–64
Z
ZIP archives, 616
ZonedDateTime object, 767–69
ZoneId class, 766–67
ZoneOffset class, 766–67
This page intentionally left blank