

CORE JAVA[®]

Volume I—Fundamentals

TENTH EDITION



CAY S. HORSTMANN

FREE SAMPLE CHAPTER



SHARE WITH OTHERS



Core Java®

Volume I—Fundamentals

Tenth Edition

This page intentionally left blank



Core Java®

Volume I—Fundamentals

Tenth Edition

Cay S. Horstmann



Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Names: Horstmann, Cay S., 1959- author.

Title: Core Java / Cay S. Horstmann.

Description: Tenth edition. | New York : Prentice Hall, [2016] | Includes index.

Identifiers: LCCN 2015038763 | ISBN 9780134177304 (volume 1 : pbk. : alk. paper) | ISBN 0134177304 (volume 1 : pbk. : alk. paper)

Subjects: LCSH: Java (Computer program language)

Classification: LCC QA76.73.J38 H6753 2016 | DDC 005.13/3—dc23

LC record available at <http://lccn.loc.gov/2015038763>

Copyright © 2016 Oracle and/or its affiliates. All rights reserved.
500 Oracle Parkway, Redwood Shores, CA 94065

Portions © Cay S. Horstmann

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

Oracle America Inc. does not make any representations or warranties as to the accuracy, adequacy or completeness of any information contained in this work, and is not responsible for any errors or omissions.

ISBN-13: 978-0-13-417730-4

ISBN-10: 0-13-417730-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing, December 2015

Contents

<i>Preface</i>	<i>xix</i>
<i>Acknowledgments</i>	<i>xxv</i>
Chapter 1: An Introduction to Java	1
1.1 Java as a Programming Platform	1
1.2 The Java “White Paper” Buzzwords	2
1.2.1 Simple	3
1.2.2 Object-Oriented	4
1.2.3 Distributed	4
1.2.4 Robust	4
1.2.5 Secure	4
1.2.6 Architecture-Neutral	5
1.2.7 Portable	6
1.2.8 Interpreted	7
1.2.9 High-Performance	7
1.2.10 Multithreaded	7
1.2.11 Dynamic	8
1.3 Java Applets and the Internet	8
1.4 A Short History of Java	10
1.5 Common Misconceptions about Java	13
Chapter 2: The Java Programming Environment	17
2.1 Installing the Java Development Kit	18
2.1.1 Downloading the JDK	18
2.1.2 Setting up the JDK	20
2.1.3 Installing Source Files and Documentation	22
2.2 Using the Command-Line Tools	23
2.3 Using an Integrated Development Environment	26
2.4 Running a Graphical Application	30
2.5 Building and Running Applets	33

Chapter 3: Fundamental Programming Structures in Java	41
3.1 A Simple Java Program	42
3.2 Comments	46
3.3 Data Types	47
3.3.1 Integer Types	47
3.3.2 Floating-Point Types	48
3.3.3 The <code>char</code> Type	50
3.3.4 Unicode and the <code>char</code> Type	51
3.3.5 The <code>boolean</code> Type	52
3.4 Variables	53
3.4.1 Initializing Variables	54
3.4.2 Constants	55
3.5 Operators	56
3.5.1 Mathematical Functions and Constants	57
3.5.2 Conversions between Numeric Types	59
3.5.3 Casts	60
3.5.4 Combining Assignment with Operators	61
3.5.5 Increment and Decrement Operators	61
3.5.6 Relational and <code>boolean</code> Operators	62
3.5.7 Bitwise Operators	63
3.5.8 Parentheses and Operator Hierarchy	64
3.5.9 Enumerated Types	65
3.6 Strings	65
3.6.1 Substrings	66
3.6.2 Concatenation	66
3.6.3 Strings Are Immutable	67
3.6.4 Testing Strings for Equality	68
3.6.5 Empty and Null Strings	69
3.6.6 Code Points and Code Units	70
3.6.7 The <code>String</code> API	71
3.6.8 Reading the Online API Documentation	74
3.6.9 Building Strings	77
3.7 Input and Output	78
3.7.1 Reading Input	79
3.7.2 Formatting Output	82

3.7.3	File Input and Output	87
3.8	Control Flow	89
3.8.1	Block Scope	89
3.8.2	Conditional Statements	90
3.8.3	Loops	94
3.8.4	Determinate Loops	99
3.8.5	Multiple Selections—The <code>switch</code> Statement	103
3.8.6	Statements That Break Control Flow	106
3.9	Big Numbers	108
3.10	Arrays	111
3.10.1	The “for each” Loop	113
3.10.2	Array Initializers and Anonymous Arrays	114
3.10.3	Array Copying	114
3.10.4	Command-Line Parameters	116
3.10.5	Array Sorting	117
3.10.6	Multidimensional Arrays	120
3.10.7	Ragged Arrays	124
Chapter 4: Objects and Classes		129
4.1	Introduction to Object-Oriented Programming	130
4.1.1	Classes	131
4.1.2	Objects	132
4.1.3	Identifying Classes	133
4.1.4	Relationships between Classes	133
4.2	Using Predefined Classes	135
4.2.1	Objects and Object Variables	136
4.2.2	The <code>LocalDate</code> Class of the Java Library	139
4.2.3	Mutator and Accessor Methods	141
4.3	Defining Your Own Classes	145
4.3.1	An <code>Employee</code> Class	145
4.3.2	Use of Multiple Source Files	149
4.3.3	Dissecting the <code>Employee</code> Class	149
4.3.4	First Steps with Constructors	150
4.3.5	Implicit and Explicit Parameters	152
4.3.6	Benefits of Encapsulation	153
4.3.7	Class-Based Access Privileges	156

4.3.8	Private Methods	156
4.3.9	Final Instance Fields	157
4.4	Static Fields and Methods	158
4.4.1	Static Fields	158
4.4.2	Static Constants	159
4.4.3	Static Methods	160
4.4.4	Factory Methods	161
4.4.5	The <code>main</code> Method	161
4.5	Method Parameters	164
4.6	Object Construction	171
4.6.1	Overloading	172
4.6.2	Default Field Initialization	172
4.6.3	The Constructor with No Arguments	173
4.6.4	Explicit Field Initialization	174
4.6.5	Parameter Names	175
4.6.6	Calling Another Constructor	176
4.6.7	Initialization Blocks	177
4.6.8	Object Destruction and the <code>finalize</code> Method	181
4.7	Packages	182
4.7.1	Class Importation	183
4.7.2	Static Imports	185
4.7.3	Addition of a Class into a Package	185
4.7.4	Package Scope	189
4.8	The Class Path	190
4.8.1	Setting the Class Path	193
4.9	Documentation Comments	194
4.9.1	Comment Insertion	194
4.9.2	Class Comments	195
4.9.3	Method Comments	195
4.9.4	Field Comments	196
4.9.5	General Comments	196
4.9.6	Package and Overview Comments	198
4.9.7	Comment Extraction	198
4.10	Class Design Hints	200

Chapter 5: Inheritance	203
5.1 Classes, Superclasses, and Subclasses	204
5.1.1 Defining Subclasses	204
5.1.2 Overriding Methods	206
5.1.3 Subclass Constructors	207
5.1.4 Inheritance Hierarchies	212
5.1.5 Polymorphism	213
5.1.6 Understanding Method Calls	214
5.1.7 Preventing Inheritance: Final Classes and Methods	217
5.1.8 Casting	219
5.1.9 Abstract Classes	221
5.1.10 Protected Access	227
5.2 Object: The Cosmic Superclass	228
5.2.1 The equals Method	229
5.2.2 Equality Testing and Inheritance	231
5.2.3 The hashCode Method	235
5.2.4 The toString Method	238
5.3 Generic Array Lists	244
5.3.1 Accessing Array List Elements	247
5.3.2 Compatibility between Typed and Raw Array Lists	251
5.4 Object Wrappers and Autoboxing	252
5.5 Methods with a Variable Number of Parameters	256
5.6 Enumeration Classes	258
5.7 Reflection	260
5.7.1 The Class Class	261
5.7.2 A Primer on Catching Exceptions	263
5.7.3 Using Reflection to Analyze the Capabilities of Classes	265
5.7.4 Using Reflection to Analyze Objects at Runtime	271
5.7.5 Using Reflection to Write Generic Array Code	276
5.7.6 Invoking Arbitrary Methods	279
5.8 Design Hints for Inheritance	283
Chapter 6: Interfaces, Lambda Expressions, and Inner Classes	287
6.1 Interfaces	288
6.1.1 The Interface Concept	288

6.1.2	Properties of Interfaces	295
6.1.3	Interfaces and Abstract Classes	297
6.1.4	Static Methods	298
6.1.5	Default Methods	298
6.1.6	Resolving Default Method Conflicts	300
6.2	Examples of Interfaces	302
6.2.1	Interfaces and Callbacks	302
6.2.2	The Comparator Interface	305
6.2.3	Object Cloning	306
6.3	Lambda Expressions	314
6.3.1	Why Lambdas?	314
6.3.2	The Syntax of Lambda Expressions	315
6.3.3	Functional Interfaces	318
6.3.4	Method References	319
6.3.5	Constructor References	321
6.3.6	Variable Scope	322
6.3.7	Processing Lambda Expressions	324
6.3.8	More about Comparators	328
6.4	Inner Classes	329
6.4.1	Use of an Inner Class to Access Object State	331
6.4.2	Special Syntax Rules for Inner Classes	334
6.4.3	Are Inner Classes Useful? Actually Necessary? Secure?	335
6.4.4	Local Inner Classes	339
6.4.5	Accessing Variables from Outer Methods	339
6.4.6	Anonymous Inner Classes	342
6.4.7	Static Inner Classes	346
6.5	Proxies	350
6.5.1	When to Use Proxies	350
6.5.2	Creating Proxy Objects	350
6.5.3	Properties of Proxy Classes	355
Chapter 7: Exceptions, Assertions, and Logging		357
7.1	Dealing with Errors	358
7.1.1	The Classification of Exceptions	359
7.1.2	Declaring Checked Exceptions	361
7.1.3	How to Throw an Exception	364

7.1.4	Creating Exception Classes	365
7.2	Catching Exceptions	367
7.2.1	Catching an Exception	367
7.2.2	Catching Multiple Exceptions	369
7.2.3	Rethrowing and Chaining Exceptions	370
7.2.4	The <code>finally</code> Clause	372
7.2.5	The Try-with-Resources Statement	376
7.2.6	Analyzing Stack Trace Elements	377
7.3	Tips for Using Exceptions	381
7.4	Using Assertions	384
7.4.1	The Assertion Concept	384
7.4.2	Assertion Enabling and Disabling	385
7.4.3	Using Assertions for Parameter Checking	386
7.4.4	Using Assertions for Documenting Assumptions	387
7.5	Logging	389
7.5.1	Basic Logging	389
7.5.2	Advanced Logging	390
7.5.3	Changing the Log Manager Configuration	392
7.5.4	Localization	393
7.5.5	Handlers	394
7.5.6	Filters	398
7.5.7	Formatters	399
7.5.8	A Logging Recipe	399
7.6	Debugging Tips	409
Chapter 8: Generic Programming		415
8.1	Why Generic Programming?	416
8.1.1	The Advantage of Type Parameters	416
8.1.2	Who Wants to Be a Generic Programmer?	417
8.2	Defining a Simple Generic Class	418
8.3	Generic Methods	421
8.4	Bounds for Type Variables	422
8.5	Generic Code and the Virtual Machine	425
8.5.1	Type Erasure	425
8.5.2	Translating Generic Expressions	426
8.5.3	Translating Generic Methods	427

8.5.4	Calling Legacy Code	429
8.6	Restrictions and Limitations	430
8.6.1	Type Parameters Cannot Be Instantiated with Primitive Types	430
8.6.2	Runtime Type Inquiry Only Works with Raw Types	431
8.6.3	You Cannot Create Arrays of Parameterized Types	431
8.6.4	Varargs Warnings	432
8.6.5	You Cannot Instantiate Type Variables	433
8.6.6	You Cannot Construct a Generic Array	434
8.6.7	Type Variables Are Not Valid in Static Contexts of Generic Classes	436
8.6.8	You Cannot Throw or Catch Instances of a Generic Class ...	436
8.6.9	You Can Defeat Checked Exception Checking	437
8.6.10	Beware of Clashes after Erasure	439
8.7	Inheritance Rules for Generic Types	440
8.8	Wildcard Types	442
8.8.1	The Wildcard Concept	442
8.8.2	Supertype Bounds for Wildcards	444
8.8.3	Unbounded Wildcards	447
8.8.4	Wildcard Capture	448
8.9	Reflection and Generics	450
8.9.1	The Generic Class Class	450
8.9.2	Using <code>Class<T></code> Parameters for Type Matching	452
8.9.3	Generic Type Information in the Virtual Machine	452
Chapter 9: Collections	459	
9.1	The Java Collections Framework	460
9.1.1	Separating Collection Interfaces and Implementation	460
9.1.2	The <code>Collection</code> Interface	463
9.1.3	Iterators	463
9.1.4	Generic Utility Methods	466
9.1.5	Interfaces in the Collections Framework	469
9.2	Concrete Collections	472
9.2.1	Linked Lists	474
9.2.2	Array Lists	484
9.2.3	Hash Sets	485

9.2.4	Tree Sets	489
9.2.5	Queues and Deques	494
9.2.6	Priority Queues	495
9.3	Maps	497
9.3.1	Basic Map Operations	497
9.3.2	Updating Map Entries	500
9.3.3	Map Views	502
9.3.4	Weak Hash Maps	504
9.3.5	Linked Hash Sets and Maps	504
9.3.6	Enumeration Sets and Maps	506
9.3.7	Identity Hash Maps	507
9.4	Views and Wrappers	509
9.4.1	Lightweight Collection Wrappers	509
9.4.2	Subranges	510
9.4.3	Unmodifiable Views	511
9.4.4	Synchronized Views	512
9.4.5	Checked Views	513
9.4.6	A Note on Optional Operations	514
9.5	Algorithms	517
9.5.1	Sorting and Shuffling	518
9.5.2	Binary Search	521
9.5.3	Simple Algorithms	522
9.5.4	Bulk Operations	524
9.5.5	Converting between Collections and Arrays	525
9.5.6	Writing Your Own Algorithms	526
9.6	Legacy Collections	528
9.6.1	The <code>Hashtable</code> Class	528
9.6.2	Enumerations	528
9.6.3	Property Maps	530
9.6.4	Stacks	531
9.6.5	Bit Sets	532
Chapter 10: Graphics Programming	537	
10.1	Introducing Swing	538
10.2	Creating a Frame	543
10.3	Positioning a Frame	546

10.3.1	Frame Properties	549
10.3.2	Determining a Good Frame Size	549
10.4	Displaying Information in a Component	554
10.5	Working with 2D Shapes	560
10.6	Using Color	569
10.7	Using Special Fonts for Text	573
10.8	Displaying Images	582
Chapter 11: Event Handling	587	
11.1	Basics of Event Handling	587
11.1.1	Example: Handling a Button Click	591
11.1.2	Specifying Listeners Concisely	595
11.1.3	Example: Changing the Look-and-Feel	598
11.1.4	Adapter Classes	603
11.2	Actions	607
11.3	Mouse Events	616
11.4	The AWT Event Hierarchy	624
11.4.1	Semantic and Low-Level Events	626
Chapter 12: User Interface Components with Swing	629	
12.1	Swing and the Model-View-Controller Design Pattern	630
12.1.1	Design Patterns	630
12.1.2	The Model-View-Controller Pattern	632
12.1.3	A Model-View-Controller Analysis of Swing Buttons	636
12.2	Introduction to Layout Management	638
12.2.1	Border Layout	641
12.2.2	Grid Layout	644
12.3	Text Input	648
12.3.1	Text Fields	649
12.3.2	Labels and Labeling Components	651
12.3.3	Password Fields	652
12.3.4	Text Areas	653
12.3.5	Scroll Panes	654
12.4	Choice Components	657
12.4.1	Checkboxes	657
12.4.2	Radio Buttons	660

12.4.3	Borders	664
12.4.4	Combo Boxes	668
12.4.5	Sliders	672
12.5	Menus	678
12.5.1	Menu Building	679
12.5.2	Icons in Menu Items	682
12.5.3	Checkbox and Radio Button Menu Items	683
12.5.4	Pop-Up Menus	684
12.5.5	Keyboard Mnemonics and Accelerators	686
12.5.6	Enabling and Disabling Menu Items	689
12.5.7	Toolbars	694
12.5.8	Tooltips	696
12.6	Sophisticated Layout Management	699
12.6.1	The Grid Bag Layout	701
12.6.1.1	The <code>gridx</code> , <code>gridy</code> , <code>gridwidth</code> , and <code>gridheight</code> Parameters ...	703
12.6.1.2	Weight Fields	703
12.6.1.3	The <code>fill</code> and <code>anchor</code> Parameters	704
12.6.1.4	Padding	704
12.6.1.5	Alternative Method to Specify the <code>gridx</code> , <code>gridy</code> , <code>gridwidth</code> , and <code>gridheight</code> Parameters	705
12.6.1.6	A Helper Class to Tame the Grid Bag Constraints	706
12.6.2	Group Layout	713
12.6.3	Using No Layout Manager	723
12.6.4	Custom Layout Managers	724
12.6.5	Traversal Order	729
12.7	Dialog Boxes	730
12.7.1	Option Dialogs	731
12.7.2	Creating Dialogs	741
12.7.3	Data Exchange	746
12.7.4	File Dialogs	752
12.7.5	Color Choosers	764
12.8	Troubleshooting GUI Programs	770
12.8.1	Debugging Tips	770
12.8.2	Letting the AWT Robot Do the Work	774

Chapter 13: Deploying Java Applications	779
13.1 JAR Files	780
13.1.1 Creating JAR files	780
13.1.2 The Manifest	781
13.1.3 Executable JAR Files	782
13.1.4 Resources	783
13.1.5 Sealing	787
13.2 Storage of Application Preferences	788
13.2.1 Property Maps	788
13.2.2 The Preferences API	794
13.3 Service Loaders	800
13.4 Applets	802
13.4.1 A Simple Applet	803
13.4.2 The <code>applet</code> HTML Tag and Its Attributes	808
13.4.3 Use of Parameters to Pass Information to Applets	810
13.4.4 Accessing Image and Audio Files	816
13.4.5 The Applet Context	818
13.4.6 Inter-Applet Communication	818
13.4.7 Displaying Items in the Browser	819
13.4.8 The Sandbox	820
13.4.9 Signed Code	822
13.5 Java Web Start	824
13.5.1 Delivering a Java Web Start Application	824
13.5.2 The JNLP API	829
Chapter 14: Concurrency	839
14.1 What Are Threads?	840
14.1.1 Using Threads to Give Other Tasks a Chance	846
14.2 Interrupting Threads	851
14.3 Thread States	855
14.3.1 New Threads	855
14.3.2 Runnable Threads	855
14.3.3 Blocked and Waiting Threads	856
14.3.4 Terminated Threads	857
14.4 Thread Properties	858
14.4.1 Thread Priorities	858

14.4.2	Daemon Threads	859
14.4.3	Handlers for Uncaught Exceptions	860
14.5	Synchronization	862
14.5.1	An Example of a Race Condition	862
14.5.2	The Race Condition Explained	866
14.5.3	Lock Objects	868
14.5.4	Condition Objects	872
14.5.5	The <code>synchronized</code> Keyword	878
14.5.6	Synchronized Blocks	882
14.5.7	The Monitor Concept	884
14.5.8	Volatile Fields	885
14.5.9	Final Variables	886
14.5.10	Atomics	886
14.5.11	Deadlocks	889
14.5.12	Thread-Local Variables	892
14.5.13	Lock Testing and Timeouts	893
14.5.14	Read/Write Locks	895
14.5.15	Why the <code>stop</code> and <code>suspend</code> Methods Are Deprecated	896
14.6	Blocking Queues	898
14.7	Thread-Safe Collections	905
14.7.1	Efficient Maps, Sets, and Queues	905
14.7.2	Atomic Update of Map Entries	907
14.7.3	Bulk Operations on Concurrent Hash Maps	909
14.7.4	Concurrent Set Views	912
14.7.5	Copy on Write Arrays	912
14.7.6	Parallel Array Algorithms	912
14.7.7	Older Thread-Safe Collections	914
14.8	Callables and Futures	915
14.9	Executors	920
14.9.1	Thread Pools	921
14.9.2	Scheduled Execution	926
14.9.3	Controlling Groups of Tasks	927
14.9.4	The Fork-Join Framework	928
14.9.5	CompletableFuture	931
14.10	Synchronizers	934

14.10.1	Semaphores	935
14.10.2	Countdown Latches	936
14.10.3	Barriers	936
14.10.4	Exchangers	937
14.10.5	Synchronous Queues	937
14.11	Threads and Swing	937
14.11.1	Running Time-Consuming Tasks	939
14.11.2	Using the Swing Worker	943
14.11.3	The Single-Thread Rule	951
<i>Appendix</i>	953
<i>Index</i>	957

Preface

To the Reader

In late 1995, the Java programming language burst onto the Internet scene and gained instant celebrity status. The promise of Java technology was that it would become the *universal glue* that connects users with information wherever it comes from—web servers, databases, information providers, or any other imaginable source. Indeed, Java is in a unique position to fulfill this promise. It is an extremely solidly engineered language that has gained wide acceptance. Its built-in security and safety features are reassuring both to programmers and to the users of Java programs. Java has built-in support for advanced programming tasks, such as network programming, database connectivity, and concurrency.

Since 1995, nine major revisions of the Java Development Kit have been released. Over the course of the last 20 years, the Application Programming Interface (API) has grown from about 200 to over 4,000 classes. The API now spans such diverse areas as user interface construction, database management, internationalization, security, and XML processing.

The book you have in your hands is the first volume of the tenth edition of *Core Java®*. Each edition closely followed a release of the Java Development Kit, and each time, we rewrote the book to take advantage of the newest Java features. This edition has been updated to reflect the features of Java Standard Edition (SE) 8.

As with the previous editions of this book, *we still target serious programmers who want to put Java to work on real projects*. We think of you, our reader, as a programmer with a solid background in a programming language other than Java, and we assume that you don't like books filled with toy examples (such as toasters, zoo animals, or "nervous text"). You won't find any of these in our book. Our goal is to enable you to fully understand the Java language and library, not to give you an illusion of understanding.

In this book you will find lots of sample code demonstrating almost every language and library feature that we discuss. We keep the sample programs purposefully simple to focus on the major points, but, for the most part, they aren't fake and they don't cut corners. They should make good starting points for your own code.

We assume you are willing, even eager, to learn about all the advanced features that Java puts at your disposal. For example, we give you a detailed treatment of

- Object-oriented programming
- Reflection and proxies
- Interfaces and inner classes
- Exception handling
- Generic programming
- The collections framework
- The event listener model
- Graphical user interface design with the Swing UI toolkit
- Concurrency

With the explosive growth of the Java class library, a one-volume treatment of all the features of Java that serious programmers need to know is no longer possible. Hence, we decided to break up the book into two volumes. The first volume, which you hold in your hands, concentrates on the fundamental concepts of the Java language, along with the basics of user-interface programming. The second volume, *Core Java®, Volume II—Advanced Features*, goes further into the enterprise features and advanced user-interface programming. It includes detailed discussions of

- The Stream API
- File processing and regular expressions
- Databases
- XML processing
- Annotations
- Internationalization
- Network programming
- Advanced GUI components
- Advanced graphics
- Native methods

When writing a book, errors and inaccuracies are inevitable. We'd very much like to know about them. But, of course, we'd prefer to learn about each of them only once. We have put up a list of frequently asked questions, bug fixes, and workarounds on a web page at <http://horstmann.com/corejava>. Strategically placed at the end of the errata page (to encourage you to read through it first) is a form you can use to report bugs and suggest improvements. Please don't be disappointed if we don't answer every query or don't get back to you immediately. We do read

all e-mail and appreciate your input to make future editions of this book clearer and more informative.

A Tour of This Book

Chapter 1 gives an overview of the capabilities of Java that set it apart from other programming languages. We explain what the designers of the language set out to do and to what extent they succeeded. Then, we give a short history of how Java came into being and how it has evolved.

In **Chapter 2**, we tell you how to download and install the JDK and the program examples for this book. Then we guide you through compiling and running three typical Java programs—a console application, a graphical application, and an applet—using the plain JDK, a Java-enabled text editor, and a Java IDE.

Chapter 3 starts the discussion of the Java language. In this chapter, we cover the basics: variables, loops, and simple functions. If you are a C or C++ programmer, this is smooth sailing because the syntax for these language features is essentially the same as in C. If you come from a non-C background such as Visual Basic, you will want to read this chapter carefully.

Object-oriented programming (OOP) is now in the mainstream of programming practice, and Java is an object-oriented programming language. **Chapter 4** introduces encapsulation, the first of two fundamental building blocks of object orientation, and the Java language mechanism to implement it—that is, classes and methods. In addition to the rules of the Java language, we also give advice on sound OOP design. Finally, we cover the marvelous javadoc tool that formats your code comments as a set of hyperlinked web pages. If you are familiar with C++, you can browse through this chapter quickly. Programmers coming from a non-object-oriented background should expect to spend some time mastering the OOP concepts before going further with Java.

Classes and encapsulation are only one part of the OOP story, and **Chapter 5** introduces the other—namely, *inheritance*. Inheritance lets you take an existing class and modify it according to your needs. This is a fundamental technique for programming in Java. The inheritance mechanism in Java is quite similar to that in C++. Once again, C++ programmers can focus on the differences between the languages.

Chapter 6 shows you how to use Java’s notion of an *interface*. Interfaces let you go beyond the simple inheritance model of Chapter 5. Mastering interfaces allows you to have full access to the power of Java’s completely object-oriented approach to programming. After we cover interfaces, we move on to *lambda expressions*, a

concise way for expressing a block of code that can be executed at a later point in time. We then cover a useful technical feature of Java called *inner classes*.

Chapter 7 discusses *exception handling*—Java’s robust mechanism to deal with the fact that bad things can happen to good programs. Exceptions give you an efficient way of separating the normal processing code from the error handling. Of course, even after hardening your program by handling all exceptional conditions, it still might fail to work as expected. In the final part of this chapter, we give you a number of useful debugging tips.

Chapter 8 gives an overview of generic programming. Generic programming makes your programs easier to read and safer. We show you how to use strong typing and remove unsightly and unsafe casts, and how to deal with the complexities that arise from the need to stay compatible with older versions of Java.

The topic of **Chapter 9** is the collections framework of the Java platform. Whenever you want to collect multiple objects and retrieve them later, you should use a collection that is best suited for your circumstances, instead of just tossing the elements into an array. This chapter shows you how to take advantage of the standard collections that are prebuilt for your use.

Chapter 10 starts the coverage of GUI programming. We show how you can make windows, how to paint on them, how to draw with geometric shapes, how to format text in multiple fonts, and how to display images.

Chapter 11 is a detailed discussion of the event model of the AWT, the *abstract window toolkit*. You’ll see how to write code that responds to events, such as mouse clicks or key presses. Along the way you’ll see how to handle basic GUI elements such as buttons and panels.

Chapter 12 discusses the Swing GUI toolkit in great detail. The Swing toolkit allows you to build cross-platform graphical user interfaces. You’ll learn all about the various kinds of buttons, text components, borders, sliders, list boxes, menus, and dialog boxes. However, some of the more advanced components are discussed in Volume II.

Chapter 13 shows you how to deploy your programs, either as applications or applets. We describe how to package programs in JAR files, and how to deliver applications over the Internet with the Java Web Start and applet mechanisms. We also explain how Java programs can store and retrieve configuration information once they have been deployed.

Chapter 14 finishes the book with a discussion of concurrency, which enables you to program tasks to be done in parallel. This is an important and exciting

application of Java technology in an era where most processors have multiple cores that you want to keep busy.

The Appendix lists the reserved words of the Java language.

Conventions

As is common in many computer books, we use monospace type to represent computer code.



NOTE: Notes are tagged with “note” icons that look like this.



TIP: Tips are tagged with “tip” icons that look like this.



CAUTION: When there is danger ahead, we warn you with a “caution” icon.



C++ NOTE: There are many C++ notes that explain the differences between Java and C++. You can skip over them if you don't have a background in C++ or if you consider your experience with that language a bad dream of which you'd rather not be reminded.

Java comes with a large programming library, or Application Programming Interface (API). When using an API call for the first time, we add a short summary description at the end of the section. These descriptions are a bit more informal but, we hope, also a little more informative than those in the official online API documentation. The names of interfaces are in italics, just like in the official documentation. The number after a class, interface, or method name is the JDK version in which the feature was introduced, as shown in the following example:

Application Programming Interface 1.2

Programs whose source code is on the book's companion web site are presented as listings, for instance:

Listing 1.1 InputTest/InputTest.java

Sample Code

The web site for this book at <http://horstmann.com/corejava> contains all sample code from the book, in compressed form. You can expand the file either with one of the familiar unzipping programs or simply with the jar utility that is part of the Java Development Kit. See Chapter 2 for more information on installing the Java Development Kit and the sample code.

Acknowledgments

Writing a book is always a monumental effort, and rewriting it doesn't seem to be much easier, especially with the continuous change in Java technology. Making a book a reality takes many dedicated people, and it is my great pleasure to acknowledge the contributions of the entire Core Java team.

A large number of individuals at Prentice Hall provided valuable assistance but managed to stay behind the scenes. I'd like them all to know how much I appreciate their efforts. As always, my warm thanks go to my editor, Greg Doench, for steering the book through the writing and production process, and for allowing me to be blissfully unaware of the existence of all those folks behind the scenes. I am very grateful to Julie Nahil for production support, and to Dmitry Kirsanov and Alina Kirsanova for copyediting and typesetting the manuscript. My thanks also to my coauthor of earlier editions, Gary Cornell, who has since moved on to other ventures.

Thanks to the many readers of earlier editions who reported embarrassing errors and made lots of thoughtful suggestions for improvement. I am particularly grateful to the excellent reviewing team who went over the manuscript with an amazing eye for detail and saved me from many embarrassing errors.

Reviewers of this and earlier editions include Chuck Allison (Utah Valley University), Lance Andersen (Oracle), Paul Anderson (Anderson Software Group), Alec Beaton (IBM), Cliff Berg, Andrew Binstock (Oracle), Joshua Bloch, David Brown, Corky Cartwright, Frank Cohen (PushToTest), Chris Crane (devXsolution), Dr. Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), David Geary (Clarity Training), Jim Gish (Oracle), Brian Goetz (Oracle), Angela Gordon, Dan Gordon (Electric Cloud), Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Marty Hall (coreservlets.com, Inc.), Vincent Hardy (Adobe Systems), Dan Harkey (San Jose State University), William Higgins (IBM), Vladimir Ivanovic (PointBase), Jerry Jackson (CA Technologies), Tim Kimmet (Walmart), Chris Laffra, Charlie Lai (Apple), Angelika Langer, Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University), Hao Pham, Paul Phlion, Blake Ragsdell, Stuart Reges (University of Arizona), Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nice, France), Dr. Paul Sanghera (San Jose State University and

Brooks College), Paul Sevinc (Teamup AG), Devang Shah (Sun Microsystems), Yoshiki Shibata, Bradley A. Smith, Steven Stelting (Oracle), Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (StreamingEdge), Janet Traub, Paul Tyma (consultant), Peter van der Linden, Christian Ullenboom, Burt Walsh, Dan Xu (Oracle), and John Zavgren (Oracle).

*Cay Horstmann
Biel/Bienne, Switzerland
November 2015*

Interfaces, Lambda Expressions, and Inner Classes

In this chapter

- 6.1 Interfaces, page 288
- 6.2 Examples of Interfaces, page 302
- 6.3 Lambda Expressions, page 314
- 6.4 Inner Classes, page 329
- 6.5 Proxies, page 350

You have now seen all the basic tools for object-oriented programming in Java. This chapter shows you several advanced techniques that are commonly used. Despite their less obvious nature, you will need to master them to complete your Java tool chest.

The first technique, called *interfaces*, is a way of describing *what* classes should do, without specifying *how* they should do it. A class can *implement* one or more interfaces. You can then use objects of these implementing classes whenever conformance to the interface is required. After we cover interfaces, we move on to *lambda expressions*, a concise way for expressing a block of code that can be

executed at a later point in time. Using lambda expressions, you can express code that uses callbacks or variable behavior in an elegant and concise fashion.

We then discuss the mechanism of *inner classes*. Inner classes are technically somewhat complex—they are defined inside other classes, and their methods can access the fields of the surrounding class. Inner classes are useful when you design collections of cooperating classes.

This chapter concludes with a discussion of *proxies*, objects that implement arbitrary interfaces. A proxy is a very specialized construct that is useful for building system-level tools. You can safely skip that section on first reading.

6.1 Interfaces

In the following sections, you will learn what Java interfaces are and how to use them. You will also find out how interfaces have been made more powerful in Java SE 8.

6.1.1 The Interface Concept

In the Java programming language, an interface is not a class but a set of *requirements* for the classes that want to conform to the interface.

Typically, the supplier of some service states: “If your class conforms to a particular interface, then I’ll perform the service.” Let’s look at a concrete example. The `sort` method of the `Arrays` class promises to sort an array of objects, but under one condition: The objects must belong to classes that implement the `Comparable` interface.

Here is what the `Comparable` interface looks like:

```
public interface Comparable
{
    int compareTo(Object other);
}
```

This means that any class that implements the `Comparable` interface is required to have a `compareTo` method, and the method must take an `Object` parameter and return an integer.



NOTE: As of Java SE 5.0, the Comparable interface has been enhanced to be a generic type.

```
public interface Comparable<T>
{
    int compareTo(T other); // parameter has type T
}
```

For example, a class that implements Comparable<Employee> must supply a method

```
int compareTo(Employee other)
```

You can still use the “raw” Comparable type without a type parameter. Then the compareTo method has a parameter of type Object, and you have to manually cast that parameter of the compareTo method to the desired type. We will do just that for a little while so that you don’t have to worry about two new concepts at the same time.

All methods of an interface are automatically public. For that reason, it is not necessary to supply the keyword public when declaring a method in an interface.

Of course, there is an additional requirement that the interface cannot spell out: When calling `x.compareTo(y)`, the compareTo method must actually be able to compare the two objects and return an indication whether `x` or `y` is larger. The method is supposed to return a negative number if `x` is smaller than `y`, zero if they are equal, and a positive number otherwise.

This particular interface has a single method. Some interfaces have multiple methods. As you will see later, interfaces can also define constants. What is more important, however, is what interfaces *cannot* supply. Interfaces never have instance fields. Before Java SE 8, methods were never implemented in interfaces. (As you will see in Section 6.1.4, “Static Methods,” on p. 298 and Section 6.1.5, “Default Methods,” on p. 298, it is now possible to supply simple methods in interfaces. Of course, those methods cannot refer to instance fields—interfaces don’t have any.)

Supplying instance fields and methods that operate on them is the job of the classes that implement the interface. You can think of an interface as being similar to an abstract class with no instance fields. However, there are some differences between these two concepts—we look at them later in some detail.

Now suppose we want to use the `sort` method of the `Arrays` class to sort an array of `Employee` objects. Then the `Employee` class must *implement* the `Comparable` interface.

To make a class implement an interface, you carry out two steps:

1. You declare that your class intends to implement the given interface.
2. You supply definitions for all methods in the interface.

To declare that a class implements an interface, use the `implements` keyword:

```
class Employee implements Comparable
```

Of course, now the `Employee` class needs to supply the `compareTo` method. Let's suppose that we want to compare employees by their salary. Here is an implementation of the `compareTo` method:

```
public int compareTo(Object otherObject)
{
    Employee other = (Employee) otherObject;
    return Double.compare(salary, other.salary);
}
```

Here, we use the static `Double.compare` method that returns a negative if the first argument is less than the second argument, 0 if they are equal, and a positive value otherwise.



CAUTION: In the interface declaration, the `compareTo` method was not declared `public` because all methods in an *interface* are automatically public. However, when implementing the interface, you must declare the method as `public`. Otherwise, the compiler assumes that the method has package visibility—the default for a *class*. The compiler then complains that you're trying to supply a more restrictive access privilege.

We can do a little better by supplying a type parameter for the generic `Comparable` interface:

```
class Employee implements Comparable<Employee>
{
    public int compareTo(Employee other)
    {
        return Double.compare(salary, other.salary);
    }
    ...
}
```

Note that the unsightly cast of the `Object` parameter has gone away.



TIP: The `compareTo` method of the `Comparable` interface returns an integer. If the objects are not equal, it does not matter what negative or positive value you return. This flexibility can be useful when you are comparing integer fields. For example, suppose each employee has a unique integer `id` and you want to sort by the employee ID number. Then you can simply return `id - other.id`. That value will be some negative value if the first ID number is less than the other, 0 if they are the same ID, and some positive value otherwise. However, there is one caveat: The range of the integers must be small enough so that the subtraction does not overflow. If you know that the IDs are not negative or that their absolute value is at most `(Integer.MAX_VALUE - 1) / 2`, you are safe. Otherwise, call the static `Integer.compare` method.

Of course, the subtraction trick doesn't work for floating-point numbers. The difference `salary - other.salary` can round to 0 if the salaries are close together but not identical. The call `Double.compare(x, y)` simply returns -1 if $x < y$ or 1 if $x > y$.



NOTE: The documentation of the `Comparable` interface suggests that the `compareTo` method should be compatible with the `equals` method. That is, `x.compareTo(y)` should be zero exactly when `x.equals(y)`. Most classes in the Java API that implement `Comparable` follow this advice. A notable exception is `BigDecimal`. Consider `x = new BigDecimal("1.0")` and `y = new BigDecimal("1.00")`. Then `x.equals(y)` is false because the numbers differ in precision. But `x.compareTo(y)` is zero. Ideally, it shouldn't be, but there was no obvious way of deciding which one should come first.

Now you saw what a class must do to avail itself of the sorting service—it must implement a `compareTo` method. That's eminently reasonable. There needs to be some way for the `sort` method to compare objects. But why can't the `Employee` class simply provide a `compareTo` method without implementing the `Comparable` interface?

The reason for interfaces is that the Java programming language is *strongly typed*. When making a method call, the compiler needs to be able to check that the method actually exists. Somewhere in the `sort` method will be statements like this:

```
if (a[i].compareTo(a[j]) > 0)
{
    // rearrange a[i] and a[j]
    ...
}
```

The compiler must know that `a[i]` actually has a `compareTo` method. If `a` is an array of `Comparable` objects, then the existence of the method is assured because every class that implements the `Comparable` interface must supply the method.



NOTE: You would expect that the `sort` method in the `Arrays` class is defined to accept a `Comparable[]` array so that the compiler can complain if anyone ever calls `sort` with an array whose element type doesn't implement the `Comparable` interface. Sadly, that is not the case. Instead, the `sort` method accepts an `Object[]` array and uses a clumsy cast:

```
// Approach used in the standard library--not recommended
if (((Comparable) a[i]).compareTo(a[j]) > 0)
{
    // rearrange a[i] and a[j]
    ...
}
```

If `a[i]` does not belong to a class that implements the `Comparable` interface, the virtual machine throws an exception.

Listing 6.1 presents the full code for sorting an array of instances of the class `Employee` (Listing 6.2) for sorting an employee array.

Listing 6.1 interfaces/EmployeeSortTest.java

```
1 package interfaces;
2
3 import java.util.*;
4
5 /**
6  * This program demonstrates the use of the Comparable interface.
7  * @version 1.30 2004-02-27
8  * @author Cay Horstmann
9 */
10 public class EmployeeSortTest
11 {
12     public static void main(String[] args)
13     {
14         Employee[] staff = new Employee[3];
15
16         staff[0] = new Employee("Harry Hacker", 35000);
17         staff[1] = new Employee("Carl Cracker", 75000);
18         staff[2] = new Employee("Tony Tester", 38000);
19
20         Arrays.sort(staff);
21
22         // print out information about all Employee objects
23         for (Employee e : staff)
24             System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
25     }
26 }
```

Listing 6.2 interfaces/Employee.java

```
1 package interfaces;
2
3 public class Employee implements Comparable<Employee>
4 {
5     private String name;
6     private double salary;
7
8     public Employee(String name, double salary)
9     {
10         this.name = name;
11         this.salary = salary;
12     }
13
14     public String getName()
15     {
16         return name;
17     }
18
19     public double getSalary()
20     {
21         return salary;
22     }
23
24     public void raiseSalary(double byPercent)
25     {
26         double raise = salary * byPercent / 100;
27         salary += raise;
28     }
29
30     /**
31      * Compares employees by salary
32      * @param other another Employee object
33      * @return a negative value if this employee has a lower salary than
34      * otherObject, 0 if the salaries are the same, a positive value otherwise
35      */
36     public int compareTo(Employee other)
37     {
38         return Double.compare(salary, other.salary);
39     }
40 }
```

java.lang.Comparable<T> 1.0

- `int compareTo(T other)`

compares this object with other and returns a negative integer if this object is less than other, zero if they are equal, and a positive integer otherwise.

java.util.Arrays 1.2

- static void sort(Object[] a)

sorts the elements in the array a. All elements in the array must belong to classes that implement the Comparable interface, and they must all be comparable to each other.

java.lang.Integer 1.0

- static int compare(int x, int y) **7**

returns a negative integer if $x < y$, zero if x and y are equal, and a positive integer otherwise.

java.lang.Double 1.0

- static int compare(double x, double y) **1.4**

returns a negative integer if $x < y$, zero if x and y are equal, and a positive integer otherwise.



NOTE: According to the language standard: “The implementor must ensure $\text{sgn}(x.\text{compareTo}(y)) = -\text{sgn}(y.\text{compareTo}(x))$ for all x and y . (This implies that $x.\text{compareTo}(y)$ must throw an exception if $y.\text{compareTo}(x)$ throws an exception.)” Here, sgn is the *sign* of a number: $\text{sgn}(n)$ is -1 if n is negative, 0 if n equals 0 , and 1 if n is positive. In plain English, if you flip the parameters of `compareTo`, the sign (but not necessarily the actual value) of the result must also flip.

As with the `equals` method, problems can arise when inheritance comes into play.

Since `Manager` extends `Employee`, it implements `Comparable<Employee>` and not `Comparable<Manager>`. If `Manager` chooses to override `compareTo`, it must be prepared to compare managers to employees. It can’t simply cast an employee to a manager:

```
class Manager extends Employee
{
    public int compareTo(Employee other)
    {
        Manager otherManager = (Manager) other; // NO
        ...
    }
    ...
}
```

That violates the “antisymmetry” rule. If `x` is an `Employee` and `y` is a `Manager`, then the call `x.compareTo(y)` doesn’t throw an exception—it simply compares `x` and `y` as employees. But the reverse, `y.compareTo(x)`, throws a `ClassCastException`.

This is the same situation as with the `equals` method that we discussed in Chapter 5, and the remedy is the same. There are two distinct scenarios.

If subclasses have different notions of comparison, then you should outlaw comparison of objects that belong to different classes. Each `compareTo` method should start out with the test

```
if (getClass() != other.getClass()) throw new ClassCastException();
```

If there is a common algorithm for comparing subclass objects, simply provide a single `compareTo` method in the superclass and declare it as `final`.

For example, suppose you want managers to be better than regular employees, regardless of salary. What about other subclasses such as `Executive` and `Secretary`? If you need to establish a pecking order, supply a method such as `rank` in the `Employee` class. Have each subclass override `rank`, and implement a single `compareTo` method that takes the `rank` values into account.

6.1.2 Properties of Interfaces

Interfaces are not classes. In particular, you can never use the `new` operator to instantiate an interface:

```
x = new Comparable(. . .); // ERROR
```

However, even though you can’t construct interface objects, you can still declare interface variables.

```
Comparable x; // OK
```

An interface variable must refer to an object of a class that implements the interface:

```
x = new Employee(. . .); // OK provided Employee implements Comparable
```

Next, just as you use `instanceof` to check whether an object is of a specific class, you can use `instanceof` to check whether an object implements an interface:

```
if (anObject instanceof Comparable) { . . . }
```

Just as you can build hierarchies of classes, you can extend interfaces. This allows for multiple chains of interfaces that go from a greater degree of generality to a greater degree of specialization. For example, suppose you had an interface called `Moveable`.

```
public interface Moveable
{
    void move(double x, double y);
}
```

Then, you could imagine an interface called `Powered` that extends it:

```
public interface Powered extends Moveable
{
    double milesPerGallon();
}
```

Although you cannot put instance fields or static methods in an interface, you can supply constants in them. For example:

```
public interface Powered extends Moveable
{
    double milesPerGallon();
    double SPEED_LIMIT = 95; // a public static final constant
}
```

Just as methods in an interface are automatically `public`, fields are always `public static final`.



NOTE: It is legal to tag interface methods as `public`, and fields as `public static final`. Some programmers do that, either out of habit or for greater clarity. However, the Java Language Specification recommends that the redundant keywords not be supplied, and we follow that recommendation.

Some interfaces define just constants and no methods. For example, the standard library contains an interface `SwingConstants` that defines constants `NORTH`, `SOUTH`, `HORIZONTAL`, and so on. Any class that chooses to implement the `SwingConstants` interface automatically inherits these constants. Its methods can simply refer to `NORTH` rather than the more cumbersome `SwingConstants.NORTH`. However, this use of interfaces seems rather degenerate, and we do not recommend it.

While each class can have only one superclass, classes can implement *multiple* interfaces. This gives you the maximum amount of flexibility in defining a class's behavior. For example, the Java programming language has an important interface built into it, called `Cloneable`. (We will discuss this interface in detail in Section 6.2.3, "Object Cloning," on p. 306.) If your class implements `Cloneable`, the `clone` method in the `Object` class will make an exact copy of your class's objects. If you want both cloneability and comparability, simply implement both interfaces. Use commas to separate the interfaces that you want to implement:

```
class Employee implements Cloneable, Comparable
```

6.1.3 Interfaces and Abstract Classes

If you read the section about abstract classes in Chapter 5, you may wonder why the designers of the Java programming language bothered with introducing the concept of interfaces. Why can't `Comparable` simply be an abstract class:

```
abstract class Comparable // why not?  
{  
    public abstract int compareTo(Object other);  
}
```

The `Employee` class would then simply extend this abstract class and supply the `compareTo` method:

```
class Employee extends Comparable // why not?  
{  
    public int compareTo(Object other) { . . . }  
}
```

There is, unfortunately, a major problem with using an abstract base class to express a generic property. A class can only extend a single class. Suppose the `Employee` class already extends a different class, say, `Person`. Then it can't extend a second class.

```
class Employee extends Person, Comparable // Error
```

But each class can implement as many interfaces as it likes:

```
class Employee extends Person implements Comparable // OK
```

Other programming languages, in particular C++, allow a class to have more than one superclass. This feature is called *multiple inheritance*. The designers of Java chose not to support multiple inheritance, because it makes the language either very complex (as in C++) or less efficient (as in Eiffel).

Instead, interfaces afford most of the benefits of multiple inheritance while avoiding the complexities and inefficiencies.



C++ NOTE: C++ has multiple inheritance and all the complications that come with it, such as virtual base classes, dominance rules, and transverse pointer casts. Few C++ programmers use multiple inheritance, and some say it should never be used. Other programmers recommend using multiple inheritance only for the “mix-in” style of inheritance. In the mix-in style, a primary base class describes the parent object, and additional base classes (the so-called mix-ins) may supply auxiliary characteristics. That style is similar to a Java class with a single superclass and additional interfaces.

6.1.4 Static Methods

As of Java SE 8, you are allowed to add static methods to interfaces. There was never a technical reason why this should be outlawed. It simply seemed to be against the spirit of interfaces as abstract specifications.

Up to now, it has been common to place static methods in companion classes. In the standard library, you find pairs of interfaces and utility classes such as `Collection`/`Collections` or `Path`/`Paths`.

Have a look at the `Paths` class. It only has a couple of factory methods. You can construct a path to a file or directory from a sequence of strings, such as `Paths.get("jdk1.8.0", "jre", "bin")`. In Java SE 8, one could have added this method to the `Path` interface:

```
public interface Path
{
    public static Path get(String first, String... more) {
        return FileSystems.getDefault().getPath(first, more);
    }
    ...
}
```

Then the `Paths` class is no longer necessary.

It is unlikely that the Java library will be refactored in this way, but when you implement your own interfaces, there is no longer a reason to provide a separate companion class for utility methods.

6.1.5 Default Methods

You can supply a *default* implementation for any interface method. You must tag such a method with the `default` modifier.

```
public interface Comparable<T>
{
    default int compareTo(T other) { return 0; }
        // By default, all elements are the same
}
```

Of course, that is not very useful since every realistic implementation of `Comparable` would override this method. But there are other situations where default methods can be useful. For example, as you will see in Chapter 11, if you want to be notified when a mouse click happens, you are supposed to implement an interface that has five methods:

```
public interface MouseListener
{
    void mouseClicked(MouseEvent event);
    void mousePressed(MouseEvent event);
    void mouseReleased(MouseEvent event);
    void mouseEntered(MouseEvent event);
    void mouseExited(MouseEvent event);
}
```

Most of the time, you only care about one or two of these event types. As of Java SE 8, you can declare all of the methods as default methods that do nothing.

```
public interface MouseListener
{
    default void mouseClicked(MouseEvent event) {}
    default void mousePressed(MouseEvent event) {}
    default void mouseReleased(MouseEvent event) {}
    default void mouseEntered(MouseEvent event) {}
    default void mouseExited(MouseEvent event) {}
}
```

Then programmers who implement this interface only need to override the listeners for the events they actually care about.

A default method can call other methods. For example, a `Collection` interface can define a convenience method

```
public interface Collection
{
    int size(); // An abstract method
    default boolean isEmpty()
    {
        return size() == 0;
    }
    ...
}
```

Then a programmer implementing `Collection` doesn't have to worry about implementing an `isEmpty` method.



NOTE: In the Java API, you will find a number of interfaces with companion classes that implement some or all of its methods, such as `Collection/AbstractCollection` or `MouseListener/MouseAdapter`. With Java SE 8, this technique is obsolete. Just implement the methods in the interface.

An important use for default methods is *interface evolution*. Consider for example the `Collection` interface that has been a part of Java for many years. Suppose that a long time ago, you provided a class

```
public class Bag implements Collection
```

Later, in Java SE 8, a `stream` method was added to the interface.

Suppose the `stream` method was not a default method. Then the `Bag` class no longer compiles since it doesn't implement the new method. Adding a nondefault method to an interface is not *source compatible*.

But suppose you don't recompile the class and simply use an old JAR file containing it. The class will still load, even with the missing method. Programs can still construct `Bag` instances, and nothing bad will happen. (Adding a method to an interface is *binary compatible*.) However, if a program calls the `stream` method on a `Bag` instance, an `AbstractMethodError` occurs.

Making the method a default method solves both problems. The `Bag` class will again compile. And if the class is loaded without being recompiled and the `stream` method is invoked on a `Bag` instance, the `Collection.stream` method is called.

6.1.6 Resolving Default Method Conflicts

What happens if the exact same method is defined as a default method in one interface and then again as a method of a superclass or another interface? Languages such as Scala and C++ have complex rules for resolving such ambiguities. Fortunately, the rules in Java are much simpler. Here they are:

1. Superclasses win. If a superclass provides a concrete method, default methods with the same name and parameter types are simply ignored.
2. Interfaces clash. If a superinterface provides a default method, and another interface supplies a method with the same name and parameter types (default or not), then you must resolve the conflict by overriding that method.

Let's look at the second rule. Consider another interface with a `getName` method:

```
interface Named
{
    default String getName() { return getClass().getName() + "_" + hashCode(); }
}
```

What happens if you form a class that implements both of them?

```
class Student implements Person, Named
{
    ...
}
```

The class inherits two inconsistent `getName` methods provided by the `Person` and `Named` interfaces. Instead of choosing one over the other, the Java compiler reports an

error and leaves it up to the programmer to resolve the ambiguity. Simply provide a `getName` method in the `Student` class. In that method, you can choose one of the two conflicting methods, like this:

```
class Student implements Person, Named
{
    public String getName() { return Person.super.getName(); }
    ...
}
```

Now assume that the `Named` interface does not provide a default implementation for `getName`:

```
interface Named
{
    String getName();
}
```

Can the `Student` class inherit the default method from the `Person` interface? This might be reasonable, but the Java designers decided in favor of uniformity. It doesn't matter how two interfaces conflict. If at least one interface provides an implementation, the compiler reports an error, and the programmer must resolve the ambiguity.



NOTE: Of course, if neither interface provides a default for a shared method, then we are in the situation before Java SE 8, and there is no conflict. An implementing class has two choices: implement the method, or leave it unimplemented. In the latter case, the class is itself abstract.

We just discussed name clashes between two interfaces. Now consider a class that extends a superclass and implements an interface, inheriting the same method from both. For example, suppose that `Person` is a class and `Student` is defined as

```
class Student extends Person implements Named { . . . }
```

In that case, only the superclass method matters, and any default method from the interface is simply ignored. In our example, `Student` inherits the `getName` method from `Person`, and it doesn't make any difference whether the `Named` interface provides a default for `getName` or not. This is the "class wins" rule.

The "class wins" rule ensures compatibility with Java SE 7. If you add default methods to an interface, it has no effect on code that worked before there were default methods.



CAUTION: You can never make a default method that redefines one of the methods in the `Object` class. For example, you can't define a default method for `toString` or `equals`, even though that might be attractive for interfaces such as `List`. As a consequence of the “classes win” rule, such a method could never win against `Object.toString` or `Objects.equals`.

6.2 Examples of Interfaces

In the next three sections, we give additional examples of interfaces so you can see how they are used in practice.

6.2.1 Interfaces and Callbacks

A common pattern in programming is the *callback* pattern. In this pattern, you specify the action that should occur whenever a particular event happens. For example, you may want a particular action to occur when a button is clicked or a menu item is selected. However, as you have not yet seen how to implement user interfaces, we will consider a similar but simpler situation.

The `javax.swing` package contains a `Timer` class that is useful if you want to be notified whenever a time interval has elapsed. For example, if a part of your program contains a clock, you can ask to be notified every second so that you can update the clock face.

When you construct a timer, you set the time interval and you tell it what it should do whenever the time interval has elapsed.

How do you tell the timer what it should do? In many programming languages, you supply the name of a function that the timer should call periodically. However, the classes in the Java standard library take an object-oriented approach. You pass an object of some class. The timer then calls one of the methods on that object. Passing an object is more flexible than passing a function because the object can carry additional information.

Of course, the timer needs to know what method to call. The timer requires that you specify an object of a class that implements the `ActionListener` interface of the `java.awt.event` package. Here is that interface:

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

The timer calls the `actionPerformed` method when the time interval has expired.

Suppose you want to print a message “At the tone, the time is . . .”, followed by a beep, once every 10 seconds. You would define a class that implements the `ActionListener` interface. You would then place whatever statements you want to have executed inside the `actionPerformed` method.

```
class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is " + new Date());
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Note the `ActionEvent` parameter of the `actionPerformed` method. This parameter gives information about the event, such as the source object that generated it—see Chapter 11 for more information. However, detailed information about the event is not important in this program, and you can safely ignore the parameter.

Next, you construct an object of this class and pass it to the `Timer` constructor.

```
ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener);
```

The first parameter of the `Timer` constructor is the time interval that must elapse between notifications, measured in milliseconds. We want to be notified every 10 seconds. The second parameter is the `listener` object.

Finally, you start the timer.

```
t.start();
```

Every 10 seconds, a message like

```
At the tone, the time is Wed Apr 13 23:29:08 PDT 2016
```

is displayed, followed by a beep.

Listing 6.3 puts the timer and its action listener to work. After the timer is started, the program puts up a message dialog and waits for the user to click the OK button to stop. While the program waits for the user, the current time is displayed at 10-second intervals.

Be patient when running the program. The “Quit program?” dialog box appears right away, but the first timer message is displayed after 10 seconds.

Note that the program imports the `javax.swing.Timer` class by name, in addition to importing `javax.swing.*` and `java.util.*`. This breaks the ambiguity between `javax.swing.Timer` and `java.util.Timer`, an unrelated class for scheduling background tasks.

Listing 6.3 timer/TimerTest.java

```
1 package timer;
2
3 /**
4  * @version 1.01 2015-05-12
5  * @author Cay Horstmann
6 */
7
8 import java.awt.*;
9 import java.awt.event.*;
10 import java.util.*;
11 import javax.swing.*;
12 import javax.swing.Timer;
13 // to resolve conflict with java.util.Timer
14
15 public class TimerTest
16 {
17     public static void main(String[] args)
18     {
19         ActionListener listener = new TimePrinter();
20
21         // construct a timer that calls the listener
22         // once every 10 seconds
23         Timer t = new Timer(10000, listener);
24         t.start();
25
26         JOptionPane.showMessageDialog(null, "Quit program?");
27         System.exit(0);
28     }
29 }
30
31 class TimePrinter implements ActionListener
32 {
33     public void actionPerformed(ActionEvent event)
34     {
35         System.out.println("At the tone, the time is " + new Date());
36         Toolkit.getDefaultToolkit().beep();
37     }
38 }
```

javax.swing.JOptionPane 1.2

- static void showMessageDialog(Component parent, Object message)

displays a dialog box with a message prompt and an OK button. The dialog is centered over the parent component. If parent is null, the dialog is centered on the screen.

javax.swing.Timer 1.2

- `Timer(int interval, ActionListener listener)`
constructs a timer that notifies `listener` whenever `interval` milliseconds have elapsed.
- `void start()`
starts the timer. Once started, the timer calls `actionPerformed` on its listeners.
- `void stop()`
stops the timer. Once stopped, the timer no longer calls `actionPerformed` on its listeners.

java.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`
gets the default toolkit. A toolkit contains information about the GUI environment.
- `void beep()`
emits a beep sound.

6.2.2 The Comparator Interface

In Section 6.1.1, “The Interface Concept,” on p. 288, you have seen how you can sort an array of objects, provided they are instances of classes that implement the `Comparable` interface. For example, you can sort an array of strings since the `String` class implements `Comparable<String>`, and the `String.compareTo` method compares strings in dictionary order.

Now suppose we want to sort strings by increasing length, not in dictionary order. We can’t have the `String` class implement the `compareTo` method in two ways—and at any rate, the `String` class isn’t ours to modify.

To deal with this situation, there is a second version of the `Arrays.sort` method whose parameters are an array and a *comparator*—an instance of a class that implements the `Comparator` interface.

```
public interface Comparator<T>
{
    int compare(T first, T second);
}
```

To compare strings by length, define a class that implements `Comparator<String>`:

```
class LengthComparator implements Comparator<String>
{
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
```

To actually do the comparison, you need to make an instance:

```
Comparator<String> comp = new LengthComparator();
if (comp.compare(words[i], words[j]) > 0) . . .
```

Contrast this call with `words[i].compareTo(words[j])`. The `compare` method is called on the comparator object, not the string itself.



NOTE: Even though the `LengthComparator` object has no state, you still need to make an instance of it. You need the instance to call the `compare` method—it is not a static method.

To sort an array, pass a `LengthComparator` object to the `Arrays.sort` method:

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends, new LengthComparator());
```

Now the array is either `["Paul", "Mary", "Peter"]` or `["Mary", "Paul", "Peter"]`.

You will see in Section 6.3, “Lambda Expressions,” on p. 314 how to use a `Comparator` much more easily with a lambda expression.

6.2.3 Object Cloning

In this section, we discuss the `Cloneable` interface that indicates that a class has provided a safe `clone` method. Since cloning is not all that common, and the details are quite technical, you may just want to glance at this material until you need it.

To understand what cloning means, recall what happens when you make a copy of a variable holding an object reference. The original and the copy are references to the same object (see Figure 6.1). This means a change to either variable also affects the other.

```
Employee original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10); // oops--also changed original
```

If you would like `copy` to be a new object that begins its life being identical to `original` but whose state can diverge over time, use the `clone` method.

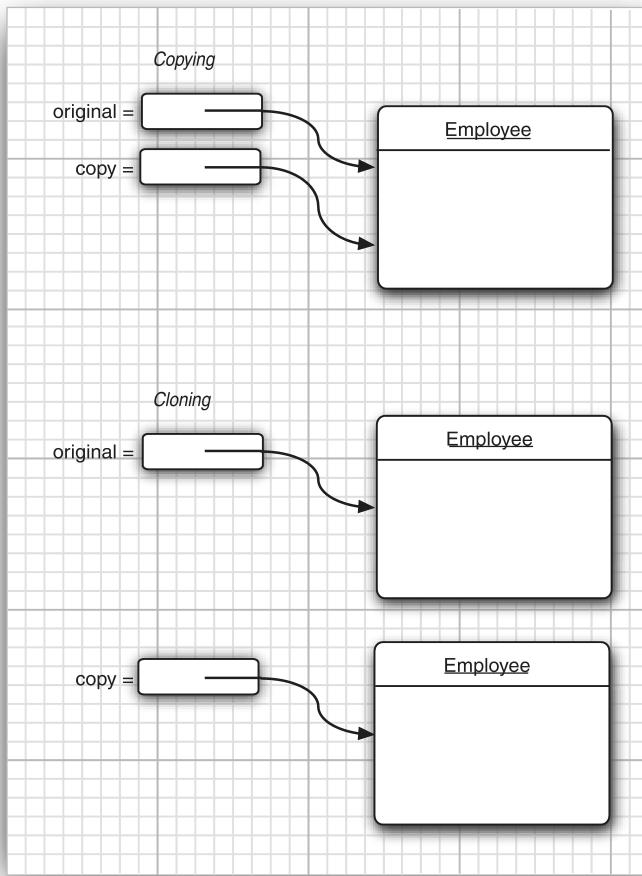


Figure 6.1 Copying and cloning

```
Employee copy = original.clone();
copy.raiseSalary(10); // OK--original unchanged
```

But it isn't quite so simple. The `clone` method is a protected method of `Object`, which means that your code cannot simply call it. Only the `Employee` class can clone `Employee` objects. There is a reason for this restriction. Think about the way in which the `Object` class can implement `clone`. It knows nothing about the object at all, so it can make only a field-by-field copy. If all data fields in the object are numbers or other basic types, copying the fields is just fine. But if the object contains references to subobjects, then copying the field gives you another reference to the same subobject, so the original and the cloned objects still share some information.

To visualize that, consider the `Employee` class that was introduced in Chapter 4. Figure 6.2 shows what happens when you use the `clone` method of the `Object` class to clone such an `Employee` object. As you can see, the default cloning operation is “shallow”—it doesn’t clone objects that are referenced inside other objects. (The figure shows a shared `Date` object. For reasons that will become clear shortly, this example uses a version of the `Employee` class in which the hire day is represented as a `Date`.)

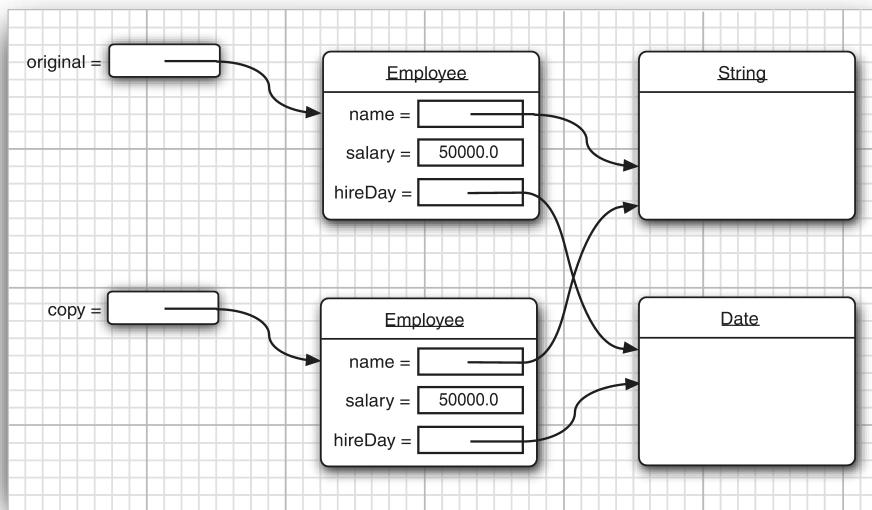


Figure 6.2 A shallow copy

Does it matter if the copy is shallow? It depends. If the subobject shared between the original and the shallow clone is *immutable*, then the sharing is safe. This certainly happens if the subobject belongs to an immutable class, such as `String`. Alternatively, the subobject may simply remain constant throughout the lifetime of the object, with no mutators touching it and no methods yielding a reference to it.

Quite frequently, however, subobjects are mutable, and you must redefine the `clone` method to make a *deep copy* that clones the subobjects as well. In our example, the `hireDay` field is a `Date`, which is mutable, so it too must be cloned. (For that reason, this example uses a field of type `Date`, not `LocalDate`, to demonstrate the cloning process. Had `hireDay` been an instance of the immutable `LocalDate` class, no further action would have been required.)

For every class, you need to decide whether

1. The default `clone` method is good enough;
2. The default `clone` method can be patched up by calling `clone` on the mutable subobjects; and
3. `clone` should not be attempted.

The third option is actually the default. To choose either the first or the second option, a class must

1. Implement the `Cloneable` interface; and
2. Redefine the `clone` method with the `public` access modifier.



NOTE: The `clone` method is declared `protected` in the `Object` class, so that your code can't simply call `anObject.clone()`. But aren't protected methods accessible from any subclass, and isn't every class a subclass of `Object`? Fortunately, the rules for protected access are more subtle (see Chapter 5). A subclass can call a protected `clone` method only to clone *its own* objects. You must redefine `clone` to be public to allow objects to be cloned by any method.

In this case, the appearance of the `Cloneable` interface has nothing to do with the normal use of interfaces. In particular, it does *not* specify the `clone` method—that method is inherited from the `Object` class. The interface merely serves as a tag, indicating that the class designer understands the cloning process. Objects are so paranoid about cloning that they generate a checked exception if an object requests cloning but does not implement that interface.

NOTE: The `Cloneable` interface is one of a handful of *tagging interfaces* that Java provides. (Some programmers call them *marker interfaces*.) Recall that the usual purpose of an interface such as `Comparable` is to ensure that a class implements a particular method or set of methods. A tagging interface has no methods; its only purpose is to allow the use of `instanceof` in a type inquiry:

```
if (obj instanceof Cloneable) . . .
```

We recommend that you do not use tagging interfaces in your own programs.

Even if the default (shallow copy) implementation of `clone` is adequate, you still need to implement the `Cloneable` interface, redefine `clone` to be public, and call `super.clone()`. Here is an example:

```
class Employee implements Cloneable
{
    // raise visibility level to public, change return type
    public Employee clone() throws CloneNotSupportedException
    {
        return (Employee) super.clone();
    }
    ...
}
```



NOTE: Up to Java SE 1.4, the `clone` method always had return type `Object`. Nowadays, you can specify the correct return type for your `clone` methods. This is an example of covariant return types (see Chapter 5).

The `clone` method that you just saw adds no functionality to the shallow copy provided by `Object.clone`. It merely makes the method public. To make a deep copy, you have to work harder and clone the mutable instance fields.

Here is an example of a `clone` method that creates a deep copy:

```
class Employee implements Cloneable
{
    ...
    public Employee clone() throws CloneNotSupportedException
    {
        // call Object.clone()
        Employee cloned = (Employee) super.clone();

        // clone mutable fields
        cloned.hireDay = (Date) hireDay.clone();

        return cloned;
    }
}
```

The `clone` method of the `Object` class threatens to throw a `CloneNotSupportedException`—it does that whenever `clone` is invoked on an object whose class does not implement the `Cloneable` interface. Of course, the `Employee` and `Date` classes implement the `Cloneable` interface, so the exception won't be thrown. However, the compiler does not know that. Therefore, we declared the exception:

```
public Employee clone() throws CloneNotSupportedException
```

Would it be better to catch the exception instead?

```
public Employee clone()
{
    try
    {
        Employee cloned = (Employee) super.clone();
        ...
    }
    catch (CloneNotSupportedException e) { return null; }
    // this won't happen, since we are Cloneable
}
```

This is appropriate for final classes. Otherwise, it is a good idea to leave the `throws` specifier in place. That gives subclasses the option of throwing a `CloneNotSupportedException` if they can't support cloning.

You have to be careful about cloning of subclasses. For example, once you have defined the `clone` method for the `Employee` class, anyone can use it to clone `Manager` objects. Can the `Employee` `clone` method do the job? It depends on the fields of the `Manager` class. In our case, there is no problem because the `bonus` field has primitive type. But `Manager` might have acquired fields that require a deep copy or are not cloneable. There is no guarantee that the implementor of the subclass has fixed `clone` to do the right thing. For that reason, the `clone` method is declared as `protected` in the `Object` class. But you don't have that luxury if you want users of your classes to invoke `clone`.

Should you implement `clone` in your own classes? If your clients need to make deep copies, then you probably should. Some authors feel that you should avoid `clone` altogether and instead implement another method for the same purpose. We agree that `clone` is rather awkward, but you'll run into the same issues if you shift the responsibility to another method. At any rate, cloning is less common than you may think. Less than 5 percent of the classes in the standard library implement `clone`.

The program in Listing 6.4 clones an instance of the class `Employee` (Listing 6.5), then invokes two mutators. The `raiseSalary` method changes the value of the `salary` field, whereas the `setHireDay` method changes the state of the `hireDay` field. Neither mutation affects the original object because `clone` has been defined to make a deep copy.



NOTE: All array types have a `clone` method that is public, not protected. You can use it to make a new array that contains copies of all elements. For example:

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
int[] cloned = luckyNumbers.clone();
cloned[5] = 12; // doesn't change luckyNumbers[5]
```



NOTE: Chapter 2 of Volume II shows an alternate mechanism for cloning objects, using the object serialization feature of Java. That mechanism is easy to implement and safe, but not very efficient.

Listing 6.4 clone/CloneTest.java

```
1 package clone;
2
3 /**
4  * This program demonstrates cloning.
5  * @version 1.10 2002-07-01
6  * @author Cay Horstmann
7 */
8 public class CloneTest
9 {
10    public static void main(String[] args)
11    {
12        try
13        {
14            Employee original = new Employee("John Q. Public", 50000);
15            original.setHireDay(2000, 1, 1);
16            Employee copy = original.clone();
17            copy.raiseSalary(10);
18            copy.setHireDay(2002, 12, 31);
19            System.out.println("original=" + original);
20            System.out.println("copy=" + copy);
21        }
22        catch (CloneNotSupportedException e)
23        {
24            e.printStackTrace();
25        }
26    }
27 }
```

Listing 6.5 clone/Employee.java

```
1 package clone;
2
3 import java.util.Date;
4 import java.util.GregorianCalendar;
5
6 public class Employee implements Cloneable
7 {
```

```
8  private String name;
9  private double salary;
10 private Date hireDay;
11
12 public Employee(String name, double salary)
13 {
14     this.name = name;
15     this.salary = salary;
16     hireDay = new Date();
17 }
18
19 public Employee clone() throws CloneNotSupportedException
20 {
21     // call Object.clone()
22     Employee cloned = (Employee) super.clone();
23
24     // clone mutable fields
25     cloned.hireDay = (Date) hireDay.clone();
26
27     return cloned;
28 }
29
30 /**
31 * Set the hire day to a given date.
32 * @param year the year of the hire day
33 * @param month the month of the hire day
34 * @param day the day of the hire day
35 */
36 public void setHireDay(int year, int month, int day)
37 {
38     Date newHireDay = new GregorianCalendar(year, month - 1, day).getTime();
39
40     // Example of instance field mutation
41     hireDay.setTime(newHireDay.getTime());
42 }
43
44 public void raiseSalary(double byPercent)
45 {
46     double raise = salary * byPercent / 100;
47     salary += raise;
48 }
49
50 public String toString()
51 {
52     return "Employee[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay + "]";
53 }
54 }
```

6.3 Lambda Expressions

Now you are ready to learn about lambda expressions, the most exciting change to the Java language in many years. You will see how to use lambda expressions for defining blocks of code with a concise syntax, and how to write code that consumes lambda expressions.

6.3.1 Why Lambdas?

A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple times. Before getting into the syntax (or even the curious name), let's step back and observe where we have used such code blocks in Java.

In Section 6.2.1, “Interfaces and Callbacks,” on p. 302, you saw how to do work in timed intervals. Put the work into the `actionPerformed` method of an `ActionListener`:

```
class Worker implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        // do some work
    }
}
```

Then, when you want to repeatedly execute this code, you construct an instance of the `Worker` class. You then submit the instance to a `Timer` object.

The key point is that the `actionPerformed` method contains code that you want to execute later.

Or consider sorting with a custom comparator. If you want to sort strings by length instead of the default dictionary order, you can pass a `Comparator` object to the `sort` method:

```
class LengthComparator implements Comparator<String>
{
    public int compare(String first, String second)
    {
        return first.length() - second.length();
    }
}
.
.
.
Arrays.sort(strings, new LengthComparator());
```

The `compare` method isn't called right away. Instead, the `sort` method keeps calling the `compare` method, rearranging the elements if they are out of order, until the array is sorted. You give the `sort` method a snippet of code needed to compare elements,

and that code is integrated into the rest of the sorting logic, which you'd probably not care to reimplement.

Both examples have something in common. A block of code was passed to someone—a timer, or a `sort` method. That code block was called at some later time.

Up to now, giving someone a block of code hasn't been easy in Java. You couldn't just pass code blocks around. Java is an object-oriented language, so you had to construct an object belonging to a class that has a method with the desired code.

In other languages, it is possible to work with blocks of code directly. The Java designers have resisted adding this feature for a long time. After all, a great strength of Java is its simplicity and consistency. A language can become an unmaintainable mess if it includes every feature that yields marginally more concise code. However, in those other languages it isn't just easier to spawn a thread or to register a button click handler; large swaths of their APIs are simpler, more consistent, and more powerful. In Java, one could have written similar APIs that take objects of classes implementing a particular function, but such APIs would be unpleasant to use.

For some time now, the question was not whether to augment Java for functional programming, but how to do it. It took several years of experimentation before a design emerged that is a good fit for Java. In the next section, you will see how you can work with blocks of code in Java SE 8.

6.3.2 The Syntax of Lambda Expressions

Consider again the sorting example from the preceding section. We pass code that checks whether one string is shorter than another. We compute

```
first.length() - second.length()
```

What are `first` and `second`? They are both strings. Java is a strongly typed language, and we must specify that as well:

```
(String first, String second)
    -> first.length() - second.length()
```

You have just seen your first *lambda expression*. Such an expression is simply a block of code, together with the specification of any variables that must be passed to the code.

Why the name? Many years ago, before there were any computers, the logician Alonzo Church wanted to formalize what it means for a mathematical function to be effectively computable. (Curiously, there are functions that are known to exist, but nobody knows how to compute their values.) He used the Greek letter

lambda (λ) to mark parameters. Had he known about the Java API, he would have written

```
 $\lambda$ first. $\lambda$ second.first.length() - second.length()
```



NOTE: Why the letter λ ? Did Church run out of other letters of the alphabet?

Actually, the venerable *Principia Mathematica* used the \wedge accent to denote free variables, which inspired Church to use an uppercase lambda Λ for parameters. But in the end, he switched to the lowercase version. Ever since, an expression with parameter variables has been called a lambda expression.

You have just seen one form of lambda expressions in Java: parameters, the \rightarrow arrow, and an expression. If the code carries out a computation that doesn't fit in a single expression, write it exactly like you would have written a method: enclosed in {} and with explicit return statements. For example,

```
(String first, String second) ->
{
    if (first.length() < second.length()) return -1;
    else if (first.length() > second.length()) return 1;
    else return 0;
}
```

If a lambda expression has no parameters, you still supply empty parentheses, just as with a parameterless method:

```
() -> { for (int i = 100; i >= 0; i--) System.out.println(i); }
```

If the parameter types of a lambda expression can be inferred, you can omit them. For example,

```
Comparator<String> comp
= (first, second) // Same as (String first, String second)
    -> first.length() - second.length();
```

Here, the compiler can deduce that first and second must be strings because the lambda expression is assigned to a string comparator. (We will have a closer look at this assignment in the next section.)

If a method has a single parameter with inferred type, you can even omit the parentheses:

```
ActionListener listener = event ->
    System.out.println("The time is " + new Date());
    // Instead of (event) -> . . . or (ActionEvent event) -> . . .
```

You never specify the result type of a lambda expression. It is always inferred from context. For example, the expression

(String first, String second) -> first.length() - second.length()
can be used in a context where a result of type int is expected.



NOTE: It is illegal for a lambda expression to return a value in some branches but not in others. For example, (int x) -> { if (x >= 0) return 1; } is invalid.

The program in Listing 6.6 shows how to use lambda expressions for a comparator and an action listener.

Listing 6.6 Lambda/LambdaTest.java

```
1 package Lambda;
2
3 import java.util.*;
4
5 import javax.swing.*;
6 import javax.swing.Timer;
7
8 /**
9  * This program demonstrates the use of lambda expressions.
10 * @version 1.0 2015-05-12
11 * @author Cay Horstmann
12 */
13 public class LambdaTest
14 {
15     public static void main(String[] args)
16     {
17         String[] planets = new String[] { "Mercury", "Venus", "Earth", "Mars",
18             "Jupiter", "Saturn", "Uranus", "Neptune" };
19         System.out.println(Arrays.toString(planets));
20         System.out.println("Sorted in dictionary order:");
21         Arrays.sort(planets);
22         System.out.println(Arrays.toString(planets));
23         System.out.println("Sorted by length:");
24         Arrays.sort(planets, (first, second) -> first.length() - second.length());
25         System.out.println(Arrays.toString(planets));
26
27         Timer t = new Timer(1000, event ->
28             System.out.println("The time is " + new Date()));
29         t.start();
30
31         // keep program running until user selects "Ok"
32         JOptionPane.showMessageDialog(null, "Quit program?");
33         System.exit(0);
34     }
35 }
```

6.3.3 Functional Interfaces

As we discussed, there are many existing interfaces in Java that encapsulate blocks of code, such as `ActionListener` or `Comparator`. Lambdas are compatible with these interfaces.

You can supply a lambda expression whenever an object of an interface with a single abstract method is expected. Such an interface is called a *functional interface*.



NOTE: You may wonder why a functional interface must have a single *abstract* method. Aren't all methods in an interface abstract? Actually, it has always been possible for an interface to redeclare methods from the `Object` class such as `toString` or `clone`, and these declarations do not make the methods abstract. (Some interfaces in the Java API redeclare `Object` methods in order to attach javadoc comments. Check out the `Comparator` API for an example.) More importantly, as you saw in Section 6.1.5, "Default Methods," on p. 298, in Java SE 8, interfaces can declare nonabstract methods.

To demonstrate the conversion to a functional interface, consider the `Arrays.sort` method. Its second parameter requires an instance of `Comparator`, an interface with a single method. Simply supply a lambda:

```
Arrays.sort(words,
            (first, second) -> first.length() - second.length());
```

Behind the scenes, the `Arrays.sort` method receives an object of some class that implements `Comparator<String>`. Invoking the `compare` method on that object executes the body of the lambda expression. The management of these objects and classes is completely implementation dependent, and it can be much more efficient than using traditional inner classes. It is best to think of a lambda expression as a function, not an object, and to accept that it can be passed to a functional interface.

This conversion to interfaces is what makes lambda expressions so compelling. The syntax is short and simple. Here is another example:

```
Timer t = new Timer(1000, event ->
{
    System.out.println("At the tone, the time is " + new Date());
    Toolkit.getDefaultToolkit().beep();
});
```

That's a lot easier to read than the alternative with a class that implements the `ActionListener` interface.

In fact, conversion to a functional interface is the *only* thing that you can do with a lambda expression in Java. In other programming languages that support function literals, you can declare function types such as `(String, String) -> int`, declare variables of those types, and use the variables to save function expressions. However, the Java designers decided to stick with the familiar concept of interfaces instead of adding function types to the language.



NOTE: You can't even assign a lambda expression to a variable of type `Object`—`Object` is not a functional interface.

The Java API defines a number of very generic functional interfaces in the `java.util.function` package. One of the interfaces, `BiFunction<T, U, R>`, describes functions with parameter types `T` and `U` and return type `R`. You can save our string comparison lambda in a variable of that type:

```
BiFunction<String, String, Integer> comp  
    = (first, second) -> first.length() - second.length();
```

However, that does not help you with sorting. There is no `Arrays.sort` method that wants a `BiFunction`. If you have used a functional programming language before, you may find this curious. But for Java programmers, it's pretty natural. An interface such as `Comparator` has a specific purpose, not just a method with given parameter and return types. Java SE 8 retains this flavor. When you want to do something with lambda expressions, you still want to keep the purpose of the expression in mind, and have a specific functional interface for it.

A particularly useful interface in the `java.util.function` package is `Predicate`:

```
public interface Predicate<T>  
{  
    boolean test(T t);  
    // Additional default and static methods  
}
```

The `ArrayList` class has a `removeIf` method whose parameter is a `Predicate`. It is specifically designed to pass a lambda expression. For example, the following statement removes all null values from an array list:

```
list.removeIf(e -> e == null);
```

6.3.4 Method References

Sometimes, there is already a method that carries out exactly the action that you'd like to pass on to some other code. For example, suppose you simply want to print the event object whenever a timer event occurs. Of course, you could call

```
Timer t = new Timer(1000, event -> System.out.println(event));
```

It would be nicer if you could just pass the `println` method to the `Timer` constructor. Here is how you do that:

```
Timer t = new Timer(1000, System.out::println);
```

The expression `System.out::println` is a *method reference* that is equivalent to the lambda expression `x -> System.out.println(x)`.

As another example, suppose you want to sort strings regardless of letter case. You can pass this method expression:

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

As you can see from these examples, the `::` operator separates the method name from the name of an object or class. There are three principal cases:

- `object::instanceMethod`
- `Class::staticMethod`
- `Class::instanceMethod`

In the first two cases, the method reference is equivalent to a lambda expression that supplies the parameters of the method. As already mentioned, `System.out::println` is equivalent to `x -> System.out.println(x)`. Similarly, `Math::pow` is equivalent to `(x, y) -> Math.pow(x, y)`.

In the third case, the first parameter becomes the target of the method. For example, `String::compareToIgnoreCase` is the same as `(x, y) -> x.compareToIgnoreCase(y)`.



NOTE: When there are multiple overloaded methods with the same name, the compiler will try to find from the context which one you mean. For example, there are two versions of the `Math.max` method, one for integers and one for double values. Which one gets picked depends on the method parameters of the functional interface to which `Math::max` is converted. Just like lambda expressions, method references don't live in isolation. They are always turned into instances of functional interfaces.

You can capture the `this` parameter in a method reference. For example, `this::equals` is the same as `x -> this.equals(x)`. It is also valid to use `super`. The method expression

```
super::instanceMethod
```

uses `this` as the target and invokes the superclass version of the given method. Here is an artificial example that shows the mechanics:

```
class Greeter
{
    public void greet()
    {
        System.out.println("Hello, world!");
    }
}

class TimedGreeter extends Greeter
{
    public void greet()
    {
        Timer t = new Timer(1000, super::greet);
        t.start();
    }
}
```

When the `TimedGreeter.greet` method starts, a `Timer` is constructed that executes the `super::greet` method on every timer tick. That method calls the `greet` method of the superclass.

6.3.5 Constructor References

Constructor references are just like method references, except that the name of the method is `new`. For example, `Person::new` is a reference to a `Person` constructor. Which constructor? It depends on the context. Suppose you have a list of strings. Then you can turn it into an array of `Person` objects, by calling the constructor on each of the strings, with the following invocation:

```
ArrayList<String> names = . . .;
Stream<Person> stream = names.stream().map(Person::new);
List<Person> people = stream.collect(Collectors.toList());
```

We will discuss the details of the `stream`, `map`, and `collect` methods in Chapter 1 of Volume II. For now, what's important is that the `map` method calls the `Person(String)` constructor for each list element. If there are multiple `Person` constructors, the compiler picks the one with a `String` parameter because it infers from the context that the constructor is called with a string.

You can form constructor references with array types. For example, `int[]::new` is a constructor reference with one parameter: the length of the array. It is equivalent to the lambda expression `x -> new int[x]`.

Array constructor references are useful to overcome a limitation of Java. It is not possible to construct an array of a generic type `T`. The expression `new T[n]` is an error since it would be erased to `new Object[n]`. That is a problem for library authors. For example, suppose we want to have an array of `Person` objects. The `Stream` interface has a `toArray` method that returns an `Object` array:

```
Object[] people = stream.toArray();
```

But that is unsatisfactory. The user wants an array of references to `Person`, not references to `Object`. The stream library solves that problem with constructor references. Pass `Person[]::new` to the `toArray` method:

```
Person[] people = stream.toArray(Person[]::new);
```

The `toArray` method invokes this constructor to obtain an array of the correct type. Then it fills and returns the array.

6.3.6 Variable Scope

Often, you want to be able to access variables from an enclosing method or class in a lambda expression. Consider this example:

```
public static void repeatMessage(String text, int delay)
{
    ActionListener listener = event ->
    {
        System.out.println(text);
        Toolkit.getDefaultToolkit().beep();
    };
    new Timer(delay, listener).start();
}
```

Consider a call

```
repeatMessage("Hello", 1000); // Prints Hello every 1,000 milliseconds
```

Now look at the variable `text` inside the lambda expression. Note that this variable is *not* defined in the lambda expression. Instead, it is a parameter variable of the `repeatMessage` method.

If you think about it, something nonobvious is going on here. The code of the lambda expression may run long after the call to `repeatMessage` has returned and the parameter variables are gone. How does the `text` variable stay around?

To understand what is happening, we need to refine our understanding of a lambda expression. A lambda expression has three ingredients:

1. A block of code
2. Parameters
3. Values for the *free* variables, that is, the variables that are not parameters and not defined inside the code

In our example, the lambda expression has one free variable, `text`. The data structure representing the lambda expression must store the values for the free

variables, in our case, the string "Hello". We say that such values have been *captured* by the lambda expression. (It's an implementation detail how that is done. For example, one can translate a lambda expression into an object with a single method, so that the values of the free variables are copied into instance variables of that object.)



NOTE: The technical term for a block of code together with the values of the free variables is a *closure*. If someone gloats that their language has closures, rest assured that Java has them as well. In Java, lambda expressions are closures.

As you have seen, a lambda expression can capture the value of a variable in the enclosing scope. In Java, to ensure that the captured value is well-defined, there is an important restriction. In a lambda expression, you can only reference variables whose value doesn't change. For example, the following is illegal:

```
public static void countDown(int start, int delay)
{
    ActionListener listener = event ->
    {
        start--; // Error: Can't mutate captured variable
        System.out.println(start);
    };
    new Timer(delay, listener).start();
}
```

There is a reason for this restriction. Mutating variables in a lambda expression is not safe when multiple actions are executed concurrently. This won't happen for the kinds of actions that we have seen so far, but in general, it is a serious problem. See Chapter 14 for more information on this important issue.

It is also illegal to refer to variable in a lambda expression that is mutated outside. For example, the following is illegal:

```
public static void repeat(String text, int count)
{
    for (int i = 1; i <= count; i++)
    {
        ActionListener listener = event ->
        {
            System.out.println(i + ": " + text);
            // Error: Cannot refer to changing i
        };
        new Timer(1000, listener).start();
    }
}
```

The rule is that any captured variable in a lambda expression must be *effectively final*. An effectively final variable is a variable that is never assigned a new value after it has been initialized. In our case, `text` always refers to the same `String` object, and it is OK to capture it. However, the value of `i` is mutated, and therefore `i` cannot be captured.

The body of a lambda expression has *the same scope as a nested block*. The same rules for name conflicts and shadowing apply. It is illegal to declare a parameter or a local variable in the lambda that has the same name as a local variable.

```
Path first = Paths.get("/usr/bin");
Comparator<String> comp =
    (first, second) -> first.length() - second.length();
// Error: Variable first already defined
```

Inside a method, you can't have two local variables with the same name, and therefore, you can't introduce such variables in a lambda expression either.

When you use the `this` keyword in a lambda expression, you refer to the `this` parameter of the method that creates the lambda. For example, consider

```
public class Application()
{
    public void init()
    {
        ActionListener listener = event ->
        {
            System.out.println(this.toString());
            ...
        }
        ...
    }
}
```

The expression `this.toString()` calls the `toString` method of the `Application` object, *not* the `ActionListener` instance. There is nothing special about the use of `this` in a lambda expression. The scope of the lambda expression is nested inside the `init` method, and `this` has the same meaning anywhere in that method.

6.3.7 Processing Lambda Expressions

Up to now, you have seen how to produce lambda expressions and pass them to a method that expects a functional interface. Now let us see how to write methods that can consume lambda expressions.

The point of using lambdas is *deferred execution*. After all, if you wanted to execute some code right now, you'd do that, without wrapping it inside a lambda. There are many reasons for executing code later, such as:

- Running the code in a separate thread
- Running the code multiple times
- Running the code at the right point in an algorithm (for example, the comparison operation in sorting)
- Running the code when something happens (a button was clicked, data has arrived, and so on)
- Running the code only when necessary

Let's look at a simple example. Suppose you want to repeat an action n times. The action and the count are passed to a `repeat` method:

```
repeat(10, () -> System.out.println("Hello, World!"));
```

To accept the lambda, we need to pick (or, in rare cases, provide) a functional interface. Table 6.1 lists the most important functional interfaces that are provided in the Java API. In this case, we can use the `Runnable` interface:

```
public static void repeat(int n, Runnable action)
{
    for (int i = 0; i < n; i++) action.run();
}
```

Note that the body of the lambda expression is executed when `action.run()` is called.

Now let's make this example a bit more sophisticated. We want to tell the action in which iteration it occurs. For that, we need to pick a functional interface that has a method with an `int` parameter and a `void` return. The standard interface for processing `int` values is

```
public interface IntConsumer
{
    void accept(int value);
}
```

Here is the improved version of the `repeat` method:

```
public static void repeat(int n, IntConsumer action)
{
    for (int i = 0; i < n; i++) action.accept(i);
}
```

And here is how you call it:

```
repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
```

Table 6.1 Common Functional Interfaces

Functional Interface	Parameter Types	Return Type	Abstract Method Name	Description	Other Methods
Runnable	none	void	run	Runs an action without arguments or return value	
Supplier<T>	none	T	get	Supplies a value of type T	
Consumer<T>	T	void	accept	Consumes a value of type T	andThen
BiConsumer<T, U>	T, U	void	accept	Consumes values of types T and U	andThen
Function<T, R>	T	R	apply	A function with argument of type T	compose, andThen, identity
BiFunction<T, U, R>	T, U	R	apply	A function with arguments of types T and U	andThen
UnaryOperator<T>	T	T	apply	A unary operator on the type T	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	A binary operator on the type T	andThen, maxBy, minBy
Predicate<T>	T	boolean	test	A boolean-valued function	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	A boolean-valued function with two arguments	and, or, negate

Table 6.2 lists the 34 available specializations for primitive types `int`, `long`, and `double`. It is a good idea to use these specializations to reduce autoboxing. For that reason, I used an `IntConsumer` instead of a `Consumer<Integer>` in the example of the preceding section.

Table 6.2 Functional Interfaces for Primitive Types

p, q is `int`, `long`, `double`; P, Q is `Int`, `Long`, `Double`

Functional Interface	Parameter Types	Return Type	Abstract Method Name
<code>BooleanSupplier</code>	none	<code>boolean</code>	<code>getAsBoolean</code>
<code>PSupplier</code>	none	p	<code>getAsP</code>
<code>PConsumer</code>	p	<code>void</code>	<code>accept</code>
<code>ObjPConsumer<T></code>	T, p	<code>void</code>	<code>accept</code>
<code>PFunction<T></code>	p	T	<code>apply</code>
<code>PToQFunction</code>	p	q	<code>applyAsQ</code>
<code>ToPFunction<T></code>	T	p	<code>applyAsP</code>
<code>ToPBiFunction<T, U></code>	T, U	p	<code>applyAsP</code>
<code>PUnaryOperator</code>	p	p	<code>applyAsP</code>
<code>PBinaryOperator</code>	p, p	p	<code>applyAsP</code>
<code>PPredicate</code>	p	<code>boolean</code>	<code>test</code>



TIP: It is a good idea to use an interface from Tables 6.1 or 6.2 whenever you can. For example, suppose you write a method to process files that match a certain criterion. There is a legacy interface `java.io.FileFilter`, but it is better to use the standard `Predicate<File>`. The only reason not to do so would be if you already have many useful methods producing `FileFilter` instances.



NOTE: Most of the standard functional interfaces have nonabstract methods for producing or combining functions. For example, `Predicate isEqual(a)` is the same as `a::equals`, but it also works if `a` is `null`. There are `default` methods `and`, `or`, `negate` for combining predicates. For example, `Predicate isEqual(a).or(Predicate isEqual(b))` is the same as `x -> a.equals(x) || b.equals(x)`.



NOTE: If you design your own interface with a single abstract method, you can tag it with the `@FunctionalInterface` annotation. This has two advantages. The compiler gives an error message if you accidentally add another nonabstract method. And the javadoc page includes a statement that your interface is a functional interface.

It is not required to use the annotation. Any interface with a single abstract method is, by definition, a functional interface. But using the `@FunctionalInterface` annotation is a good idea.

6.3.8 More about Comparators

The `Comparator` interface has a number of convenient static methods for creating comparators. These methods are intended to be used with lambda expressions or method references.

The static `comparing` method takes a “key extractor” function that maps a type `T` to a comparable type (such as `String`). The function is applied to the objects to be compared, and the comparison is then made on the returned keys. For example, suppose you have an array of `Person` objects. Here is how you can sort them by name:

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

This is certainly much easier than implementing a `Comparator` by hand. Moreover, the code is clearer since it is obvious that we want to compare people by name.

You can chain comparators with the `thenComparing` method for breaking ties. For example,

```
Arrays.sort(people,
    Comparator.comparing(Person::getLastName)
        .thenComparing(Person::getFirstName));
```

If two people have the same last name, then the second comparator is used.

There are a few variations of these methods. You can specify a comparator to be used for the keys that the `comparing` and `thenComparing` methods extract. For example, here we sort people by the length of their names:

```
Arrays.sort(people, Comparator.comparing(Person::getName,
    (s, t) -> Integer.compare(s.length(), t.length())));
```

Moreover, both the `comparing` and `thenComparing` methods have variants that avoid boxing of `int`, `long`, or `double` values. An easier way of producing the preceding operation would be

```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

If your key function can return `null`, you will like the `nullsFirst` and `nullsLast` adapters. These static methods take an existing comparator and modify it so that it doesn't throw an exception when encountering `null` values but ranks them as smaller or larger than regular values. For example, suppose `getMiddleName` returns a `null` when a person has no middle name. Then you can use `Comparator.comparing(Person::getMiddleName(), Comparator.nullsFirst(...))`.

The `nullsFirst` method needs a comparator—in this case, one that compares two strings. The `naturalOrder` method makes a comparator for any class implementing `Comparable`. A `Comparator.<String>naturalOrder()` is what we need. Here is the complete call for sorting by potentially null middle names. I use a static import of `java.util.Comparator.*`, to make the expression more legible. Note that the type for `naturalOrder` is inferred.

```
Arrays.sort(people, comparing(Person::getMiddleName, nullsFirst(naturalOrder())));
```

The static `reverseOrder` method gives the reverse of the natural order. To reverse any comparator, use the `reversed` instance method. For example, `naturalOrder().reversed()` is the same as `reverseOrder()`.

6.4 Inner Classes

An *inner class* is a class that is defined inside another class. Why would you want to do that? There are three reasons:

- Inner class methods can access the data from the scope in which they are defined—including the data that would otherwise be private.
- Inner classes can be hidden from other classes in the same package.
- *Anonymous inner classes* are handy when you want to define callbacks without writing a lot of code.

We will break up this rather complex topic into several steps.

1. Starting on page 331, you will see a simple inner class that accesses an instance field of its outer class.
2. On page 334, we cover the special syntax rules for inner classes.
3. Starting on page 335, we peek inside inner classes to see how they are translated into regular classes. Squeamish readers may want to skip that section.
4. Starting on page 339, we discuss *local inner classes* that can access local variables of the enclosing scope.
5. Starting on page 342, we introduce *anonymous inner classes* and show how they were commonly used to implement callbacks before Java had lambda expressions.

6. Finally, starting on page 346, you will see how *static inner classes* can be used for nested helper classes.



C++ NOTE: C++ has *nested classes*. A nested class is contained inside the scope of the enclosing class. Here is a typical example: A linked list class defines a class to hold the links, and a class to define an iterator position.

```
class LinkedList
{
public:
    class Iterator // a nested class
    {
    public:
        void insert(int x);
        int erase();
        ...
    };
    ...
private:
    class Link // a nested class
    {
    public:
        Link* next;
        int data;
    };
    ...
};
```

The nesting is a relationship between *classes*, not *objects*. A `LinkedList` object does *not* have subobjects of type `Iterator` or `Link`.

There are two benefits: *name control* and *access control*. The name `Iterator` is nested inside the `LinkedList` class, so it is known externally as `LinkedList::Iterator` and cannot conflict with another class called `Iterator`. In Java, this benefit is not as important because Java *packages* give the same kind of name control. Note that the `Link` class is in the *private* part of the `LinkedList` class. It is completely hidden from all other code. For that reason, it is safe to make its data fields public. They can be accessed by the methods of the `LinkedList` class (which has a legitimate need to access them) but they are not visible elsewhere. In Java, this kind of control was not possible until inner classes were introduced.

However, the Java inner classes have an additional feature that makes them richer and more useful than nested classes in C++. An object that comes from an inner class has an implicit reference to the outer class object that instantiated it. Through this pointer, it gains access to the total state of the outer object. You will see the details of the Java mechanism later in this chapter.

In Java, static inner classes do not have this added pointer. They are the Java analog to nested classes in C++.

6.4.1 Use of an Inner Class to Access Object State

The syntax for inner classes is rather complex. For that reason, we present a simple but somewhat artificial example to demonstrate the use of inner classes. We refactor the `TimerTest` example and extract a `TalkingClock` class. A talking clock is constructed with two parameters: the interval between announcements and a flag to turn beeps on or off.

```
public class TalkingClock
{
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep) { . . . }
    public void start() { . . . }

    public class TimePrinter implements ActionListener
        // an inner class
    {
        . . .
    }
}
```

Note that the `TimePrinter` class is now located inside the `TalkingClock` class. This does *not* mean that every `TalkingClock` has a `TimePrinter` instance field. As you will see, the `TimePrinter` objects are constructed by methods of the `TalkingClock` class.

Here is the `TimePrinter` class in greater detail. Note that the `actionPerformed` method checks the `beep` flag before emitting a beep.

```
public class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is " + new Date());
        if (beep) Toolkit.getDefaultToolkit().beep();
    }
}
```

Something surprising is going on. The `TimePrinter` class has no instance field or variable named `beep`. Instead, `beep` refers to the field of the `TalkingClock` object that created this `TimePrinter`. This is quite innovative. Traditionally, a method could refer to the data fields of the object invoking the method. An inner class method gets to access both its own data fields *and* those of the outer object creating it.

For this to work, an object of an inner class always gets an implicit reference to the object that created it (see Figure 6.3).

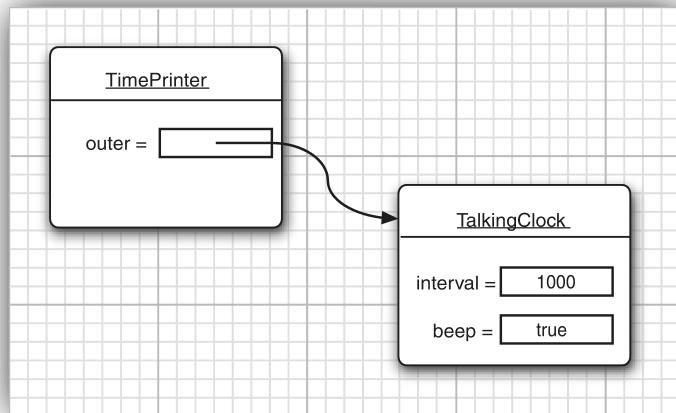


Figure 6.3 An inner class object has a reference to an outer class object

This reference is invisible in the definition of the inner class. However, to illuminate the concept, let us call the reference to the outer object `outer`. Then the `actionPerformed` method is equivalent to the following:

```
public void actionPerformed(ActionEvent event)
{
    System.out.println("At the tone, the time is " + new Date());
    if (outer.beep) Toolkit.getDefaultToolkit().beep();
}
```

The outer class reference is set in the constructor. The compiler modifies all inner class constructors, adding a parameter for the outer class reference. The `TimePrinter` class defines no constructors; therefore, the compiler synthesizes a no-argument constructor, generating code like this:

```
public TimePrinter(TalkingClock clock) // automatically generated code
{
    outer = clock;
}
```

Again, please note that `outer` is not a Java keyword. We just use it to illustrate the mechanism involved in an inner class.

When a `TimePrinter` object is constructed in the `start` method, the compiler passes the `this` reference to the current talking clock into the constructor:

```
ActionListener listener = new TimePrinter(this); // parameter automatically added
```

Listing 6.7 shows the complete program that tests the inner class. Have another look at the access control. Had the `TimePrinter` class been a regular class, it would have needed to access the `beep` flag through a public method of the `TalkingClock` class. Using an inner class is an improvement. There is no need to provide accessors that are of interest only to one other class.



NOTE: We could have declared the `TimePrinter` class as private. Then only `TalkingClock` methods would be able to construct `TimePrinter` objects. Only inner classes can be private. Regular classes always have either package or public visibility.

Listing 6.7 innerClass/InnerClassTest.java

```
1 package innerClass;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.util.*;
6 import javax.swing.*;
7 import javax.swing.Timer;
8
9 /**
10  * This program demonstrates the use of inner classes.
11  * @version 1.11 2015-05-12
12  * @author Cay Horstmann
13  */
14 public class InnerClassTest
15 {
16     public static void main(String[] args)
17     {
18         TalkingClock clock = new TalkingClock(1000, true);
19         clock.start();
20
21         // keep program running until user selects "Ok"
22         JOptionPane.showMessageDialog(null, "Quit program?");
23         System.exit(0);
24     }
25 }
26
27 /**
28  * A clock that prints the time in regular intervals.
29 */
```

(Continues)

Listing 6.7 (Continued)

```
30 class TalkingClock
31 {
32     private int interval;
33     private boolean beep;
34
35     /**
36      * Constructs a talking clock
37      * @param interval the interval between messages (in milliseconds)
38      * @param beep true if the clock should beep
39      */
40     public TalkingClock(int interval, boolean beep)
41     {
42         this.interval = interval;
43         this.beep = beep;
44     }
45
46     /**
47      * Starts the clock.
48      */
49     public void start()
50     {
51         ActionListener listener = new TimePrinter();
52         Timer t = new Timer(interval, listener);
53         t.start();
54     }
55
56     public class TimePrinter implements ActionListener
57     {
58         public void actionPerformed(ActionEvent event)
59         {
60             System.out.println("At the tone, the time is " + new Date());
61             if (beep) Toolkit.getDefaultToolkit().beep();
62         }
63     }
64 }
```

6.4.2 Special Syntax Rules for Inner Classes

In the preceding section, we explained the outer class reference of an inner class by calling it `outer`. Actually, the proper syntax for the outer reference is a bit more complex. The expression

`OuterClass.this`

denotes the outer class reference. For example, you can write the `actionPerformed` method of the `TimePrinter` inner class as

```
public void actionPerformed(ActionEvent event)
{
    ...
    if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

Conversely, you can write the inner object constructor more explicitly, using the syntax

outerObject.new InnerClass(construction parameters)

For example:

```
ActionListener listener = this.new TimePrinter();
```

Here, the outer class reference of the newly constructed `TimePrinter` object is set to the `this` reference of the method that creates the inner class object. This is the most common case. As always, the `this.` qualifier is redundant. However, it is also possible to set the outer class reference to another object by explicitly naming it. For example, since `TimePrinter` is a public inner class, you can construct a `TimePrinter` for any talking clock:

```
TalkingClock jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

Note that you refer to an inner class as

OuterClass.InnerClass

when it occurs outside the scope of the outer class.



NOTE: Any static fields declared in an inner class must be `final`. There is a simple reason. One expects a unique instance of a static field, but there is a separate instance of the inner class for each outer object. If the field was not `final`, it might not be unique.

An inner class cannot have `static` methods. The Java Language Specification gives no reason for this limitation. It would have been possible to allow static methods that only access static fields and methods from the enclosing class. Apparently, the language designers decided that the complexities outweighed the benefits.

6.4.3 Are Inner Classes Useful? Actually Necessary? Secure?

When inner classes were added to the Java language in Java 1.1, many programmers considered them a major new feature that was out of character with the Java philosophy of being simpler than C++. The inner class syntax is undeniably

complex. (It gets more complex as we study anonymous inner classes later in this chapter.) It is not obvious how inner classes interact with other features of the language, such as access control and security.

By adding a feature that was elegant and interesting rather than needed, has Java started down the road to ruin which has afflicted so many other languages?

While we won't try to answer this question completely, it is worth noting that inner classes are a phenomenon of the *compiler*, not the virtual machine. Inner classes are translated into regular class files with \$ (dollar signs) delimiting outer and inner class names, and the virtual machine does not have any special knowledge about them.

For example, the `TimePrinter` class inside the `TalkingClock` class is translated to a class file `TalkingClock$TimePrinter.class`. To see this at work, try the following experiment: run the `ReflectionTest` program of Chapter 5, and give it the class `TalkingClock$TimePrinter` to reflect upon. Alternatively, simply use the `javap` utility:

```
javap -private ClassName
```



NOTE: If you use UNIX, remember to escape the \$ character when you supply the class name on the command line. That is, run the `ReflectionTest` or `javap` program as

```
java reflection.ReflectionTest innerClass.TalkingClock\$TimePrinter
```

or

```
javap -private innerClass.TalkingClock\$TimePrinter
```

You will get the following printout:

```
public class TalkingClock$TimePrinter
{
    public TalkingClock$TimePrinter(TalkingClock);
    public void actionPerformed(java.awt.event.ActionEvent);
    final TalkingClock this$0;
}
```

You can plainly see that the compiler has generated an additional instance field, `this$0`, for the reference to the outer class. (The name `this$0` is synthesized by the compiler—you cannot refer to it in your code.) You can also see the `TalkingClock` parameter for the constructor.

If the compiler can automatically do this transformation, couldn't you simply program the same mechanism by hand? Let's try it. We would make `TimePrinter` a

regular class, outside the TalkingClock class. When constructing a TimePrinter object, we pass it the `this` reference of the object that is creating it.

```
class TalkingClock
{
    .
    .
    public void start()
    {
        ActionListener listener = new TimePrinter(this);
        Timer t = new Timer(interval, listener);
        t.start();
    }
}

class TimePrinter implements ActionListener
{
    private TalkingClock outer;
    .
    .
    public TimePrinter(TalkingClock clock)
    {
        outer = clock;
    }
}
```

Now let us look at the `actionPerformed` method. It needs to access `outer.beep`.

```
if (outer.beep) . . . // Error
```

Here we run into a problem. The inner class can access the private data of the outer class, but our external `TimePrinter` class cannot.

Thus, inner classes are genuinely more powerful than regular classes because they have more access privileges.

You may well wonder how inner classes manage to acquire those added access privileges, if they are translated to regular classes with funny names—the virtual machine knows nothing at all about them. To solve this mystery, let's again use the `ReflectionTest` program to spy on the `TalkingClock` class:

```
class TalkingClock
{
    private int interval;
    private boolean beep;

    public TalkingClock(int, boolean);

    static boolean access$0(TalkingClock);
    public void start();
}
```

Notice the static `access$0` method that the compiler added to the outer class. It returns the `beep` field of the object that is passed as a parameter. (The method name might be slightly different, such as `access$000`, depending on your compiler.)

The inner class methods call that method. The statement

```
if (beep)
```

in the `actionPerformed` method of the `TimePrinter` class effectively makes the following call:

```
if (TalkingClock.access$0(outer))
```

Is this a security risk? You bet it is. It is an easy matter for someone else to invoke the `access$0` method to read the private `beep` field. Of course, `access$0` is not a legal name for a Java method. However, hackers who are familiar with the structure of class files can easily produce a class file with virtual machine instructions to call that method, for example, by using a hex editor. Since the secret access methods have package visibility, the attack code would need to be placed inside the same package as the class under attack.

To summarize, if an inner class accesses a private data field, then it is possible to access that data field through other classes added to the package of the outer class, but to do so requires skill and determination. A programmer cannot accidentally obtain access but must intentionally build or modify a class file for that purpose.



NOTE: The synthesized constructors and methods can get quite convoluted. (Skip this note if you are squeamish.) Suppose we turn `TimePrinter` into a private inner class. There are no private classes in the virtual machine, so the compiler produces the next best thing: a package-visible class with a private constructor

```
private TalkingClock$TimePrinter(TalkingClock);
```

Of course, nobody can call that constructor, so there is a second package-visible constructor

```
TalkingClock$TimePrinter(TalkingClock, TalkingClock$1);
```

that calls the first one. The `TalkingClock$1` class is synthesized solely to distinguish this constructor from others.

The compiler translates the constructor call in the `start` method of the `TalkingClock` class to

```
new TalkingClock$TimePrinter(this, null)
```

6.4.4 Local Inner Classes

If you look carefully at the code of the TalkingClock example, you will find that you need the name of the type TimePrinter only once: when you create an object of that type in the start method.

In a situation like this, you can define the class *locally in a single method*.

```
public void start()
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone, the time is " + new Date());
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }

    ActionListener listener = new TimePrinter();
    Timer t = new Timer(interval, listener);
    t.start();
}
```

Local classes are never declared with an access specifier (that is, `public` or `private`). Their scope is always restricted to the block in which they are declared.

Local classes have one great advantage: They are completely hidden from the outside world—not even other code in the TalkingClock class can access them. No method except start has any knowledge of the TimePrinter class.

6.4.5 Accessing Variables from Outer Methods

Local classes have another advantage over other inner classes. Not only can they access the fields of their outer classes; they can even access local variables! However, those local variables must be *effectively final*. That means, they may never change once they have been assigned.

Here is a typical example. Let's move the `interval` and `beep` parameters from the TalkingClock constructor to the start method.

```
public void start(int interval, boolean beep)
{
    class TimePrinter implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
```

```
        System.out.println("At the tone, the time is " + new Date());
        if (beep) Toolkit.getDefaultToolkit().beep();
    }
}

ActionListener listener = new TimePrinter();
Timer t = new Timer(interval, listener);
t.start();
}
```

Note that the `TalkingClock` class no longer needs to store a `beep` instance field. It simply refers to the `beep` parameter variable of the `start` method.

Maybe this should not be so surprising. The line

```
if (beep) . . .
```

is, after all, ultimately inside the `start` method, so why shouldn't it have access to the value of the `beep` variable?

To see why there is a subtle issue here, let's consider the flow of control more closely.

1. The `start` method is called.
2. The object variable `listener` is initialized by a call to the constructor of the inner class `TimePrinter`.
3. The `listener` reference is passed to the `Timer` constructor, the timer is started, and the `start` method exits. At this point, the `beep` parameter variable of the `start` method no longer exists.
4. A second later, the `actionPerformed` method executes `if (beep) . . .`

For the code in the `actionPerformed` method to work, the `TimePrinter` class must have copied the `beep` field as a local variable of the `start` method, before the `beep` parameter value went away. That is indeed exactly what happens. In our example, the compiler synthesizes the name `TalkingClock$1TimePrinter` for the local inner class. If you use the `ReflectionTest` program again to spy on the `TalkingClock$1TimePrinter` class, you will get the following output:

```
class TalkingClock$1TimePrinter
{
    TalkingClock$1TimePrinter(TalkingClock, boolean);

    public void actionPerformed(java.awt.event.ActionEvent);

    final boolean val$beep;
    final TalkingClock this$0;
}
```

Note the `boolean` parameter to the constructor and the `val$beep` instance variable. When an object is created, the value `beep` is passed into the constructor and stored in the `val$beep` field. The compiler detects access of local variables, makes matching instance fields for each one, and copies the local variables into the constructor so that the instance fields can be initialized.

From the programmer's point of view, local variable access is quite pleasant. It makes your inner classes simpler by reducing the instance fields that you need to program explicitly.

As we already mentioned, the methods of a local class can refer only to local variables that are declared `final`. For that reason, the `beep` parameter was declared `final` in our example. A local variable that is declared `final` cannot be modified after it has been initialized. Thus, it is guaranteed that the local variable and the copy made inside the local class will always have the same value.



NOTE: Before Java SE 8, it was necessary to declare any local variables that are accessed from local classes as `final`. For example, this is how the `start` method would have been declared so that the inner class can access the `beep` parameter:

```
public void start(int interval, final boolean beep)
```

The “effectively `final`” restriction is sometimes inconvenient. Suppose, for example, that you want to update a counter in the enclosing scope. Here, we want to count how often the `compareTo` method is called during sorting:

```
int counter = 0;
Date[] dates = new Date[100];
for (int i = 0; i < dates.length; i++)
    dates[i] = new Date()
{
    public int compareTo(Date other)
    {
        counter++; // Error
        return super.compareTo(other);
    }
};
Arrays.sort(dates);
System.out.println(counter + " comparisons.");
```

You can't declare `counter` as `final` because you clearly need to update it. You can't replace it with an `Integer` because `Integer` objects are immutable. A remedy is to use an array of length 1:

```

int[] counter = new int[1];
for (int i = 0; i < dates.length; i++)
    dates[i] = new Date()
{
    public int compareTo(Date other)
    {
        counter[0]++;
        return super.compareTo(other);
    }
};

```

When inner classes were first invented, a prototype version of the compiler automatically made this transformation for all local variables that were modified in the inner class. However, this was later abandoned. After all, there is a danger. When the code in the inner class is executed at the same time in multiple threads, the concurrent updates can lead to race conditions—see Chapter 14.

6.4.6 Anonymous Inner Classes

When using local inner classes, you can often go a step further. If you want to make only a single object of this class, you don't even need to give the class a name. Such a class is called an *anonymous inner class*.

```

public void start(int interval, boolean beep)
{
    ActionListener listener = new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone, the time is " + new Date());
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    };
    Timer t = new Timer(interval, listener);
    t.start();
}

```

This syntax is very cryptic indeed. What it means is this: Create a new object of a class that implements the `ActionListener` interface, where the required method `actionPerformed` is the one defined inside the braces `{ }`.

In general, the syntax is

```

new SuperType(construction parameters)
{
    inner class methods and data
}

```

Here, *SuperType* can be an interface, such as `ActionListener`; then, the inner class implements that interface. *SuperType* can also be a class; then, the inner class extends that class.

An anonymous inner class cannot have constructors because the name of a constructor must be the same as the name of a class, and the class has no name. Instead, the construction parameters are given to the *superclass* constructor. In particular, whenever an inner class implements an interface, it cannot have any construction parameters. Nevertheless, you must supply a set of parentheses as in

```
new InterfaceType()
{
    methods and data
}
```

You have to look carefully to see the difference between the construction of a new object of a class and the construction of an object of an anonymous inner class extending that class.

```
Person queen = new Person("Mary");
// a Person object
Person count = new Person("Dracula") { . . . };
// an object of an inner class extending Person
```

If the closing parenthesis of the construction parameter list is followed by an opening brace, then an anonymous inner class is being defined.

Listing 6.8 contains the complete source code for the talking clock program with an anonymous inner class. If you compare this program with Listing 6.7, you will see that in this case, the solution with the anonymous inner class is quite a bit shorter and, hopefully, with some practice, as easy to comprehend.

For many years, Java programmers routinely used anonymous inner classes for event listeners and other callbacks. Nowadays, you are better off using a lambda expression. For example, the start method from the beginning of this section can be written much more concisely with a lambda expression like this:

```
public void start(int interval, boolean beep)
{
    Timer t = new Timer(interval, event ->
    {
        System.out.println("At the tone, the time is " + new Date());
        if (beep) Toolkit.getDefaultToolkit().beep();
    });
    t.start();
}
```



NOTE: The following trick, called *double brace initialization*, takes advantage of the inner class syntax. Suppose you want to construct an array list and pass it to a method:

```
ArrayList<String> friends = new ArrayList<>();
friends.add("Harry");
friends.add("Tony");
invite(friends);
```

If you don't need the array list again, it would be nice to make it anonymous. But then how can you add the elements? Here is how:

```
invite(new ArrayList<String>() {{ add("Harry"); add("Tony"); }});
```

Note the double braces. The outer braces make an anonymous subclass of `ArrayList`. The inner braces are an object construction block (see Chapter 4).



CAUTION: It is often convenient to make an anonymous subclass that is almost, but not quite, like its superclass. But you need to be careful with the `equals` method. In Chapter 5, we recommended that your `equals` methods use a test

```
if (getClass() != other.getClass()) return false;
```

An anonymous subclass will fail this test.



TIP: When you produce logging or debugging messages, you often want to include the name of the current class, such as

```
System.err.println("Something awful happened in " + getClass());
```

But that fails in a static method. After all, the call to `getClass` calls `this.getClass()`, and a static method has no `this`. Use the following expression instead:

```
new Object().getClass().getEnclosingClass() // gets class of static method
```

Here, `new Object()` makes an anonymous object of an anonymous subclass of `Object`, and `getEnclosingClass` gets its enclosing class—that is, the class containing the static method.

Listing 6.8 anonymousInnerClass/AnonymousInnerClassTest.java

```
1 package anonymousInnerClass;
2
3 import java.awt.*;
4 import java.awt.event.*;
```

```
5 import java.util.*;
6 import javax.swing.*;
7 import javax.swing.Timer;
8
9 /**
10  * This program demonstrates anonymous inner classes.
11  * @version 1.11 2015-05-12
12  * @author Cay Horstmann
13 */
14 public class AnonymousInnerClassTest
15 {
16     public static void main(String[] args)
17     {
18         TalkingClock clock = new TalkingClock();
19         clock.start(1000, true);
20
21         // keep program running until user selects "Ok"
22         JOptionPane.showMessageDialog(null, "Quit program?");
23         System.exit(0);
24     }
25 }
26
27 /**
28  * A clock that prints the time in regular intervals.
29 */
30 class TalkingClock
31 {
32     /**
33      * Starts the clock.
34      * @param interval the interval between messages (in milliseconds)
35      * @param beep true if the clock should beep
36     */
37     public void start(int interval, boolean beep)
38     {
39         ActionListener listener = new ActionListener()
40         {
41             public void actionPerformed(ActionEvent event)
42             {
43                 System.out.println("At the tone, the time is " + new Date());
44                 if (beep) Toolkit.getDefaultToolkit().beep();
45             }
46         };
47         Timer t = new Timer(interval, listener);
48         t.start();
49     }
50 }
```

6.4.7 Static Inner Classes

Occasionally, you may want to use an inner class simply to hide one class inside another—but you don’t need the inner class to have a reference to the outer class object. You can suppress the generation of that reference by declaring the inner class static.

Here is a typical example of where you would want to do this. Consider the task of computing the minimum and maximum value in an array. Of course, you write one method to compute the minimum and another method to compute the maximum. When you call both methods, the array is traversed twice. It would be more efficient to traverse the array only once, computing both the minimum and the maximum simultaneously.

```
double min = Double.POSITIVE_INFINITY;
double max = Double.NEGATIVE_INFINITY;
for (double v : values)
{
    if (min > v) min = v;
    if (max < v) max = v;
}
```

However, the method must return two numbers. We can achieve that by defining a class `Pair` that holds two values:

```
class Pair
{
    private double first;
    private double second;

    public Pair(double f, double s)
    {
        first = f;
        second = s;
    }
    public double getFirst() { return first; }
    public double getSecond() { return second; }
}
```

The `minmax` method can then return an object of type `Pair`.

```
class ArrayAlg
{
    public static Pair minmax(double[] values)
    {
        ...
        return new Pair(min, max);
    }
}
```

The caller of the method uses the `getFirst` and `getSecond` methods to retrieve the answers:

```
Pair p = ArrayAlg.minmax(d);
System.out.println("min = " + p.getFirst());
System.out.println("max = " + p.getSecond());
```

Of course, the name `Pair` is an exceedingly common name, and in a large project, it is quite possible that some other programmer had the same bright idea—but made a `Pair` class that contains a pair of strings. We can solve this potential name clash by making `Pair` a public inner class inside `ArrayAlg`. Then the class will be known to the public as `ArrayAlg.Pair`:

```
ArrayAlg.Pair p = ArrayAlg.minmax(d);
```

However, unlike the inner classes that we used in previous examples, we do not want to have a reference to any other object inside a `Pair` object. That reference can be suppressed by declaring the inner class static:

```
class ArrayAlg
{
    public static class Pair
    {
        ...
    }
    ...
}
```

Of course, only inner classes can be declared static. A static inner class is exactly like any other inner class, except that an object of a static inner class does not have a reference to the outer class object that generated it. In our example, we must use a static inner class because the inner class object is constructed inside a static method:

```
public static Pair minmax(double[] d)
{
    ...
    return new Pair(min, max);
}
```

Had the `Pair` class not been declared as static, the compiler would have complained that there was no implicit object of type `ArrayAlg` available to initialize the inner class object.



NOTE: Use a static inner class whenever the inner class does not need to access an outer class object. Some programmers use the term *nested class* to describe static inner classes.



NOTE: Unlike regular inner classes, static inner classes can have static fields and methods.

NOTE: Inner classes that are declared inside an interface are automatically static and public.

Listing 6.9 contains the complete source code of the `ArrayAlg` class and the nested `Pair` class.

Listing 6.9 staticInnerClass/StaticInnerClassTest.java

```
1 package staticInnerClass;
2
3 /**
4 * This program demonstrates the use of static inner classes.
5 * @version 1.02 2015-05-12
6 * @author Cay Horstmann
7 */
8 public class StaticInnerClassTest
9 {
10     public static void main(String[] args)
11     {
12         double[] d = new double[20];
13         for (int i = 0; i < d.length; i++)
14             d[i] = 100 * Math.random();
15         ArrayAlg.Pair p = ArrayAlg.minmax(d);
16         System.out.println("min = " + p.getFirst());
17         System.out.println("max = " + p.getSecond());
18     }
19 }
20
21 class ArrayAlg
22 {
23     /**
24      * A pair of floating-point numbers
25      */
26     public static class Pair
27     {
28         private double first;
29         private double second;
30     }
```

```
31  /**
32   * Constructs a pair from two floating-point numbers
33   * @param f the first number
34   * @param s the second number
35   */
36  public Pair(double f, double s)
37  {
38      first = f;
39      second = s;
40  }
41
42  /**
43   * Returns the first number of the pair
44   * @return the first number
45   */
46  public double getFirst()
47  {
48      return first;
49  }
50
51  /**
52   * Returns the second number of the pair
53   * @return the second number
54   */
55  public double getSecond()
56  {
57      return second;
58  }
59 }
60
61 /**
62  * Computes both the minimum and the maximum of an array
63  * @param values an array of floating-point numbers
64  * @return a pair whose first element is the minimum and whose second element
65  * is the maximum
66  */
67  public static Pair minmax(double[] values)
68  {
69      double min = Double.POSITIVE_INFINITY;
70      double max = Double.NEGATIVE_INFINITY;
71      for (double v : values)
72      {
73          if (min > v) min = v;
74          if (max < v) max = v;
75      }
76      return new Pair(min, max);
77  }
78 }
```

6.5 Proxies

In the final section of this chapter, we discuss *proxies*. You can use a proxy to create, at runtime, new classes that implement a given set of interfaces. Proxies are only necessary when you don't yet know at compile time which interfaces you need to implement. This is not a common situation for application programmers, and you should feel free to skip this section if you are not interested in advanced wizardry. However, for certain systems programming applications, the flexibility that proxies offer can be very important.

6.5.1 When to Use Proxies

Suppose you want to construct an object of a class that implements one or more interfaces whose exact nature you may not know at compile time. This is a difficult problem. To construct an actual class, you can simply use the `newInstance` method or use reflection to find a constructor. But you can't instantiate an interface. You need to define a new class in a running program.

To overcome this problem, some programs generate code, place it into a file, invoke the compiler, and then load the resulting class file. Naturally, this is slow, and it also requires deployment of the compiler together with the program. The *proxy* mechanism is a better solution. The proxy class can create brand-new classes at runtime. Such a proxy class implements the interfaces that you specify. In particular, the proxy class has the following methods:

- All methods required by the specified interfaces; and
- All methods defined in the `Object` class (`toString`, `equals`, and so on).

However, you cannot define new code for these methods at runtime. Instead, you must supply an *invocation handler*. An invocation handler is an object of any class that implements the `InvocationHandler` interface. That interface has a single method:

```
Object invoke(Object proxy, Method method, Object[] args)
```

Whenever a method is called on the proxy object, the `invoke` method of the invocation handler gets called, with the `Method` object and parameters of the original call. The invocation handler must then figure out how to handle the call.

6.5.2 Creating Proxy Objects

To create a proxy object, use the `newProxyInstance` method of the `Proxy` class. The method has three parameters:

- A *class loader*. As part of the Java security model, different class loaders can be used for system classes, classes that are downloaded from the Internet, and so on. We will discuss class loaders in Chapter 9 of Volume II. For now, we specify `null` to use the default class loader.
- An array of `Class` objects, one for each interface to be implemented.
- An invocation handler.

There are two remaining questions. How do we define the handler? And what can we do with the resulting proxy object? The answers depend, of course, on the problem that we want to solve with the proxy mechanism. Proxies can be used for many purposes, such as

- Routing method calls to remote servers
- Associating user interface events with actions in a running program
- Tracing method calls for debugging purposes

In our example program, we use proxies and invocation handlers to trace method calls. We define a `TraceHandler` wrapper class that stores a wrapped object. Its `invoke` method simply prints the name and parameters of the method to be called and then calls the method with the wrapped object as the implicit parameter.

```
class TraceHandler implements InvocationHandler
{
    private Object target;

    public TraceHandler(Object t)
    {
        target = t;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable
    {
        // print method name and parameters
        . . .
        // invoke actual method
        return m.invoke(target, args);
    }
}
```

Here is how you construct a proxy object that causes the tracing behavior whenever one of its methods is called:

```
Object value = . . .;
// construct wrapper
InvocationHandler handler = new TraceHandler(value);
// construct proxy for one or more interfaces
```

```
Class[] interfaces = new Class[] { Comparable.class};  
Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
```

Now, whenever a method from one of the interfaces is called on `proxy`, the method name and parameters are printed out and the method is then invoked on `value`.

In the program shown in Listing 6.10, we use proxy objects to trace a binary search. We fill an array with proxies to the integers 1 . . . 1000. Then we invoke the `binarySearch` method of the `Arrays` class to search for a random integer in the array. Finally, we print the matching element.

```
Object[] elements = new Object[1000];  
// fill elements with proxies for the integers 1 . . . 1000  
for (int i = 0; i < elements.length; i++)  
{  
    Integer value = i + 1;  
    elements[i] = Proxy.newProxyInstance(. . .); // proxy for value;  
  
    // construct a random integer  
    Integer key = new Random().nextInt(elements.length) + 1;  
  
    // search for the key  
    int result = Arrays.binarySearch(elements, key);  
  
    // print match if found  
    if (result >= 0) System.out.println(elements[result]);
```

The `Integer` class implements the `Comparable` interface. The proxy objects belong to a class that is defined at runtime. (It has a name such as `$Proxy0`.) That class also implements the `Comparable` interface. However, its `compareTo` method calls the `invoke` method of the proxy object's handler.



NOTE: As you saw earlier in this chapter, the `Integer` class actually implements `Comparable<Integer>`. However, at runtime, all generic types are erased and the proxy is constructed with the class object for the raw `Comparable` class.

The `binarySearch` method makes calls like this:

```
if (elements[i].compareTo(key) < 0) . . .
```

Since we filled the array with proxy objects, the `compareTo` calls call the `invoke` method of the `TraceHandler` class. That method prints the method name and parameters and then invokes `compareTo` on the wrapped `Integer` object.

Finally, at the end of the sample program, we call

```
System.out.println(elements[result]);
```

The `println` method calls `toString` on the proxy object, and that call is also redirected to the invocation handler.

Here is the complete trace of a program run:

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
288.compareTo(288)
288.toString()
```

You can see how the binary search algorithm homes in on the key by cutting the search interval in half in every step. Note that the `toString` method is proxied even though it does not belong to the `Comparable` interface—as you will see in the next section, certain `Object` methods are always proxied.

Listing 6.10 proxy/ProxyTest.java

```
1 package proxy;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5
6 /**
7  * This program demonstrates the use of proxies.
8  * @version 1.00 2000-04-13
9  * @author Cay Horstmann
10 */
11 public class ProxyTest
12 {
13     public static void main(String[] args)
14     {
15         Object[] elements = new Object[1000];
16
17         // fill elements with proxies for the integers 1 ... 1000
18         for (int i = 0; i < elements.length; i++)
19         {
20             Integer value = i + 1;
21             InvocationHandler handler = new TraceHandler(value);
22             Object proxy = Proxy.newProxyInstance(null, new Class[] { Comparable.class } , handler);
23             elements[i] = proxy;
24         }
25
26         // construct a random integer
27         Integer key = new Random().nextInt(elements.length) + 1;
```

(Continues)

Listing 6.10 (Continued)

```
28      // search for the key
29      int result = Arrays.binarySearch(elements, key);
30
31      // print match if found
32      if (result >= 0) System.out.println(elements[result]);
33  }
34 }
35 }
36
37 /**
38 * An invocation handler that prints out the method name and parameters, then
39 * invokes the original method
40 */
41 class TraceHandler implements InvocationHandler
42 {
43     private Object target;
44
45     /**
46     * Constructs a TraceHandler
47     * @param t the implicit parameter of the method call
48     */
49     public TraceHandler(Object t)
50     {
51         target = t;
52     }
53
54     public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
55     {
56         // print implicit argument
57         System.out.print(target);
58         // print method name
59         System.out.print("." + m.getName() + "(");
60         // print explicit arguments
61         if (args != null)
62         {
63             for (int i = 0; i < args.length; i++)
64             {
65                 System.out.print(args[i]);
66                 if (i < args.length - 1) System.out.print(", ");
67             }
68         }
69         System.out.println(")");
70
71         // invoke actual method
72         return m.invoke(target, args);
73     }
74 }
```

6.5.3 Properties of Proxy Classes

Now that you have seen proxy classes in action, let's go over some of their properties. Remember that proxy classes are created on the fly in a running program. However, once they are created, they are regular classes, just like any other classes in the virtual machine.

All proxy classes extend the class `Proxy`. A proxy class has only one instance field—the invocation handler, which is defined in the `Proxy` superclass. Any additional data required to carry out the proxy objects' tasks must be stored in the invocation handler. For example, when we proxied `Comparable` objects in the program shown in Listing 6.10, the `TraceHandler` wrapped the actual objects.

All proxy classes override the `toString`, `equals`, and `hashCode` methods of the `Object` class. Like all proxy methods, these methods simply call `invoke` on the invocation handler. The other methods of the `Object` class (such as `clone` and `getClass`) are not redefined.

The names of proxy classes are not defined. The `Proxy` class in Oracle's virtual machine generates class names that begin with the string `$Proxy`.

There is only one proxy class for a particular class loader and ordered set of interfaces. That is, if you call the `newProxyInstance` method twice with the same class loader and interface array, you get two objects of the same class. You can also obtain that class with the `getProxyClass` method:

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

A proxy class is always `public` and `final`. If all interfaces that the proxy class implements are `public`, the proxy class does not belong to any particular package. Otherwise, all non-`public` interfaces must belong to the same package, and the proxy class will also belong to that package.

You can test whether a particular `Class` object represents a proxy class by calling the `isProxyClass` method of the `Proxy` class.

java.lang.reflect.InvocationHandler 1.3

- `Object invoke(Object proxy, Method method, Object[] args)`

define this method to contain the action that you want carried out whenever a method was invoked on the proxy object.

java.lang.reflect.Proxy 1.3

- static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces)
returns the proxy class that implements the given interfaces.
- static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler handler)
constructs a new instance of the proxy class that implements the given interfaces.
All methods call the invoke method of the given handler object.
- static boolean isProxyClass(Class<?> cl)
returns true if cl is a proxy class.

This ends our final chapter on the fundamentals of the Java programming language. Interfaces, lambda expressions, and inner classes are concepts that you will encounter frequently. However, as we already mentioned, cloning and proxies are advanced techniques that are of interest mainly to library designers and tool builders, not application programmers. You are now ready to learn how to deal with exceptional situations in your programs in Chapter 7.

This page intentionally left blank

Index

Numbers

- (minus sign)
 - arithmetic operator, 56, 64
 - printf flag, 84
- operator, 61, 64
- _ (underscore)
 - delimiter, in number literals, 48
 - in instance field names (C++), 176
- , (comma)
 - operator (C++), 65
 - printf flag, 83–84
- ; (semicolon)
 - for statements, 45, 53
 - in class path (Windows), 191
- : (colon)
 - in assertions, 385
 - in class path (UNIX), 191
 - inheritance token (C++), 204
- :: operator (C++), 153, 161, 207, 320
- ! operator, 62, 64
- != operator, 62, 64, 101
- ? operator, 62, 64
- / (slash)
 - arithmetic operator, 56, 64
 - in file names, 785
- // comments, 46
- /* . . . */ comments, 46
- /** . . . */ comments, 46, 194
- . (period)
 - in class path, 191–192
 - in directory names (UNIX), 788
- ... (ellipsis), in varargs, 257
- ^ operator, 63–64, 316
- ~ operator, 63–64
- ', " (single, double quote), escape sequences for, 50
- ". . ." (double quotes), for strings, 45
- (left parenthesis), printf flag, 83–84
- () (empty parentheses), in method calls, 46
- (. . .) (parentheses)
 - for casts, 60, 64, 219

- for operator hierarchy, 64–65
- [] (empty square brackets), in generics, 421
- [. . .] (square brackets), for arrays, 111, 115
- {. . .} (curly braces)
 - for blocks, 44–45, 89
 - for enumerated type, 65
 - in lambda expressions, 316
- {}{. . .}} (double curly braces), in inner classes, 344
- @ (at), in javadoc comments, 194, 196
- \$ (dollar sign)
 - delimiter, for inner classes, 336
 - in variable names, 53
 - printf flag, 84
- * (asterisk)
 - arithmetic operator, 56, 64
 - echo character, 652
 - in class path, 191
 - in imports, 183
- \ (backslash)
 - escape sequence for, 50
 - in file names, 87, 785
- & (ampersand)
 - bitwise operator, 63–64
 - in bounding types, 423
 - in reference parameters (C++), 169
- && operator, 62, 64
- # (number sign)
 - in javadoc hyperlinks, 197
 - in property files, 599
 - printf flag, 84
- % (percent sign)
 - arithmetic operator, 56, 64
 - formatting output for, 83
 - printf flag, 84
- + (plus sign)
 - arithmetic operator, 56, 60, 64
 - for objects and strings, 66–67, 239
 - printf flag, 84
- ++ operator, 61, 64

- < (left angle bracket)
 - in shell syntax, 88
 - printf flag, 84–85
 - relational operator, 62, 64
- <? (in wildcard types), 443
- <<, >>, >>> operators, 63–64
- <= operator, 62, 64
 - <. .> (angle brackets), for type parameters, 245, 419
- > (right angle bracket)
 - in shell syntax, 88, 411
 - relational operator, 62, 64
- > (in lambda expressions), 315–317
- & (in shell syntax), 411
- = operator, 62, 64
- = operator, 54, 61
- == operator, 62, 64
 - for class objects, 262
 - for enumerated types, 258
 - for floating-point numbers, 101
 - for identity hash maps, 507
 - for strings, 69
 - for wrappers, 254
- | operator, 63–64
- || operator, 62, 64
- 0, 0b, 0x prefixes (in integers), 48
- 0, printf flag, 84
- >> (in shell syntax), 411
- 2D shapes, 560–569

- A**
- Absolute positioning (Swing), 723
- Abstract classes, 221–227
 - extending, 223
 - interfaces and, 297
 - object variables of, 223
- abstract keyword, 221–227
- Abstract methods, 222
 - in functional interfaces, 318
- AbstractAction class, 609, 612, 680, 683
- AbstractButton class, 627, 681–684
 - is/isSelected methods, 684
 - setAction method, 681
 - setActionCommand method, 663
 - setDisplayedMnemonicIndex method, 686, 688
 - setHorizontalTextPosition method, 682–683
 - setMnemonic method, 688
- abstractClasses/Employee.java, 226
- abstractClasses/Person.java, 226
- abstractClasses/PersonTest.java, 225
- abstractClasses/Student.java, 227
- AbstractCollection class, 467, 479
- AbstractQueue class, 463
- Accelerators (in menus), 687–688
- accept method (*FileFilter*), 755, 764
- acceptEither method (*CompletableFuture*), 934
- Access modifiers
 - checking, 265
 - final, 55, 157, 217–218, 295, 339–342, 886
 - private, 150, 189–190, 333
 - protected, 227–228, 283, 311
 - public, 42–43, 56, 147–150, 189–190, 289–290
 - public static final, 296
 - static, 44–45, 158–164
 - static final, 55
 - void, 44–45
- Access order, 505
- AccessibleObject class
 - isAccessible method, 275
 - setAccessible method, 272, 275
- Accessor methods, 141–145, 153–154, 444
- Accessory components, 757
- accumulate method (*LongAccumulator*), 888
- accumulateAndGet method (*AtomicType*), 887
- Action interface, 607–615, 680
 - actionPerformed method, 608
 - add/removePropertyChangeListener methods, 608–609
 - get/putValue methods, 608, 615
 - is/setEnabled methods, 608, 615
 - predefined action table names, 609
- Action listeners, 607–615
 - action/ActionFrame.java, 613
 - ActionEvent class, 588, 626–627
 - getActionCommand method, 598, 627
 - getModifiers method, 627
 - ActionListener interface, 626
 - actionPerformed method, 302–303, 314, 331–332, 337, 342, 589–593, 597, 601, 607, 609, 627, 897
 - overriding, 680
 - implementing, 318, 589, 597
- ActionMap class, 612
- Actions, 607–615
 - associating with keystrokes, 610
 - asynchronous, 931
 - names of, 612

- predefined, 609
ActiveX, 5, 15
Adapter classes, 603–607
add method
 of *ArrayList*, 245–251
 of *BigDecimal*, *BigInteger*, 110–111
 of *BlockingQueue*, 898–899
 of *ButtonGroup*, 663
 of *Collection*, 463, 467–469
 of *Container*, 591, 595, 641
 of *GregorianCalendar*, 142
 of *HashSet*, 487
 of *JFrame*, 555, 559
 of *JMenu*, 679, 681
 of *JToolBar*, 695–699
 of *List*, 470, 482
 of *ListIterator*, 470, 476–478, 483
 of *LongAdder*, 888
 of *Queue*, 494
 of *Set*, 471
addAll method
 of *ArrayList*, 417
 of *Collection*, 467–468
 of *Collections*, 523
 of *List*, 482
addChoosableFileFilter method (*JFileChooser*), 763
addComponent, addGroup methods (*GroupLayout*), 723
addFirst/Last methods
 of *Deque*, 494
 of *LinkedList*, 484
addHandler method (*Logger*), 406
addItem method (*JComboBox*), 669–671
Addition operator, 56, 64
 for different numeric types, 60
 for objects and strings, 66–67, 239
addLayoutComponent method (*LayoutManager*), 728
addPropertyChangeListener method (*Action*),
 608–609
addSeparator method
 of *JMenu*, 679, 681
 of *JToolBar*, 695–699
addShutdownHook method (*Runtime*), 182
addSuppressed method (*Throwable*), 377, 380
AdjustmentEvent class, 626
 methods of, 627
AdjustmentListener interface, 626
 adjustmentValueChanged method, 627
Adobe Flash, 9
Aggregation, 133–135
Algorithms, 130
 for binary search, 521–522
 for shuffling, 520
 for sorting, 518–521
 QuickSort, 117, 519
 simple, in the standard library, 522–524
 writing, 526–528
Algorithms + Data Structures = Programs
 (Wirth), 130
Algorithms in C++ (Sedgewick), 519
Alice in Wonderland (Carroll), 487, 490
allOf method (*EnumSet*), 508, 934
Alt+F4, in Windows, 688
and, andNot methods (*BitSet*), 533
Andreessen, Mark, 10
Annotations, 430
Anonymous arrays, 114
Anonymous inner classes, 329, 342–345
anonymousInnerClass/AnonymousInnerClassTest.java, 344
Antisymmetry rule, 295
anyOf method (*CompletableFuture*), 934
append method
 of *JTextArea*, 656, 951
 of *StringBuilder*, 77–78
appendCodePoint method (*StringBuilder*), 78
Applet class, 803
 destroy method, 808
 getAppletContext method, 818–820
 getAppletInfo method, 816
 getCodeBase, getDocumentBase methods, 816–817
 getImage, getAudioClip methods, 817
 getParameter method, 810–811, 816
 getParameterInfo method, 816
 init method, 807, 811
 play method, 817
 resize method, 808
 showStatus method, 819–820
 start, stop methods, 808
applet element (HTML), 34, 805, 808–810
 align attribute, 808
 alt attribute, 809
 archive attribute, 809
 code attribute, 809
 codebase attribute, 809
 height, width attributes, 807–808
 hspace, vspace attributes, 808
 name attribute, 810
 object attribute (obsolete), 809
applet/NotHelloWorld.java, 805

- AppletContext* interface, 818
 `getApplet`, `getApplets` methods, 818, 820
 `showDocument` method, 819–820
- Applets, 8–9, 14, 802–824
 accessing from JavaScript, 810
 aligning, 808
 changing warning string in, 190
 communicating to each other, 810, 818
 context of, 818
 debugging, 807
 digitally signed, 822–824
 executing, 805
 image and audio files in, 816–817
 multiple copies of, 813
 no title bars for, 807
 passing information to, 816
 printing in, 832
 resizing, 808–810
 running in a browser, 8, 33–39, 802–803,
 818–820
 serialized objects of, 809
 testing, 805–806
 trusted local, 35, 806
- appletviewer program, 33, 805–806
- Application Programming Interfaces (APIs),
 online documentation, 71, 74–77
- Applications
 cache of, 827
 closing by user, 545
 codebase of, 831
 compiling/running from the command
 line, 30–33
 debugging, 25–26, 358–366
 deploying, 779–838
 extensible, 217
 launching, 43
 localization of, 136, 393–394, 785
 monitoring and managing in JVM, 412
 platform-independent, 724
 preferences of, 788–800
 terminating, 45
 testing, 384–388
- applyToEither method (`CompletableFuture`), 934
- Arguments. *See* Parameters
- Arithmetic operators, 56–65
 accuracy of, 56
 autoboxing with, 253
 combining with assignment, 61
 precedence of, 64
- Array class, 276–279
 `get`, `getXxx`, `set`, `setXxx` methods, 279
 `getLength` method, 277, 279
 `newInstance` method, 276, 279
- Array lists, 112, 484
 anonymous, 344
 capacity of, 246
 elements of:
 accessing, 247–251
 adding, 245–249
 removing, 249
 traversing, 249
 generic, 244–252
 raw vs. typed, 251–252
- Array variables, 111
- ArrayBlockingQueue class, 899, 903
- ArrayDeque class, 462, 494–495
 as a concrete collection type, 472
- ArrayList class, 113, 244–252, 416–418, 474
 `add` method, 245–251
 `addAll` method, 417
 as a concrete collection type, 472
 `ensureCapacity` method, 246–247
 `get`, `set` methods, 247, 251
 `remove` method, 249, 251
 `removeIf` method, 319
 `size` method, 246–247
 `synchronized`, 914
 `toArray` method, 435
 `trimToSize` method, 246–247
- arrayList/ArrayListTest.java, 250
- Arrays, 111–127
 anonymous, 114
 circular, 462–463
 cloning, 311
 converting to collections, 525–526
 copying, 114–115
 on write, 912
 creating, 111
 elements of:
 computing in parallel, 913
 numbering, 112
 remembering types of, 214
 removing from the middle, 474
 traversing, 112–113, 122
 equality testing for, 234
 generic methods for, 276–279
 hash codes of, 238

- in command-line parameters, 116
initializing, 112, 114
multidimensional, 120–125, 240
not of generic types, 321, 431–432, 441
of integers, 240
of subclass/superclass references, 214
of wildcard types, 432
out-of-bounds access in, 360
parallel operations on, 912
printing, 122, 240
ragged, 124–127
size of, 112, 246, 277
 equal to 0, 114, 526
 equal to 1, 341
 increasing, 115
 setting at runtime, 244
 sorting, 117–120, 292, 912
 type erasure and, 434–436
- Arrays class
- asList method, 509, 516, 526
 - binarySearch method, 120, 352
 - copyOf method, 115, 119, 276
 - copyOfRange method, 119
 - deepToString method, 122, 240
 - equals method, 120, 234
 - fill method, 120
 - hashCode method, 238
 - sort method, 117–119, 290, 292, 294, 314, 318
 - toString method, 114, 119
- arrays/CopyOfTest.java, 278
- ArrayStoreException, 431, 433, 441
- Ascender, ascent (in typesetting), 576
- ASCII standard, 51, 575
- asList method (Arrays), 509, 516, 526
- assert keyword, 384–388
- Assertions, 384–388
- checking parameters with, 386–387
 - defined, 384
 - documenting assumptions with, 387–388
 - enabling/disabling, 385–386
- Assignment operator, 54, 61
- Asynchronous methods, 915
- atan, atan2 methods (`Math`), 58
- Atomic operations, 886–889
- client-side locking for, 883
 - in concurrent hash maps, 907–909
 - performance of, 888
- AtomicType classes, 887
- Audio files, accessing from applets, 816–817
- @author comment (javadoc), 196, 199
- Autoboxing, 252–256
- `AutoCloseable` interface, 376
- `close` method, 376–377
- await method (`Condition`), 856, 873–877, 893–895
- awaitUninterruptibly method (`Condition`), 893–895
- AWT (Abstract Window Toolkit), 538
- events in:
 - debugging, 774–778
 - hierarchy of, 624–628
 - tracing, 771
 - preferred field sizes in, 649
- AWTEvent class, 624
- B**
- \b (backspace escape sequence), 50
- Background
- default color for, 570–571
 - erasing, 842
 - painting, 558
- Backspace, escape sequence for, 50
- BadCastException, 451
- Barriers, 936–937
- Base classes. *See* Superclasses
- Baseline (in typesetting), 576, 718
- Basic multilingual planes, 51
- BasicButtonUI class, 637
- `BasicService` interface, 831
- `getCodeBase` method, 831, 836
 - `isWebBrowserSupported` method, 836
 - `showDocument` method, 836
- Batch files, 193
- Beans, 780
- beep method (`Toolkit`), 305
- `BiConsumer` interface, 326
- `BiFunction` interface, 319, 326
- BIG-5 standard, 51
- `BigDecimal`, `BigInteger` classes, 108–111
- `add`, `compareTo`, `subtract`, `multiply`, `divide`, `mod` methods, 110–111
 - `valueOf` method, 108, 110–111
- `BigIntegerTest/BigIntegerTest.java`, 109
- Binary search, 521–522
- `BinaryOperator` interface, 326
- `binarySearch` method
 - of Arrays, 120, 352
 - of Collections, 521–522

- BiPredicate* interface, 326
 Bit masks, 63, 616
 Bit sets, 532–536
 and the sieve of Eratosthenes benchmark, 533–536
 Bytecode files, 43
BitSet interface, 460, 532–536
 methods of, 533
 Bitwise operators, 63–64
 Blank lines, printing, 46
 Blocking queues, 898–905
BlockingQueue interface
 offerFirst/Last, pollFirst/Last methods, 905
 putFirst/Last, takeFirst/Last methods, 904
BlockingQueue interface
 add, element, peek, remove methods, 898–899
 offer, poll, put, take methods, 898–899,
 904
`blockingQueue/BlockingQueueTest.java`, 900
 Blocks, 44–45, 89–90
 nested, 89
 Boolean class
 converting from boolean, 252
 hashCode method, 237
 boolean operators, 62, 64
 boolean type, 52
 default initialization of, 172
 formatting output for, 83
 no casting to numeric types for, 61
 BooleanHolder class, 255
 Border layout manager, 641–644
`border/BorderFrame.java`, 665
 BorderFactory class, 664–668
 createTypeBorder methods, 664–667
 BorderLayout class, 641–644
 constants of, 642
 Borders, 664–668
 compound, 664
 rounded corners of, 665
 styles of, 664
 with a title, 664
`bounce/Ball.java`, 844
`bounce/BallComponent.java`, 845
`bounce/Bounce.java`, 842
`bounceThread/BounceThread.java`, 849
 Bounded collections, 463
 Bounding rectangle, 563–565
 Bounds checking, 115
 Box layout, 700
 break statement, 104–108
 labeled/unlabeled, 106
 missing, 412
 Bridge methods, 428–429, 440
 brighter method (*Color*), 571
 BrokenBarrierException, 937
 Browsers
 default, 831
 display area of, 819–820
 installing Java Plug-in in, 803
 Java-enabled, 809
 MIME types in, 825
 running applets in, 8, 33–39, 802–803,
 818–820
 status bar of, 819–820
 Buckets (of hash tables), 485
 Bulk operations, 524–525
`button/ButtonFrame.java`, 594
 ButtonGroup class, 660
 add method, 663
 getSelection method, 661, 663
ButtonModel interface, 636–638
 getActionCommand method, 661, 663
 getSelectedObjects method, 661
 properties of, 637
 Buttons
 appearance of, 632
 associating actions with, 610
 clicking, 592
 creating, 591
 event handling for, 591–595
 listening to, 592
 model-view-controller analysis of,
 636–638
 rearranging automatically, 639
`buttonUIListener` class, 637
 Byte class
 converting from byte, 252
 hashCode method, 237
 byte type, 47
`ByteArrayOutputStream` class, 830

C

- C programming language
 assert macro in, 385
 event-driven programming in,
 588
 function pointers in, 279
 integer types in, 6

C# programming language, 8
 delegates in, 280
 polymorphism in, 218
 useful features of, 11

C++ programming language
 , (comma) operator in, 65
 :: operator in, 153, 207
 >> operator in, 64
 access privileges in, 156
 algorithms in, 518
 arrays in, 115, 126
 bitset template in, 532
 boolean values in, 52
 classes in, 45
 nested, 330
 code units and code points in, 70
 control flow in, 89
 copy constructors in, 139
 dynamic binding in, 209
 dynamic casts in, 221
 exceptions in, 361, 364–365, 369
 fields in:
 instance, 175–176
 static, 161
 for loop in, 100
 function pointers in, 279
 #include in, 184
 inheritance in, 204, 213, 297
 integer types in, 6, 47
 methods in:
 accessor, 142
 default, 300
 destructor, 181
 static, 161
 namespace, using directives in, 184
 new operator in, 151
 NULL, object pointers in, 139
 operator overloading in, 109
 passing parameters in, 167, 169
 performance of, compared to Java, 534
 polymorphism in, 218
 protected modifier in, 228
 pure virtual functions (= 0) in, 224
 references in, 139
 Standard Template Library in, 460, 465
 static member functions in, 45
 strings in, 68–69
 superclasses in, 208
 syntax of, 3
 templates in, 11, 420, 423, 426
 this pointer in, 176
 type parameters in, 422
 using iterators as parameters in, 530
 variables in, 55
 redefining in nested blocks, 90
 vector template in, 247
 virtual constructors in, 263
 void* pointer in, 229

Cache, 827

calculator/CalculatorPanel.java, 645

Calendar class, 140
 get/setTime methods, 218

Calendars
 displaying, 142–144
 vs. time measurement, 140

CalendarTest/CalendarTest.java, 144

Call by reference, 164

Call by value, 164–171

Callable interface, 927
 call method, 915, 919
 wrapper for, 916

Callables, 915–920

Callbacks, 302–305

Camel case (CamelCase), 43

cancel method (*Future*), 915, 920–921, 945

CancellationException, 945

Canned functionality, 934

canRead/write methods (*FileContents*), 837

Carriage return, escape sequence for, 50

case statement, 104

cast method (*Class*), 451

Casts, 60–61, 219–221
 bad, 360
 checking before attempting, 220

catch statement, 367–381

ceiling method (*NavigableSet*), 493

ChangeListener interface, 672
 stateChanged method, 672–673

char type, 50–51

Character class
 converting from char, 252
 hashCode method, 237
 isJavaIdentifierXxx methods, 53

Characters, formatting output for, 83

charAt method (*String*), 70, 72

chart/Chart.java, 813

checkBox/CheckBoxFrame.java, 658

- checkboxes, 657–659
 in menus, 683–684
- Checked exceptions, 261–264
 applicability of, 383
 declaring, 361–364
 suppressing with generics, 437–439
- Checked views, 513
- checkedCollection* methods (Collections), 515
- Child classes. *See* Subclasses
- Choice components, 657–678
 borders, 664–668
 checkboxes, 657–659
 combo boxes, 668–671
 radio buttons, 660–663
 sliders, 672–678
- ChronoLocalDate* interface, 446
- Church, Alonzo, 315
- circleLayout/CircleLayout.java*, 725
- circleLayout/CircleLayoutFrame.java*, 728
- Circular arrays, 462–463
- Clark, Jim, 10
- Clarke, Arthur C., 717
- Class class, 261–263
 cast method, 451
 forName method, 261, 265
 generic, 434, 450–453
 getClass method, 261
 getComponentType method, 277
 getConstructor, getDeclaredConstructor methods, 451
 getConstructors, getDeclaredConstructors methods, 266, 270
 getDeclaredMethods method, 266, 270, 280
 getEnumConstants method, 451
 getField, getDeclaredField methods, 275
 getFields, getDeclaredFields methods, 266, 270, 272, 275
 getGenericXxx methods, 457
 getImage, getAudioClip methods, 784
 getMethod method, 280
 getMethods method, 266, 270
 getName method, 244, 261–262
 getResource, getResourceAsStream methods, 784, 787
 getSuperclass method, 244, 451
 getTypeParameters method, 457
 newInstance method, 263, 265, 451
- Class constants, 55
- Class diagrams, 134–135
- .class file extension, 43
- Class files, 185, 190
 locating, 192
 names of, 43, 147
- class keyword, 42
- Class loaders, 351, 385
- Class path, 190–193
 checking directories on, 412
 setting, 193
- Class wins rule, 301
- Class<T> parameters, 452
- ClassCastException, 220, 276, 295, 435, 441, 513
- Classes, 131–132, 204–228
 abstract, 221–227, 297
 access privileges for, 156
 adapter, 603–607
 adding to packages, 185–188
 analyzing:
 capabilities of, 265–271
 objects of, at runtime, 271–276
 companion, 298–299
 constructors for, 149
 defining, 145–157
 at runtime, 350
 deprecated, 197
 designing, 133, 200–202
 documentation comments for, 194–198
 encapsulation of, 131–132, 153–156
 extending, 132
 final, 217–218
 generic, 245, 418–420, 441, 669
 helper, 706–712
 immutable, 157
 implementing multiple interfaces, 296–297
 importing, 183–184
 inner, 329–349
 anonymous, 606
 instances of, 131, 136
 loading, 262, 411
 multiple source files for, 149
 names of, 25, 43, 182, 201
 full package, 183
 number of basic types in, 200
 package scope of, 189
 parameters in, 152–153
 predefined, 135–145
 private methods in, 156–157
 protected, 227–228
 public, 194

accessing, 183
relationships between, 133–135
serializable, 412
sharing, among programs, 191
unit testing, 162
wrapper, 252–256

`ClassLoader` class, 388
`CLASSPATH` environment variable, 26, 193
`clear` method
 of `BitSet`, 533
 of `Collection`, 467, 469
`clearAssertionStatus` method (`ClassLoader`), 388
Client-side locking, 882–883
`clone` method
 of array types, 311
 of `Object`, 156, 306–313, 318
`clone/CloneTest.java`, 312
`clone/Employee.java`, 312
`Cloneable` interface, 306–313
`CloneNotSupportedException`, 310
`close` method
 of `AutoCloseable`, 376–377
 of `Closeable`, 376
 of `Handler`, 406
`Closeable` interface, 376
Closures, 323
Code errors, 359
Code planes, 52
Code points, code units, 52, 70
Codebase (in JNLP files), 831
`codePointAt`, `codePoints` methods (`String`), 72
`codePointCount` method (`String`), 70, 73
`Collection` interface, 463, 469, 479
 `add` method, 463, 467–469
 `addAll` method, 467–468
 `clear` method, 467, 469
 `contains`, `containsAll` methods, 467–468, 479
 `equals` method, 467
 generic, 466–469
 `isEmpty` method, 299, 467–468
 `iterator` method, 463, 468
 `remove`, `removeAll` methods, 467–468
 `removeIf` method, 468, 524
 `retain` method, 467
 `retainAll` method, 469
 `size` method, 467–468
 `toArray` method, 249, 467, 469

Collections, 459–536
 algorithms for, 517–528

bounded, 463
bulk operations in, 524–525
concrete, 472–496
concurrent modifications of, 479
converting to arrays, 525–526
debugging, 479
elements of:
 inserting, 469
 maximum, 517
 removing, 465
 traversing, 464
interfaces for, 460–471
legacy, 528–536
lightweight wrappers for, 509–510
ordered, 470, 476
performance of, 471, 486
searching in, 521–522
sorted, 489
thread-safe, 512–513, 905–915
type parameters for, 418
using for method parameters, 527

`Collections` class, 520
 `addAll` method, 523
 `binarySearch` method, 521–522
 `checkedCollection`, `emptyCollection` methods, 515
 `copy` method, 523
 `disjoint` method, 524
 `fill` method, 523
 `frequency` method, 524
 `indexOfSubList`, `lastIndexOfSubList` methods, 524
 `min`, `max` methods, 523
 `nCopies` method, 510, 515
 `replaceAll` method, 523
 `reverse` method, 524
 `rotate` method, 524
 `shuffle` method, 520–521
 `singleton`, `singletonCollection` methods, 510, 515
 `sort` method, 518–521
 `swap` method, 524
 `synchronizedCollection` methods, 512–513, 515, 915
 `unmodifiableCollection` methods, 511–512, 514

Collections framework. *See Java collections framework (JCF)*

Color choosers, 764–770

`Color` class, 569–573
 brighter, darker methods, 571
 predefined constants, 570

- colorChooser/ColorChooserPanel.java, 767
Colors
 background, 558, 570–571
 changing, 609
 custom, 570
 foreground, 570
 predefined, 570–572
 system, 571
Columns (of a text field), 649
com.sun.java package, 599
Combo boxes, 668–671
 adding items to, 669
 current selection in, 669
comboBox/ComboBoxFrame.java, 670
Command line
 compiling/launching Java from, 24
 parameters in, 116
Comments, 46–47
 blocks of, 46
 for automatic documentation, 46, 194–199
 in property files, 599
 not nesting, 47
 to the end of line, 46
Companion classes, 298–299
Comparable interface, 288, 352, 422–423, 446, 519
 compareTo method, 289–293
comparator method (*SortedMap*), 493, 500
Comparator interface, 305–306, 314, 328–329, 519
 chaining comparators in, 328
 comparing method, 328–329
 lambdas and, 318
 naturalOrder method, 329
 nullFirst/Last methods, 329
 reversed, reverseOrder methods, 329, 519, 521
 thenComparing method, 328–329
compare method (integer types), 294, 318
compareAndSet method (*AtomicType*), 887
compareTo method
 in subclasses, 295
 of *BigDecimal*, *BigInteger*, 110–111
 of *Comparable*, 289–293, 422, 446
 of *Enum*, 260
 of *String*, 72
Compilation errors, 29
Compiler
 autoboxing in, 254
 bridge methods in, 428
command-line options of, 412
creating bytecode files in, 43
deducting method types in, 421
enforcing throws specifiers in, 368
error messages in, 29, 363
just-in-time, 6–7, 14, 153, 218, 413, 534
launching, 25
optimizing method calls in, 7, 218
overloading resolution in, 215
shared strings in, 67, 69
translating inner classes in, 336
translating typed array lists in, 252
type parameters in, 417
warnings in, 105, 252
whitespace in, 44
CompletableFuture class
 acceptEither, applyToEither methods, 934
 allOf, anyOf methods, 934
 handle method, 933
 runAfterXXX methods, 934
 thenAccept, thenApply, thenApplyAsync, thenRun methods, 933
 thenAcceptBoth, thenCombine methods, 934
 thenCompose method, 932–933
 whenComplete method, 933
CompletionStage interface, 934
Component class, 627
 getBackground/Foreground methods, 573
 getFont method, 651
 getPreferredSize method, 557, 559
 getSize method, 552
 inheritance hierarchies of, 640
 isVisible method, 552
 repaint method, 556, 559
 setBackground/Foreground methods, 570, 573
 setBounds method, 546, 548, 552, 724
 setCursor method, 624
 setLocation method, 546, 548, 552
 setSize method, 552
 setVisible method, 546, 552, 951
 validate method, 651, 951
Components, 639
 displaying information in, 553
 labeling, 651–652
 realized, 951

Composite design pattern, 631
`CompoundInterest/CompoundInterest.java`, 122
Computations
 performance of, 56, 59
 truncated, 56
compute, `computeIfPresent/Absent` methods (*Map*), 501
Concrete collections, 472–496
Concrete methods, 222
Concurrent hash maps
 atomic updates in, 907–909
 buckets as trees in, 906
 bulk operations on, 909–911
 efficiency of, 906
 size of, 906
Concurrent modification detection, 479
Concurrent programming, 7, 839–952
 synchronization in, 862–897
Concurrent sets, 912
`ConcurrentHashMap` class, 905–907
 atomic updates in, 907–909
 compute, `computeIfXxx` methods, 908–909
 `forEach` method, 910–911
 `get` method, 908
 `keySet`, `newKeySet` methods, 912
 `mappingCount` method, 906
 `merge` method, 909
 organizing buckets as trees in, 906
 `put`, `putIfAbsent` methods, 908
 `reduce`, `reduceXxx` methods, 910–911
 `replace` method, 908
 `search`, `searchXxx` methods, 910–911
 vs. synchronization wrappers, 914
`ConcurrentLinkedQueue` class, 905, 907
`ConcurrentModificationException`, 478–479, 906, 914
`ConcurrentSkipListMap` class, 905–907
`ConcurrentSkipListSet` class, 905, 907
Condition interface, 878
 `await` method, 856, 893–895
 `awaitUninterruptibly` method, 893–895
 `signal`, `signalAll` methods, 890
 vs. synchronization methods, 880
Condition objects, 872–877
Condition variables, 872
Conditional statements, 90–94
`config` method (`Logger`), 390, 404
Configuration files, 794–800
Confirmation dialogs, 733
Console
 debugging applets in, 807
 printing messages to, 42–46
`Console` class
 reading passwords with, 80
 `readLine/Password` methods, 81
`console` method (`System`), 81
`ConsoleHandler` class, 394–399, 407
`ConsoleWindow` class, 770
const keyword, 56
Constants, 55–56
 documentation comments for, 196
 names of, 55
 public, 56, 159
 static, 159
`Constructor` class, 265
 `getDeclaringClass` method, 270
 `getModifiers` method, 265, 270
 `getName` method, 265, 270
 `getXxxTypes` methods, 270
 `newInstance` method, 265, 452
Constructor references, 321–322
Constructors, 149–151, 171–182
 calling another constructor in, 176
 defined, 136
 documentation comments for, 194
 field initialization in:
 default, 172–173
 explicit, 174
 final, 265
 initialization blocks in, 177–181
 names of, 136, 150
 no-argument, 173, 208, 801
 overloading, 172
 parameter names in, 175
 private, 265
 protected, 194
 public, 194, 265
 with `super` keyword, 207
`ConstructorTest/ConstructorTest.java`, 179
Consumer interface, 326
Consumer threads, 898
Container class, 639
 `add` method, 591, 595, 641
 `setLayout` method, 641
Containers, 639
contains method
 of *Collection*, 467–468, 479
 of *HashSet*, 487

- containsAll method (*Collection*), 467–468, 479
containsKey/Value methods (*Map*), 499
Content pane, 554
continue statement, 108
Control flow, 89–108
 block scope, 89–90
 breaking, 106–108
 conditional statements, 90–94
 loops, 94–99
 determinate, 99–103
 “for each,” 113–114
 multiple selections, 103–105
Controllers, 633
Conversion characters, 82–83
Cooperative scheduling, 856
Coordinated Universal Time (UTC), 139
copy method (*Collections*), 523
copyArea method (*Graphics*), 583, 586
copyOf method (*Arrays*), 115, 119, 276
copyOfRange method (*Arrays*), 119
CopyOnWriteArrayList class, 912, 914
CopyOnWriteArraySet class, 912
Core Java program examples, 23
Cornell, Gary, 1
Corruption of data, 862–868
cos method (*Math*), 58
Count of Monte Cristo, The (Dumas), 490, 944–946
Countdown latches, 936
CountDownLatch class, 935–936
Covariant return types, 429
create method
 of EventHandler, 598
 of PersistenceService, 831, 837
createCustomCursor method (*Toolkit*), 618, 623
createDialog method (*JColorChooser*), 770
createFont method (*Font*), 575
createScreenCapture method (*Robot*), 778
createTypeBorder methods (*BorderFactory*), 664–667
createXxxGroup methods (*GroupLayout*), 722
Ctrl+\\, for thread dump, 889
Ctrl+C, for program termination, 863, 875
Ctrl+O, Ctrl+S accelerators, 687
Ctrl+Shift+F1, in Swing, 770
Ctrl+Tab, in text fields, 729
current method (*ThreadLocalRandom*), 893
Current user, 794
currentThread method (*Thread*), 851–854
Cursor class, *getPredefinedCursor* method, 617
Cursor shapes, 618
Custom layout managers, 724–728
Customizations. *See* Preferences
cyclicBarrier class, 935–937
- D**
- D suffix (double numbers), 49
Daemon threads, 859
darker method (*Color*), 571
Data exchange, 746–752
Data fields
 initializing, 176–181
 public, 150
Data types, 47–53
 boolean type, 52
 casting between, 60–61
 char type, 50–51
 conversions between, 59–60, 219–221
 floating-point, 48–49
 integer, 47–48
Databases, closing connections to, 372
dataExchange/DataExchangeFrame.java, 748
dataExchange/PasswordChooser.java, 749
Date and time
 formatting output for, 83–84
 no built-in types for, 136
Date class, 140
 getDay/Month/Year methods (deprecated), 141
 toString method, 137
DateInterval class, 428
Deadlocks, 874, 889–892, 896
 breaking up, 893
 in GUI, 897
Debugging, 8, 409–414
applets, 807
AWT events, 771, 774–778
collections, 479
debuggers for, 409
generic types, 513
GUI programs, 367, 770–778
including class names in, 344
intermittent bugs, 69, 545, 952
messages for, 366
reflection for, 272
trapping program errors in a file for, 411
when running applications in terminal window, 25–26
DebugGraphics class, 771

- Decorator design pattern, 631
Decrement operators, 61–62
Deep copies, 308
`deepToString` method (`Arrays`), 122, 240
Default methods, 298–300
 resolving conflicts in, 300–302
Default packages, 185
`default` statement, 104, 298–300
`DefaultButtonModel` class, 636
`DefaultComboBoxModel` class, 669
Deferred execution, 325
`delay` method (`Robot`), 778
Delayed interface, 900
 `getDelay` method, 900, 903
`DelayQueue` class, 900, 903
Delegates, 280
`delete` method
 of `PersistenceService`, 838
 of `StringBuilder`, 78
Dependence, 133–135
Deprecated classes, 197
Deprecated methods, 141, 197, 412
Deprecated variables, 197
`@deprecated` comment (`javadoc`), 197
Deque interface, 494–495
 `addFirst/Last` methods, 494
 `getFirst/Last` methods, 495
 `offerFirst/Last` methods, 494
 `peekFirst/Last` methods, 495
 `pollFirst/Last` methods, 495
 `removeFirst/Last` methods, 495
Deques, 494–495
Derived classes. *See* Subclasses
`deriveFont` method (`Font`), 575, 581
Descender, descent (in typesetting), 576
`descendingIterator` method (`NavigableSet`), 493
Design patterns, 630–632
Design Patterns—Elements of Reusable Object-Oriented Software (Gamma et al.), 632
`destroy` method (`Applet`), 808
Determinate loops, 99–103
Development environments
 choosing, 23–26
 in terminal window, 25
 integrated, 26–30
Device errors, 359
`dialog/AboutDialog.java`, 744
`dialog/DialogFrame.java`, 743
Dialogs, 730–770
 accepting/canceling, 746
 centering, 304
 closing, 603–607, 688, 743, 746
 color choosers, 764–770
 confirmation, 733
 creating, 741–745
 data exchange in, 746–752
 default button in, 748
 displaying, 743
 document- and toolkit-modal, 742
 file, 752–764
 input, 733
 maximized, 603
 modal, 730–741
 modeless, 730, 742–743
 data exchange with, 747
 option, 731–741
 pop-up, 821
 root pane of, 748
 traversal order of, 729–730
Diamond syntax, 245
Dijkstra, Edsger, 935
`disjoint` method (`Collections`), 524
`divide` method (`BigDecimal`, `BigInteger`), 110–111
Division operator, 56
`do/while` loop, 96, 99
Doclets, 199
Documentation comments, 46, 194–199
 extracting, 198–199
 for fields, 196
 for methods, 195–196
 for packages, 198
 general, 196
 HTML markup in, 194
 hyperlinks in, 198
 inserting, 194–195
 links to other files in, 195
 overview, 198
Document-modal dialogs, 742
`doInBackground` method (`SwingWorker`), 944–945, 950
Do-nothing methods, 604
Double brace initialization, 344
Double buffering, 771
Double class
 `compare` method, 294
 converting from `double`, 252
 `hashCode` method, 237

- D**
- Double class (*continued*)
 - POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN constants, 49
 - double type, 48
 - arithmetic computations with, 56
 - converting to other numeric types, 59–60
 - DoubleAccumulator, DoubleAdder classes, 889
 - Double-precision numbers, 48–49
 - Doubly linked lists, 474
 - draw method (Graphics2D), 561
 - draw/DrawTest.java, 566
 - drawImage method (Graphics), 582, 585
 - Drawing with mouse, 616–624
 - drawString method (Graphics / Graphics2D), 581
 - Drop-down lists, 668
 - Dynamic binding, 209, 214–217
 - Dynamic languages, 8
- E**
- e (exponent), in numbers, 49
 - E
 - as type variable, 419
 - constant (Math), 58
 - Echo character, 652–653
 - Eclipse, 24, 26–30, 409
 - configuring projects in, 28
 - editing source files in, 29
 - error messages in, 29–30
 - imports in, 183
 - SWT toolkit, 543
 - ECMA-262 (JavaScript subset), 15
 - Eiffel programming language, multiple inheritance in, 297
 - element method
 - of BlockingQueue, 898–899
 - of Queue, 494
 - elements method (Hashtable, Vector), 530
 - Ellington, Duke, 539
 - Ellipse2D class, 560, 564–565
 - setFrameFromCenter method, 565
 - setFrameFromDiagonal method, 564
 - Ellipse2D.Double class, 569
 - Ellipses, 560, 564–565
 - bounding rectangles of, 563–565
 - constructing, 565
 - filling with color, 569
 - else statement, 92–93
 - else if statement, 93–94
- E**
- EmployeeTest/EmployeeTest.java, 147
 - emptyCollection methods (Collections), 515
 - EmptyStackException, 381, 383
 - Encapsulation, 131–132
 - benefits of, 153–156
 - protected instance fields and, 284
 - endsWith method (String), 72
 - ensureCapacity method (ArrayList), 246–247
 - entering method (Logger), 405
 - Enterprise Edition (Java EE), 11, 18
 - entrySet method (Map), 502–503
 - Enum class, 258–260
 - compareTo, ordinal methods, 260
 - toString, valueOf methods, 258, 260
 - enum keyword, 65
 - Enumerated types, 65
 - equality testing for, 258
 - in switch statement, 105
 - Enumeration interface, 460, 528–530
 - nextElement, hasMoreElements methods, 465, 528, 530
 - Enumeration maps/sets, 506
 - Enumerations, 258–260, 818
 - legacy, 528–530
 - EnumMap class, 506, 508
 - as a concrete collection type, 472
 - enums/EnumTest.java, 259
 - EnumSet class, 506
 - allOf, noneOf, of, range methods, 508
 - as a concrete collection type, 472
 - EOFException, 364
 - Epoch, 139
 - equals method, 302
 - for wrappers, 254
 - hashCode method and, 236–237
 - implementing, 233
 - inheritance and, 231–235
 - of Arrays, 120, 234
 - of Collection, 467
 - of Object, 229–235, 244, 512
 - of proxy classes, 355
 - of Set, 471
 - of String, 68, 72
 - redefining, 236–237
 - equals/Employee.java, 241
 - equals/EqualsTest.java, 240
 - equals/Manager.java, 243
 - equalsIgnoreCase method (String), 68, 72
 - Error class, 360

- Errors
 checking, in mutator methods, 154
 code, 359
 compilation, 29
 device, 359
 internal, 360, 363, 386
 messages for, 369
 NoClassDefFoundError, 26
 physical limitations, 359
 ThreadDeath, 857, 862, 896
 user input, 359
- Escape sequences, 50
- Event delegation model, 588
- Event dispatch thread, 545, 846, 897
 time-consuming tasks and, 939
- Event handling, 587–628
 defined, 587
 for asynchronous actions, 931
 semantic vs. low-level events, 626
 summary of, 626–628
- Event listeners, 588–589
 with a single method call, 597
 with lambda expressions, 595
- Event objects, 588
- Event procedures, 587
- Event sources, 588–589
- EventHandler class
 create method, 598
 creating listeners automatically with, 597
- EventObject class, 588, 624
 getActionCommand method, 624
 getSource method, 598, 624
- EventQueue class
 invokeAndWait method, 940, 943
 invokeLater method, 940, 943, 952
 isDispatchThread method, 943
- eventTracer/EventTracer.java, 772
- ExampleFileView class, 757
- Exception class, 360, 380
- Exception handlers, 263, 359
- Exception specification, 362
- Exceptions
 ArrayListOutOfBoundsException, 112, 361–363, 938
 ArrayStoreException, 431, 433, 441
 BadCastException, 451
 BrokenBarrierException, 937
 CancellationException, 945
 catching, 263–265, 363, 367–381
 multiple, 369–370
 changing type of, 370
 checked, 261–264, 361–364, 383
 ClassCastException, 220, 276, 295, 435, 441, 513
 classification of, 359–361
 CloneNotSupportedException, 310
 ConcurrentModificationException, 478–479, 906, 914
 creating classes for, 365–366
 documentation comments for, 196
 EmptyStackException, 381, 383
 EOFException, 364
 FileNotFoundException, 362–364, 438
 finally clause in, 372–376
 generics in, 437–439
 hierarchy of, 359, 383
 IllegalAccessException, 272
 IllegalStateException, 465, 469, 483, 494, 899
 InterruptedException, 841, 847, 851–854, 893–895, 915
 IOException, 88, 361, 364, 368, 375
 logging, 392, 400
 micromanaging, 381
 NoSuchElementException, 464, 469, 483, 494–495
 NullPointerException, 361, 383
 NumberFormatException, 383
 propagating, 368, 384
 rethrowing and chaining, 370, 410
 RuntimeException, 360, 383
 ServletException, 370
 squelching, 383
 stack trace for, 377–381
 “throw early, catch late,” 384
 throwing, 263–265, 364–365
 TimeoutException, 915
 tips for using, 381–384
 UnavailableServiceException, 830
 uncaught, 411, 857, 860–862
 unchecked, 264, 361–363, 383
 unexpected, 392, 400
 UnsupportedOperationException, 503, 510, 512, 514
 using type variables in, 437
 variables for, implicitly final, 370
 vs. simple tests, 381
 wrapping, 371
- Exchanger class, 935, 937
- Exchangers, 937
- .exe file extension, 783
- Executable JAR files, 782–783

- Executable path, 20
 execute method (*SwingWorker*), 945, 950
 Execution flow, tracing, 391
ExecutionException, 933
ExecutorCompletionService class, 927
 poll, submit, take methods, 928
Executors, 920–934
 groups of tasks, controlling, 927–928
 scheduled, 926
Executors class
 newCachedThreadPool method, 921, 925
 newFixedThreadPool method, 921, 925
 newScheduledThreadPool method, 921, 926
 newSingleThreadExecutor method, 921, 925
 newSingleThreadScheduledExecutor method, 921, 926
ExecutorService interface, 921–922
 invokeAny/All methods, 927–928
 shutdown method, 922, 925
 shutdownNow method, 922, 927
 submit method, 921, 925
 Exit codes, 45
 exit method (*System*), 45
 exiting method (*Logger*), 391, 405
 exp method (*Math*), 58
 Explicit parameters, 152–153
 exportXxx methods (*Preferences*), 795, 800
ExtendedService class, 830
 extends keyword, 204–228, 422–423
 External padding, 704
- F**
- f suffix (float numbers), 49
 Factorial functions, 378
 Factory methods, 161
 Fair locks, 872
 Fallthrough behavior, 105, 412
fdlibm (Freely Distributable Math Library), 59
Field class, 265
 get method, 276
 getDeclaringClass method, 270
 getModifiers method, 265, 270
 getName method, 265, 270
 getType method, 265
 set method, 276
 Field width, of numbers, 82
 Fields
 adding, in subclasses, 207
 default initialization of, 172–173
 documentation comments for, 194, 196
 final, 159, 218
 instance, 131, 150–153, 157, 174, 200
 private, 200, 206
 protected, 194, 228, 283
 public, 194, 196
 public static final, 296
 static, 158–159, 178, 185, 436
 volatile, 885–886
 File access warning, 831
 File dialogs, 752–764
 adding accessory components to, 757
fileChooser/FileIconView.java, 762
fileChooser/ImagePreviewer.java, 761
fileChooser/ImageViewerFrame.java, 759
FileContents class
 canRead/Write methods, 837
 getName method, 837
 getXxxStream methods, 830, 837
FileFilter class (Swing)
 accept method, 755, 764
 getDescription method, 755, 764
FileFilter interface (java.io package), 755
FileHandler class, 394–399, 407
 configuration parameters of, 396
FileNameExtensionFilter interface, 764
FileNotFoundException, 362–364, 438
FileOpenService class
 openFileDialog method, 830, 837
 openMultiFileDialog method, 837
 Files
 extensions of, 757
 filters for, 755–757
 MIME types of, 825
 names of, 25, 87
 opening/saving in GUI, 752–764
 reading, 87
 all words from, 376
 in a separate thread, 944
 writing, 87
FileSaveService class
 saveAsFileDialog method, 837
 saveFileDialog method, 830, 837
FileView class, 756
 getIcon, getName, getDescription, getTypeDescription methods, 756, 764
 isTraversable method, 756, 764

fill method
 of Arrays, 120
 of Collections, 523
 of Graphics2D, 569–570, 573

Filter interface, 398
 `isLoggable` method, 398, 408

final access modifier, 55, 217–218
 checking, 265
 for fields in interfaces, 296
 for instance fields, 157
 for methods in superclass, 295
 for shared fields, 886
 inner classes and, 339–342

finalize method, 181–182

finally clause, 372–376
 not completed normally, 412
 return statements in, 374
 unlock operation in, 869
 without catch, 373

Financial calculations, 49

fine, finer, finest methods (*Logger*), 390, 404

Firefox, 34

first method (*SortedSet*), 493

First Person, Inc., 10

firstKey method (*SortedMap*), 500

FirstSample/FirstSample.java, 46

Float class
 converting from float, 252
 hashCode method, 237
 POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN
 constants, 49

float type, 48
 converting to other numeric types, 59–60

Floating-point numbers, 48–49
 arithmetic computations with, 56
 equality of, 101
 formatting output for, 82–83
 rounding, 49, 60

Floating-point overflow, 57

floor method (*NavigableSet*), 493

floorMod method (*Math*), 57

Flow layout manager, 638

FlowLayout class, 641

flush method (*Handler*), 406

FocusAdapter class, 626

FocusEvent class, 626
 `isTemporary` method, 627

FocusListener interface, 626
 `focusGained/Lost` methods, 627

Font class, 574–581
 `createFont` method, 575
 `deriveFont` method, 575, 581
 `getFamily`, `getFontName`, `getName` methods, 580
 `getLineMetrics` method, 577, 580
 `getStringBounds` method, 576–577, 580

font/FontTest.java, 578

fontconfig.properties file, 575

FontMetrics class, `getFontRenderContext` method, 582

Fonts, 573–582
 checking availability of, 573
 face/family names of, 573
 logical names of, 574
 size of, 574–575
 styles of, 575
 typesetting properties of, 576

“for each” loop, 112–114
 for array lists, 249
 for collections, 464, 914
 for multidimensional arrays, 122

for loop, 99–103
 comma-separated lists of expressions in, 65
 defining variables inside, 101
 for collections, 464

foreach method
 of ConcurrentHashMap, 910–911
 of Map, 499

Foreground color, specifying, 570

Fork-join framework, 928

forkJoin/ForkJoinTest.java, 930

Format specifiers (`printf`), 82

format, formatTo methods (*String*), 83

Formattable interface, 83

Formatter class, methods of, 399, 408

forName method (*Class*), 261, 265

Frame class, 543
 `get/setExtendedState` method, 553
 `getIconImage` method, 553
 `getTitle` method, 553
 `is/setUndecorated` methods, 553
 `isResizable` method, 553
 `setIconImage` method, 546, 553
 `setResizable` method, 546, 553
 `setTitle` method, 546, 553

Frames
 closing by user, 545
 creating, 543–546

- Frames (*continued*)
 decorating, 546
 displaying:
 information in, 554–560
 text in, 557
 positioning, 546–554
 properties of, 549
 size of, 549–554
 frequency method (*Collections*), 524
 Full-screen mode, 550
Function interface, 326
 Functional interfaces, 318–319
 abstract methods in, 318
 annotating, 328
 conversion to, 318
 generic, 319
 using supertype bounds in, 447
@FunctionalInterface annotation, 328
 Functions. *See* Methods
Future interface, 927
 cancel method, 915, 920–921, 945
 get method, 915, 919, 921, 945
 isCancelled, isDone methods, 915, 920–921
future/FutureTest.java, 917
 Futures, 915–920
 combining multiple, 934
 completable, 931–934
 FutureTask class, 915–920
- G**
- Garbage collection, 68, 139
 hash maps and, 504
GB18030 standard, 51
General Public License (GPL), 14
Generic programming, 415–458
 classes in, 245, 418–420, 669
 extending/implementing other generic classes, 441
 no throwing or catching instances of, 436–437
 collection interfaces in, 525
 converting to raw types, 412, 441
 debugging, 513
 expressions in, 426
 in JVM, 425, 452–458
 inheritance rules for, 440–442
 legacy code and, 429
 methods in, 421–422, 427–429, 466–469
 not for arrays, 434–436
- reflection and, 450–458
 required skill levels for, 417
 static fields or methods and, 436
 type erasure in, 425–430, 434
 clashes after, 439–440
 type matching in, 452
 vs. arrays, 321
 vs. inheritance, 416–418
 wildcard types in, 442–450
GenericArrayType interface, 453
 getGenericComponentType method, 458
genericReflection/GenericReflectionTest.java, 454
get method
 of *Array*, 279
 of *ArrayList*, 247, 251
 of *BitSet*, 533
 of *ConcurrentHashMap*, 908
 of *Field*, 276
 of *Future*, 915, 919, 921, 945
 of *LinkedList*, 480
 of *List*, 470, 483
 of *LongAccumulator*, 888
 of *Map*, 469, 497, 499
 of *PersistenceService*, 838
 of *Preferences*, 795, 800
 of *ThreadLocal*, 893
 of *Vector*, 883
 getActionCommand method
 of *ActionEvent*, 598, 627
 of *ButtonModel*, 661, 663
 of *EventObject*, 624
 getActionMap method (*JComponent*), 615
 getActualTypeArguments method (*ParameterizedType*), 458
 getAdjustable, getAdjustmentType methods
 (*AdjustmentEvent*), 627
 getAncestorOfClass method (*SwingUtilities*), 747, 752
 getAndType methods (*AtomicType*), 887
 getApplet, getApplets methods (*AppletContext*), 818, 820
 getAppletContext method (*Applet*), 818–820
 getAppletInfo method (*Applet*), 816
 getAscent method (*LineMetrics*), 581
 getAudioClip method (*Class*), 784, 817
 getAutoCreateXxx methods (*GroupLayout*), 722
 getAvailableFontFamilyNames method
 (*GraphicsEnvironment*), 573
 getBackground method (*Component*), 573

getBoolean method (Array), 279
getBounds method (TypeVariable), 457
getByte method (Array), 279
getCause method (Throwable), 379
getCenterX/Y methods (RectangularShape), 563, 568
getChar method (Array), 279
getClass method
 always returning raw types, 431
 of Class, 261
 of Object, 244
getClassName method
 of LookAndFeelInfo, 603
 of StackTraceElement, 380
getClickCount method (MouseEvent), 616, 623, 627
getCodeBase method
 of Applet, 816–817
 of BasicService, 831, 836
getColor method
 of Graphics, 572
 of JColorChooser, 770
getColumns method (JTextField), 650
getComponentPopupMenu method (JComponent), 686
getComponentType method (Class), 277
getConstructor method (Class), 451
getConstructors method (Class), 266, 270
getContentPane method (JFrame), 559
getDataType methods (Preferences), 795, 800
getDay method (Date, deprecated), 141
getDayXxx methods (LocalDate), 141, 145
getDeclaredConstructor method (Class), 451
getDeclaredConstructors method (Class), 266, 270
getDeclaredField method (Class), 275
getDeclaredFields method (Class), 266, 270, 272, 275
getDeclaredMethods method (Class), 266, 270, 280
getDeclaringClass method (java.lang.reflect), 270
getDefaultScreenDevice method (GraphicsEnvironment), 774, 778
getDefaultToolkit method (Toolkit), 305, 549, 553
getDefaultUncaughtExceptionHandler method (Thread), 861
getDelay method (*Delayed*), 900, 903
getDescent method (LineMetrics), 581
getDescription method
 of FileFilter, 755, 764
 of FileView, 756, 764
getDocumentBase method (Applet), 816–817
getDouble method (Array), 279
getEnumConstants method (Class), 451
getExceptionTypes method (Constructor), 270
getExtendedState method (Frame), 553
getFamily method (Font), 580
getField method (Class), 275
getFields method (Class), 266, 270, 275
getFileName method (StackTraceElement), 380
getFilter method (Handler, Logger), 406
getFirst/Last methods
 of Deque, 495
 of LinkedList, 484
getFloat method (Array), 279
getFont method
 of Component, 651
 of Graphics, 581
getFontMetrics method (JComponent), 577, 582
getFontName method (Font), 580
getFontRenderContext method
 of FontMetrics, 582
 of Graphics2D, 576, 582
getForeground method (Component), 573
getFormatter method (Handler), 406
getGenericComponentType method (GenericArrayType), 458
getGenericParameterTypes, getGenericReturnType
 methods (Method), 457
getGenericXxx methods (Class), 457
getGlobal method (Logger), 389, 410
getHandlers method (Logger), 406
getHead method (Formatter), 399, 408
getHeight method
 of LineMetrics, 581
 of RectangularShape, 563, 568
getHonorsVisibility, getHorizontalGroup methods
 (GridLayout), 722
getIcon method
 of FileView, 756, 764
 of JLabel, 652
getIconImage method (Frame), 553
getImage method
 of Applet, 817
 of Class, 784
 of ImageIcon, 554, 582
getInheritsPopupMenu method (JComponent), 686
getInputMap method (JComponent), 612, 615
getInputStream method (FileContents), 830, 837
getInstalledLookAndFeels method (UIManager), 602
getInt method (Array), 279
getItem, getItemSelectable methods (ItemEvent), 627

- getSelectedItem method (`JComboBox`), 669
- getKey method (`Map.Entry`), 503
- getKeyStroke method (`KeyStroke`), 610, 615
- getKeyXxx methods (`KeyEvent`), 627
- getLargestPoolSize method (`ThreadPoolExecutor`), 926
- getLeading method (`LineMetrics`), 581
- getLength method (`Array`), 277, 279
- getLevel method
 - of `Handler`, 406
 - of `Logger`, 405
 - of `LogRecord`, 407
- getLineMetrics method (`Font`), 577, 580
- getLineNumber method (`StackTraceElement`), 380
- getLocalGraphicsEnvironment method
 - (`GraphicsEnvironment`), 774, 778
- getLogger method (`Logger`), 390, 404
- getLoggerName method (`LogRecord`), 407
- getLong method (`Array`), 279
- getLowerBounds method (`WildcardType`), 458
- getMaxX/Y methods (`RectangularShape`), 568
- getMessage method
 - of `LogRecord`, 407
 - of `Throwable`, 366
- getMethod method (`Class`), 280
- getMethodName method (`StackTraceElement`), 380
- getMethods method (`Class`), 266, 270
- getMillis method (`LogRecord`), 408
- getMinX/Y methods (`RectangularShape`), 568
- getModifiers method
 - of `ActionEvent`, 627
 - of `java.lang.reflect`, 265, 270
- getModifiersEx method (`InputEvent`), 617, 623
- getModifiersExText method (`InputEvent`), 623
- getMonth method (`Date`, *deprecated*), 141
- getMonthXxx methods (`LocalDate`), 141, 145
- getName method
 - of `Class`, 244, 261–262
 - of `FileContents`, 837
 - of `FileView`, 756, 764
 - of `Font`, 580
 - of `java.lang.reflect`, 265, 270
 - of `LookAndFeelInfo`, 603
 - of `TypeVariable`, 457
- getNames method (`PersistenceService`), 838
- getNewState, `getOldState` methods (`WindowEvent`), 607, 628
- getOppositeWindow method (`WindowEvent`), 628
- getOrDefault method (`Map`), 499
- getOutputStream method (`FileContents`), 830, 837
- getOwnerType method (`ParameterizedType`), 458
- getPaint method (`Graphics2D`), 573
- getParameter method (`Applet`), 810–811, 816
- getParameterInfo method (`Applet`), 816
- getParameters method (`LogRecord`), 407
- getParameterTypes method (`Method`), 270
- getParent method (`Logger`), 406
- getPassword method (`JPasswordField`), 653
- getPoint method (`MouseEvent`), 623, 627
- getPredefinedCursor method (`Cursor`), 617
- getPreferredSize method (`Component`), 557, 559
- getProperties method (`System`), 789, 793
- getProperty method
 - of `Properties`, 531, 789, 792
 - of `System`, 793
- getProxyClass method (`Proxy`), 355–356
- getRawType method (`ParameterizedType`), 458
- getResource, `getResourceAsStream` methods (`Class`), 784, 787
- getResourceBundle, `getResourceBundleName` methods (`LogRecord`), 407
- getReturnType method (`Method`), 270
- getRootPane method (`JComponent`), 748, 752
- getScreenSize method (`Toolkit`), 549, 553
- getScrollAmount method (`MouseWheelEvent`), 628
- getSelectedFile/files methods (`JFileChooser`), 754, 763
- getSelectedItem method (`JComboBox`), 669–671
- getSelectedObjects method (`ItemSelectable`), 661
- getSelection method (`ButtonGroup`), 661, 663
- getSequenceNumber method (`LogRecord`), 408
- getServiceNames method (`ServiceManager`), 836
- getShort method (`Array`), 279
- getSize method (`Component`), 552
- getSource method (`EventObject`), 598, 624
- getSourceXxxName methods (`LogRecord`), 408
- getStackTrace method (`Throwable`), 377, 379
- getState method
 - of `SwingWorker`, 950
 - of `Thread`, 858
- getStateChange method (`ItemEvent`), 627
- getStringBounds method (`Font`), 576–577, 580
- getSuperclass method (`Class`), 244, 451
- getSuppressed method (`Throwable`), 377, 380
- getTail method (`Formatter`), 399, 408
- Getter/setter pairs. *See* Properties
- getText method
 - of `JLabel`, 652

- of `JTextComponent`, 650
- `getThreadID` method (`LogRecord`), 408
- `getThrown` method (`LogRecord`), 408
- `getTime` method (`Calendar`), 218
- `getTitle` method (`Frame`), 553
- `getType` method (`Field`), 265
- `getTypeDescription` method (`FileView`), 756, 764
- `getTypeParameters` method (`Class`, `Method`), 457
- `getUncaughtExceptionHandler` method (`Thread`), 861
- `getUpperBounds` method (`WildcardType`), 458
- `getUseParentHandlers` method (`Logger`), 406
- `getValue` method
 - of `Action`, 608, 615
 - of `AdjustmentEvent`, 627
 - of `Map.Entry`, 503
- `getWheelRotation` method (`MouseWheelEvent`), 628
- `getWidth` method
 - of `Rectangle2D`, 563
 - of `RectangularShape`, 563, 568
- `getWindow` method (`WindowEvent`), 628
- `getX/Y` methods
 - of `MouseEvent`, 616, 623, 627
 - of `RectangularShape`, 568
- `getYear` method
 - of `Date` (deprecated), 141
 - of `LocalDate`, 141, 145
- GMT (Greenwich Mean Time), 139
- Goetz, Brian, 840, 885
- Gosling, James, 10–11
- `goto` statement, 89, 106
- Graphical User Interface (GUI), 537–586
 - automatic testing, 774–778
 - components of, 629–778
 - choice components, 657–678
 - dialog boxes, 730–770
 - text input, 648–656
 - toolbars, 694–696
 - tooltips, 696–699
 - traversal order of, 729–730
 - deadlocks in, 897
 - debugging, 367, 770–778
 - events in, 587
 - keyboard focus in, 611
 - layout of, 638–648, 699–730
 - multithreading for, 846–851
- Graphics class, 560, 582–586
 - `copyArea` method, 583, 586
 - `drawImage` method, 582, 585
 - `drawString` method, 581
- get/setFont methods, 581
- `getColor` method, 572
- `setColor` method, 570, 572
- Graphics editor applications, 616–624
- Graphics2D class, 560–569
 - `draw` method, 561
 - `drawString` method, 582
 - `fill` method, 569–570, 573
 - `getFontRenderContext` method, 576, 582
 - `getPaint` method, 573
 - `setPaint` method, 569, 573
- GraphicsDevice class, 550, 774
- GraphicsEnvironment class, 550
 - `getAvailableFontFamilyNames` method, 573
 - `getDefaultScreenDevice` method, 774, 778
 - `getLocalGraphicsEnvironment` method, 774, 778
- Green project, 10
- GregorianCalendar class, 142
 - `add` method, 142
 - constructors for, 140, 172
- Grid bag layout, 700–712
 - padding in, 704
- Grid layout, 644–648
 - `gridbag/FontFrame.java`, 707
 - `gridbag/GBC.java`, 709
- GridBagConstraints class, 703
 - `fill`, `anchor` parameters, 704, 712
 - `gridx/y`, `gridwidth/height` parameters, 703–706, 712
- helper class for, 706–712
- `insets` field, 704, 712
 - `ipadx/y` parameters, 712
 - `weightx/y` fields, 703, 712
- GridLayout class, 641, 644–648
- Group layout, 701, 713–723
 - `GroupLayout` class, 713–723
 - methods of, 722
 - `groupLayout/FontFrame.java`, 719
 - `GroupLayout.Group` class, 723
 - `GroupLayout.ParallelGroup` class, 723
 - `GroupLayout.SequentialGroup` class, 723
- GTK look-and-feel, 539–540
- GUI. *See* Graphical User Interface

H

- `handle` method (`CompletableFuture`), 933
- Handler class, 397
 - `close` method, 406
 - `flush` method, 406

- Handler class (*continued*)
 get/setFilter methods, 406
 get/setLevel methods, 406
 getFormatter method, 406
 publish method, 398, 406
 setFormatter method, 399, 406
- Handlers, 394–398
- Hansen, Per Brinch, 884
- “Has-a” relationship, 133–135
- hash method (`Objects`), 237
- Hash codes, 235–238, 485
 default, 235
 formatting output for, 83
- Hash collisions, 486
- Hash maps, 497
 concurrent, 905–907
 identity, 507–509
 linked, 504–506
 setting, 497
 vs. tree maps, 497
 weak, 504
- Hash sets, 485–489
 adding elements to, 490
 linked, 504–506
- Hash tables, 485
 legacy, 528
 load factor of, 486
 rehashing, 486
- hashCode method, 235–238
 equals method and, 236–237
 null-safe, 236
 of `Arrays`, 238
 of `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`, 237
 of `Object`, 237, 489
 of `Objects`, 236–237
 of proxy classes, 355
 of `Set`, 471
 of `String`, 485
- HashMap class, 497, 500
 as a concrete collection type, 472
- HashSet class, 464, 487–488
 add method, 487
 as a concrete collection type, 472
 contains method, 487
- Hashtable interface, 460, 528, 914–915
 as a concrete collection type, 472
 elements, keys methods, 530
 synchronized methods, 528
- hasMoreElements method (`Enumeration`), 465, 528, 530
- hasNext method
 of `Iterator`, 463, 465, 469
 of `Scanner`, 81
- hasNextType methods (`Scanner`), 81
- hasPrevious method (`ListIterator`), 476, 483
- headMap method
 of `NavigableMap`, 517
 of `SortedMap`, 511, 516
- headSet method (`NavigableSet`, `SortedSet`), 511, 516
- Heap, 495
 dumping, 413
- Height (in typesetting), 576
- Helper classes, 706–712
- Helper methods, 156, 448
- Hexadecimal numbers
 formatting output for, 82–83
 prefix for, 48
- higher method (`NavigableSet`), 493
- Hoare, Tony, 884
- Hold count, 870
- Holder types, 255
- HotJava browser, 11, 802
- Hotspot just-in-time compiler, 534
- HTML (HyperText Markup Language), 12–13
 applet element, 34, 805, 808–810
 param element, 810–816
 tables, 701
 target attribute, 820
 title element, 807
- HTML editors, 633
- |
- Icons
 associating with file extensions, 757
 in menu items, 682–683
 in sliders, 674
- Identity hash maps, 507–509
- identityHashCode method (`System`), 507, 509
- IdentityHashMap class, 507–509
 as a concrete collection type, 472
- IEEE 754 specification, 49, 59
- if statement, 90–94
- IFC (Internet Foundation Classes), 538
- IllegalAccessException, 272
- IllegalStateException, 465, 469, 483, 494, 899

image/ImageTest.java, 583
ImageIcon class, 550
 getImage method, 554, 582
Images
 accessing from applets, 816–817
 displaying, 582–586
ImageViewer/ImageViewer.java, 31
Immutable classes, 157
Implementations, 460
implements keyword, 290
Implicit parameters, 152–153
 none, in static methods, 160
 state of, 409
import statement, 183–184
 static, 185
importPreferences method (*Preferences*), 795, 800
Inconsistent state, 896
increment method (*LongAdder*), 888
Increment operators, 61–62
Incremental linking, 7
incrementAndGet method (*AtomicType*), 887
Index (in arrays), 111
indexOf method
 of *List*, 483
 of *String*, 73
indexOfSubList method (*Collections*), 524
info method (*Logger*), 389–390, 404
Information hiding. *See* Encapsulation
Inheritance, 133–135, 203–286
 design hints for, 283–286
 equality testing and, 231–235
 hierarchies of, 212–213
 multiple, 213, 297
 preventing, 217–218
 vs. type parameters, 416, 440–442
inheritance/Employee.java, 210
inheritance/Manager.java, 211
inheritance/ManagerTest.java, 210
init method (*Applet*), 807, 811
initCause method (*Throwable*), 379
Initialization blocks, 177–181
 static, 178
initialize method (*ThreadLocal*), 893
Inlining, 7, 218
Inner classes, 329–349
 accessing object state with, 331–334
 anonymous, 329, 342–345, 606
 applicability of, 335–338
 defined, 329
 local, 339
 private, 333
 static, 331, 346–349
 syntax of, 334–335
 vs. lambdas, 318
innerClass/InnerClassTest.java, 333
Input dialogs, 733
Input maps, 611–613
Input, reading, 79–81
InputEvent class
 getModifiersEx method, 617, 623
 getModifiersExText method, 623
InputTest/InputTest.java, 80
insert method
 of *JMenu*, 681
 of *JTextArea*, 951
 of *StringBuilder*, 78
insertItemAt method (*JComboBox*), 669, 671
insertSeparator method (*JMenu*), 681
Instance fields, 131
 final, 157
 initializing, 200
 explicit, 174
 not present in interfaces, 289, 296
 private, 150, 200
 protected, 283
 public, 150
 shadowing, 151, 175–176
 values of, 153–154
 volatile, 885–886
 vs. local variables, 151–152, 173
instanceof operator, 64, 220–221, 295
Instances, 131
 creating on the fly, 263
int type, 47
 converting to other numeric types, 59–60
 fixed size for, 6
 platform-independence of, 48
 random number generator for, 181
Integer class
 compare method, 294, 318
 converting from *int*, 252
 hashCode method, 237
 intValue method, 255
 parseInt method, 254, 256, 811
 toString method, 256
 valueOf method, 256

- Integer types, 47–48
arithmetic computations with, 56
arrays of, 240
formatting output for, 82
no unsigned types in Java, 48
Integrated Development Environment (IDE),
20, 26–30
Inter-applet communication, 810, 818
interface keyword, 288
Interface types, 462
Interface variables, 295
Interfaces, 288–305
abstract classes and, 297
callbacks and, 302–305
constants in, 296
defined, 288
documentation comments for, 194
evolution of, 299
extending, 295
for custom algorithms, 526–528
functional, 318–319
listener, 588
marker, 309
methods in, 298
clashes between, 300–302
do-nothing, 604
nonabstract, 318
no instance fields in, 289, 296
properties of, 295–296
public, 194
tagging, 309, 426, 471
vs. implementations, 460–463
interfaces/Employee.java, 293
interfaces/EmployeeSortTest.java, 292
Intermittent bugs, 69, 545, 952
Internal errors, 360, 363, 386
Internal padding, 704
Internationalization. *See* Localization
Internet Explorer
applets in, 810
Java in, 9
limited Java support in, 803
security of, 15
Interpreted languages, 14
Interpreter, 7
interrupt method (*Thread*), 851–854
interrupted method (*Thread*), 853–854
InterruptedException, 841, 847, 851–854, 893–895,
915
IntHolder class, 255
Intrinsic locks, 878, 884
Introduction to Algorithms (Cormen et al.),
489
intValue method (*Integer*), 255
Invocation handlers, 350
InvocationHandler interface, 350, 355
invoke method
of *InvocationHandler*, 350, 355
of *Method*, 279–283
invokeAndWait method (*EventQueue*), 940, 943
invokeAny/All methods (*ExecutorService*),
927–928
invokeLater method (*EventQueue*), 940, 943, 952
IOException, 88, 361, 364, 368, 375
“Is-a” relationship, 133–135, 213, 284
isAbstract method (*Modifier*), 271
isAccessible method (*AccessibleObject*), 275
isActionKey method (*KeyEvent*), 627
isCancelled, isDone methods (*Future*), 915,
920–921
isDefaultButton method (*JButton*), 752
isDispatchThread method (*EventQueue*), 943
isEditable method
of *JComboBox*, 671
of *JTextComponent*, 648
isEmpty method (*Collection*), 299, 467–468
isEnabled method (*Action*), 608, 615
isFinal method (*Modifier*), 265, 271
isInterface method (*Modifier*), 271
isInterrupted method (*Thread*), 851–854
isJavaIdentifierXxx methods (*Character*), 53
isLocationByPlatform method (*Window*), 552
isLoggable method (*Filter*), 398, 408
isNaN method (*Double*), 49
isNative method (*Modifier*), 271
isNativeMethod method (*StackTraceElement*), 381
ISO 8859–1 standard, 51
isPopupTrigger method
of *JPopupMenu*, 685
of *MouseEvent*, 686
isPrivate method (*Modifier*), 265, 271
isProtected method (*Modifier*), 271
isProxyClass method (*Proxy*), 355–356
isPublic method (*Modifier*), 265, 271
isResizable method (*Frame*), 553
isSelected method
of *AbstractButton*, 684
of *JCheckBox*, 658–659

isStatic, isStrict, isSynchronized methods (Modifier), 271
isTemporary method (FocusEvent), 627
isTraversable method (FileView), 756, 764
isUndecorated method (Frame), 553
isVisible method (Component), 552
isVolatile method (Modifier), 271
isWebBrowserSupported method (*BasicService*), 836
ItemEvent class, 626
 getItem, getItemSelectable, getStateChange methods, 627
ItemListener interface, 626
 itemStateChanged method, 627
ItemSelectable interface, getSelectedObjects method, 661
Iterable interface, 113
iterator method
 of *Collection*, 463, 468
 of ServiceLoader, 802
Iterator interface, 463–466
 “for each” loop, 464
 generic, 466
 hasNext, next, remove methods, 463, 465, 469
Iterators, 463–466
 being between elements, 465
 weakly consistent, 906
IzPack utility, 783

J

J# programming language, 8
J++ programming language, 8
 delegates in, 280
JApplet class, 803–808
Jar Bundler utility, 783
JAR files, 190, 780–787
 creating, 780
 digitally signed, 822–824
 dropping in *jre/lib/ext* directory, 193
 executable, 782–783
 manifest of, 781–782
 resources and, 783–787
 sealing, 787
jar program, 780
 command-line options of, 781–782
Java programming language
 architecture-neutral object file format of, 5
 as a programming platform, 1–2

available under GPL, 14
basic syntax of, 42–46, 145
calling by value in, 165
case-sensitivity of, 26, 42, 53–56, 528
communicating with JavaScript, 809
compiling/launching from the command line, 24
design of, 2–8
documentation for, 23
dynamic, 8
dynamic binding in, 209, 214–217
garbage collection in, 68, 139
history of, 10–12
interpreter in, 7
libraries in, 12–13
 installing, 22–23
misconceptions about, 13–15
multithreading in, 7, 839–952
networking capabilities of, 4
no multiple inheritance in, 297
no operator overloading in, 109
no unsigned types in, 48
performance of, 7, 14, 534
portability of, 6, 13, 56
reliability of, 4
reserved words in, 43, 53, 56
security of, 4–5, 14, 820–822
simplicity of, 3, 315
strongly typed, 47, 291
versions of, 11–12, 538, 700
 vs. C++, 3
Java 2 (J2), 18
Java 2D library, 560–569
 floating-point coordinates in, 561
Java bug parade, 44, 393
Java collections framework (JCF), 459–536
 algorithms in, 517–528
 converting between collections and arrays in, 525–526
 interfaces in, 469–471
 legacy classes in, 528–536
 operations in:
 bulk, 524–525
 optional, 514
 separating interfaces from
 implementations in, 460–463
 views and wrappers in, 509–517
 vs. traditional collections libraries, 465
Java Concurrency in Practice (Goetz), 840

- Java Development Kit (JDK), 5, 17–39
 applet viewer, 805–806
 documentation in, 74–77, 612
 downloading, 18–20
 fonts shipped with, 574
 installation of, 18–23
 default, 780
 setting up, 20–22
 .java file extension, 43
Java Language Specification, 43
Java look-and-feel, 611
Java Memory Model and Thread Specification, 885
Java Network Launch Protocol (JNLP), 824–838
Java Plug-in, 802–824
 control panel of, 827
 enabling, 34
 installing, 803
 Java console in, 807
 restrictiveness of, 9, 822
 running local applets in, 806
 signed code in, 822–824
java program, 25
 command-line options of, 385–386
Java Runtime Environment (JRE), 18
Java SE 8, 12, 18
 adding static methods to interfaces in, 298–299, 523
 completable futures in, 931
 concurrent hash maps in, 906–911
 constructor expressions in, 433
 hash tables in, 486
Java Plug-in for, 34
lambda expressions in, 314–329, 464, 887
LongAdder, LongAccumulator classes in, 888
parallelized operations on arrays in, 912
Java virtual machine (JVM), 6
 generics in, 425, 452–458
 launching, 25
 monitoring and managing applications in, 412
 optimizing execution in, 391
 precomputing method tables in, 216
 security vulnerabilities in, 803
 thread priority levels in, 859
 truncating arithmetic computations in, 56
 watching class loading in, 411
Java Virtual Machine Specification, 44
Java Web Start, 824–838
 launching, 826
 printing in, 832
 security of, 829
java.awt.Applet API, 807–808, 816–817, 820
java.awt.AppletContext API, 820
java.awt.BorderLayout API, 644
java.awt.Color API, 572
java.awt.Component API, 552, 559, 573, 624, 651, 724
java.awt.Container API, 595, 641
java.awt.event.ActionEvent API, 598
java.awt.event.InputEvent API, 623
java.awt.event.MouseEvent API, 623, 686
java.awt.event.WindowEvent API, 607
java.awt.event.WindowListener API, 606
java.awt.event.WindowStateListener API, 607
java.awt.EventQueue API, 943
java.awt.FlowLayout API, 641
java.awt.Font API, 580–581
java.awt.font.LineMetrics API, 581
java.awt.FontMetrics API, 582
java.awt.Frame API, 553
java.awt.geom.Ellipse2D.Double API, 569
java.awt.geom.Line2D.Double API, 569
java.awt.geom.Point2D.Double API, 569
java.awt.geom.Rectangle2D.Double API, 568
java.awt.geom.Rectangle2D.Float API, 569
java.awt.geom.RectangularShape API, 568
java.awt.Graphics API, 572, 581, 585–586
java.awt.Graphics2D API, 573, 582
java.awt.GraphicsEnvironment API, 778
java.awt.GridLayout API, 648
java.awt.LayoutManager API, 728
java.awt.Robot API, 778
java.awt.Toolkit API, 305, 553, 623
java.awt.Window API, 552, 560
java.beans.EventHandler API, 598
java.io.Console API, 81
java.io.PrintWriter API, 89
java.lang.Boolean API, 237
java.lang.Byte API, 237
java.lang.Character API, 237
java.lang.Class API, 244, 265, 270, 275, 451, 457, 787
java.lang.ClassLoader API, 388
java.lang.Comparable API, 293
java.lang.Double API, 237, 294

java.lang.Enum API, 260
java.lang.Exception API, 380
java.lang.Float API, 237
java.lang.Integer API, 237, 255–256, 294
java.lang.Long API, 237
java.lang.Object API, 132, 237, 244, 489, 881–882
java.lang.Objects API, 237
java.lang.reflect package, 265, 276
java.lang.reflect.AccessibleObject API, 275
java.lang.reflect.Array API, 279
java.lang.reflect.Constructor API, 265, 270, 452
java.lang.reflect.Field API, 270, 276
java.lang.reflect.GenericArrayType API, 458
java.lang.reflect.InvocationHandler API, 355
java.lang.reflect.Method API, 270, 283, 457
java.lang.reflect.Modifier API, 271
java.lang.reflect.ParameterizedType API, 458
java.lang.reflect.Proxy API, 356
java.lang.reflect.TypeVariable API, 457
java.lang.reflect.WildcardType API, 458
java.lang.Runnable API, 851
java.lang.RuntimeException API, 380
java.lang.Short API, 237
java.lang.StackTraceElement API, 380–381
java.lang.String API, 72–73
java.lang.StringBuilder API, 78
java.lang.System API, 81, 509, 793
java.lang.Thread API, 846, 851, 854, 858–861
java.lang.Thread.UncaughtExceptionHandler API, 861
java.lang.ThreadGroup API, 862
java.lang.ThreadLocal API, 893
java.lang.Throwable API, 265, 366, 379–380
java.math.BigDecimal API, 111
java.math.BigInteger API, 110
java.nio.file.Paths API, 89
java.text.NumberFormat API, 256
java.time.LocalDate API, 145
java.util.ArrayDeque API, 495
java.util.ArrayList API, 247, 251
java.util.Arrays API, 119–120, 234, 238, 294, 516
java.util.BitSet API, 533
java.util.Collection API, 468–469, 524
java.util.Collections API, 514–515, 520–524, 915
java.util.Comparator API, 521
java.util.concurrent package, 868
canned functionality classes in, 934–937
efficient collections in, 905–907
java.util.concurrent.ArrayBlockingQueue API, 903
java.util.concurrent.atomic package, 886
java.util.concurrent.BlockingDeque API, 904–905
java.util.concurrent.BlockingQueue API, 904
java.util.concurrent.Callable API, 919
java.util.concurrent.ConcurrentHashMap API, 907
java.util.concurrent.ConcurrentLinkedQueue API, 907
java.util.concurrent.ConcurrentSkipListMap API, 907
java.util.concurrent.ConcurrentSkipListSet API, 907
java.util.concurrent.Delayed API, 903
java.util.concurrent.DelayQueue API, 903
java.util.concurrent.ExecutorCompletionService API, 928
java.util.concurrent.Executors API, 925–926
java.util.concurrent.ExecutorService API, 925, 928
java.util.concurrent.Future API, 919–920
java.util.concurrent.FutureTask API, 920
java.util.concurrent.LinkedBlockingQueue API, 903
java.util.concurrent.locks.Condition API, 877, 895
java.util.concurrent.locks.Lock API, 871, 877, 894
java.util.concurrent.locks.ReentrantLock API, 872
java.util.concurrent.locks.ReentrantReadWriteLock API, 896
java.util.concurrent.PriorityBlockingQueue API, 904
java.util.concurrent.ScheduledExecutorService API, 926
java.util.concurrent.ThreadLocalRandom API, 893
java.util.concurrent.ThreadPoolExecutor API, 926
java.util.concurrent.TransferQueue API, 905
java.util.Deque API, 494–495
java.util Enumeration API, 530
java.util EnumMap API, 508
java.util EnumSet API, 508
java.util EventObject API, 598
java.util function API, 319
java.util HashMap API, 500
java.util HashSet API, 488
java.util Hashtable API, 530
java.util IdentityHashMap API, 509
java.util Iterator API, 469
java.util LinkedHashMap API, 508
java.util LinkedHashSet API, 507
java.util LinkedList API, 484
java.util List API, 482–483, 516, 521, 524
java.util ListIterator API, 483
java.util.logging.ConsoleHandler API, 407
java.util.logging.FileHandler API, 407
java.util.logging.Filter API, 408

- `java.util.logging.Formatter` API, 408
`java.util.logging.Handler` API, 406
`java.util.logging.Logger` API, 404–406
`java.util.logging.LogRecord` API, 407–408
`java.util.Map` API, 499, 501–503
`java.util.Map.Entry` API, 503
`java.util.NavigableMap` API, 517
`java.util.NavigableSet` API, 493, 516
`java.util.Objects` API, 235
`java.util.prefs.Preferences` API, 799–800
`java.util.PriorityQueue` API, 496
`java.util.Properties` API, 531, 792–793
`java.util.Queue` API, 494
`java.util.Random` API, 181
`java.util.Scanner` API, 81, 89
`java.util.ServiceLoader` API, 802
`java.util.SortedMap` API, 500, 516
`java.util.SortedSet` API, 493, 516
`java.util.Stack` API, 532
`java.util.TreeMap` API, 500
`java.util.TreeSet` API, 493
`java.util.Vector` API, 530
`java.util.WeakHashMap` API, 507
JavaBeans, 260, 758, 813
javac program, 25
 - current directory in, 192
javadoc program, 194–199
 - command-line options of, 199
 - comments in:
 - class, 194–198
 - extracting, 198–199
 - field, 194, 196
 - general, 196
 - method, 194–195, 198
 - overview, 198
 - package, 194, 198
 - redeclaring `Object` methods for, 318
HTML markup in, 194
hyperlinks in, 198
links to other files in, 195
online documentation of, 199
JavaFX, 543
javap program, 336
JavaScript, 15
 - accessing applets from, 810
 - communicating with Java, 809
javaws program, 828
`javaws.jar` file, 830
`javax.jnlp.BasicService` API, 836
`javax.jnlp.FileContents` API, 837
`javax.jnlp.FileOpenService` API, 837
`javax.jnlp.FileSaveService` API, 837
`javax.jnlp.PersistenceService` API, 837–838
`javax.jnlp.ServiceManager` API, 836
`javax.swing` package, 545
`javax.swing.AbstractAction` API, 683
`javax.swing.AbstractButton` API, 663, 681–684, 688
`javax.swing.Action` API, 615
`javax.swing.border.LineBorder` API, 668
`javax.swing.border.SoftBevelBorder` API, 667
`javax.swing.BorderFactory` API, 666–667
`javax.swing.ButtonGroup` API, 663
`javax.swing.ButtonModel` API, 663
`javax.swing.event` package, 627
`javax.swing.event.MenuListener` API, 690
`javax.swing.filechooser.FileFilter` API, 764
`javax.swing.filechooser.FileNameExtensionFilter` API, 764
`javax.swing.filechooser.FileView` API, 764
`javax.swing.GroupLayout` API, 722
`javax.swing.GroupLayout.Group` API, 723
`javax.swing.GroupLayout.ParallelGroup` API, 723
`javax.swing.GroupLayout.SequentialGroup` API, 723
`javax.swing.ImageIcon` API, 554
`javax.swing.JButton` API, 595, 752
`javax.swing.JCheckBox` API, 659
`javax.swing.JCheckBoxMenuItem` API, 684
`javax.swing.JColorChooser` API, 770
`javax.swing.JComboBox` API, 671
`javax.swing.JComponent` API, 560, 582, 615, 650, 668, 686, 699, 752
`javax.swing.JDialog` API, 745
`javax.swing.JFileChooser` API, 762–763
`javax.swing.JFrame` API, 559, 682
`javax.swing.JLabel` API, 652
`javax.swing.JMenu` API, 681
`javax.swing.JMenuItem` API, 681–682, 688, 690
`javax.swing.JOptionPane` API, 304, 739–741
`javax.swing.JPasswordField` API, 653
`javax.swing.JPopupMenu` API, 685
`javax.swing.JRadioButton` API, 663
`javax.swing.JRadioButtonMenuItem` API, 684
`javax.swing.JRootPane` API, 752
`javax.swing.JScrollPane` API, 656
`javax.swing.JSlider` API, 678
`javax.swing.JTextArea` API, 656
`javax.swing.JTextField` API, 650
`javax.swing.JToolBar` API, 699

javax.swing.KeyStroke API, 615
javax.swing.SwingUtilities API, 752
javax.swing.SwingWorker API, 950
javax.swing.text.JTextComponent API, 648
javax.swing.Timer API, 305
javax.swing.UIManager API, 602
javax.swing.UIManager.LookAndFeelInfo API, 603
JButton class, 591, 595, 610, 636–638
 isDefaultButton method, 752
JCheckBox class, 657–659
 isSelected method, 658–659
 setSelected method, 657, 659
JCheckBoxMenuItem class, 683–684
JColorChooser class, 764–770
 methods of, 770
 JComboBox class, 627, 668–671
 addItem method, 669–671
 getItemAt method, 669
 getSelectedItem method, 669–671
 insertItemAt method, 669, 671
 isEditable method, 671
 removeXxx methods, 669, 671
 setEditable method, 669, 671
 setModel method, 669
JComponent class, 554
 action maps, 612
 get/setComponentPopupMenu methods, 685–686
 get/setInheritsPopupMenu methods, 685–686
 getActionMap method, 615
 getFontMetrics method, 577, 582
 getInputMap method, 612, 615
 getRootPane method, 748, 752
 input maps, 611–613
 paintComponent method, 554–556, 560, 577,
 583
 repaint method, 951
 revalidate method, 649–650, 951
 setBorder method, 664, 668
 setDebugGraphicsOptions method, 771
 setFont method, 650
 setSelectionStart/End methods, 952
 setToolTipText method, 699
jconsole program, 393, 412, 771, 889
jcontrol program, 807
JDialog class, 741–745
 setDefaultCloseOperation method, 743, 807
 setVisible method, 743, 746, 807
JDK. *See* Java Development Kit
JEditorPane class, 654

JFC (Java Foundation Classes), 539
JFileChooser class, 752–764
 addChoosableFileFilter method, 763
 getSelectedFile/Files methods, 754, 763
 resetChoosableFilters method, 756, 763
 setAcceptAllFileFilterUsed method, 756, 763
 setAccessory method, 763
 setCurrentDirectory method, 754, 762
 setFileFilter method, 755, 763
 setSelectionMode method, 754, 763
 setFileView method, 756–757, 763
 setMultiSelectionEnabled method, 754, 763
 setSelectedFile/Files methods, 754, 763
 showDialog, showXXXDialog methods, 747, 752,
 754, 763
JFrame class, 543–547, 640
 add method, 555, 559
 getContentPane method, 559
 internal structure of, 554–555
 setJMenuBar method, 679, 682
JLabel class, 651–652, 757
 getIcon, getText methods, 652
 setIcon, setText methods, 651–652
JList class, 670
jmap program, 413
JMenu class
 add, addSeparator methods, 679, 681
 insert, insertSeparator methods, 681
 remove method, 681
JMenuBar class, 679–682
JMenuItem class, 681–682
 setAccelerator method, 687–688
 setEnabled method, 689–690
 setIcon method, 682
Jmol applet, 9
JNLP API, 829–838
 compiling programs with, 830
join method (Thread), 73, 856–858
JOptionPane class, 730–741
 message types, 731
 showConfirmDialog method, 731–732, 739
 showInputDialog method, 731–732, 740
 showInternalConfirmDialog method, 739
 showInternalInputDialog method, 741
 showInternalMessageDialog method, 739
 showInternalOptionDialog method, 740
 showMessageDialog method, 304, 731–732, 739
 showOptionDialog method, 731–732, 739–740
 JPanel class, 558, 638, 842

- JPasswordField class, 652–653
 getPassword, setEchoChar methods, 653
- JPopupMenu class, 684–686
 isPopupTrigger, show methods, 685
- JRadioButton class, 660–663
- JRadioButtonMenuItem class, 684
- JRootPane class, setDefaultButton method, 748, 752
- JScrollbar class, 627
- JScrollPane class, 656
- JSlider class, 672–678
 setInverted method, 674, 678
 setLabelTable method, 429, 673, 678
 setPaintLabels method, 673, 678
 setPaintTicks method, 673, 678
 setPaintTrack method, 674, 678
 setSnapToTicks method, 673, 678
 setXxxTickSpacing methods, 678
- JTextArea class, 653–654
 append method, 656, 951
 insert method, 951
 replaceRange method, 951
 setColumns, setRows methods, 654, 656
 setLineWrap method, 654, 656
 setTabSize method, 656
 setWrapStyleWord method, 656
- JTextComponent class
 getText method, 650
 is/setEditable methods, 648
 setText method, 648, 650, 951
- JTextField class, 627, 649–651
 getColumns method, 650
 setColumns method, 649–650
- JToolBar class, 695–696
 add, addSeparator methods, 695–699
- JUnit framework, 410
- Just-in-time compiler, 6–7, 14, 153, 218, 413, 534
- JVM. *See* Java virtual machine
- K**
- K type variable, 419
- Key/value pairs. *See* Properties
- KeyAdapter class, 626
- Keyboard
 associating with actions, 610
 focus of, 611
 mnemonics for, 686–688
- Keyboard focus, 729
- KeyEvent class, 626
 getKeyXxx, isActionKey methods, 627
- KeyListener interface, 626
 keyXxx methods, 627
- keyPress/Release methods (*Robot*), 778
- keys method
 of Hashtable, 530
 of Preferences, 795, 799
- keySet method
 of ConcurrentHashMap, 912
 of Map, 502–503
- KeyStroke class, getKeyStroke method, 610, 615
- Knuth, Donald, 106
- KOI-8 standard, 51
- L**
- L suffix (*long* integers), 48
- Labeled break statement, 106
- Labels
 for components, 651–652
 for slider ticks, 673
- Lambda expressions, 314–329
 accessing variables in, 322–324
 atomic updates with, 887
 capturing values by, 323
 for event listeners, 595
 functional interfaces and, 318
 method references and, 320
 no assigning to a variable of type Object, 319
- parameter types of, 316
- processing, 324–328
- result type of, 316
- scope of, 324
- syntax of, 315–317
 this keyword in, 324
 vs. inner classes, 318
- Lambda/LambdaTest.java, 317
- Langer, Angelika, 458
- last method (*SortedSet*), 493
- lastIndexOf method
 of List, 483
 of String, 73
- lastIndexOfSubList method (*Collections*), 524
- lastKey method (*SortedMap*), 500
- Launch4J utility, 783
- Layout management, 638–648
 absolute positioning, 723
 border, 641–644

- box, 700
- custom, 724–728
- flow, 638
- grid, 644–648
- grid bag, 700–712
- group, 701, 713–723
- sophisticated, 699–730
- spring, 700
- LayoutManager* interface
 - designing custom, 724–728
 - methods of, 728
- LayoutManager2* interface, 725
- Leading (in typesetting), 576
- Legacy code and generics, 429–430
- Legacy collections, 528–536
 - bit sets, 532–536
 - enumerations, 528–530
 - hash tables, 528
 - property maps, 530–531
 - stacks, 531
- `length` method
 - of arrays, 112
 - of `BitSet`, 533
 - of `String`, 69–70, 73
 - of `StringBuilder`, 78
- Lightweight collection wrappers, 509–510
- `Line2D` class, 560, 565
- `Line2D.Double` class, 569
- `LineBorder` class, 665, 668
- Linefeed, escape sequence for, 50
- `LineMetrics` class, 577
 - `getXxx` methods, 581
- Lines, 560
 - constructing, 565
- @link comment (javadoc), 198
- Linked hash maps/sets, 504–506
- Linked lists, 474–484
 - concurrent modifications of, 479
 - doubly linked, 474
 - printing, 481
 - random access in, 479, 517
 - removing elements from, 475
- `LinkedBlockingDeque` class, 903
- `LinkedBlockingQueue` class, 899
- `LinkedHashMap` class, 504–508
 - access vs. insertion order, 505
 - as a concrete collection type, 472
 - `removeEldestEntry` method, 506, 508
- `LinkedHashSet` class, 504–507
 - as a concrete collection type, 472
- `LinkedList` class, 462, 476, 479, 494
 - `addFirst/Last`, `getFirst/Last` methods, 484
 - as a concrete collection type, 472
 - `get` method, 480
 - `listIterator` method, 476
 - `next/previousIndex` methods, 480
 - `removeAll` method, 480
 - `removeFirst/Last` methods, 484
- `linkedList/LinkedListTest.java`, 481
- `linkSize` method (`GroupLayout`), 722
- Linux
 - debugging applets in, 807
 - Eclipse versions for, 27
 - JDK versions for, 18
 - no thread priorities in Oracle JVM for, 859
 - pop-up trigger in, 685
 - running applets in, 34–35
 - setting paths in, 20, 191–193
 - setting up JDK in, 20
 - troubleshooting Java programs in, 26
- List* interface, 470, 509
 - `add` method, 470, 482
 - `addAll` method, 482
 - `get`, `set` methods, 470, 483
 - `indexOf`, `lastIndexOf` methods, 483
 - `listIterator` method, 482
 - `remove` method, 470, 483
 - `replaceAll` method, 524
 - `sort` method, 521
 - `subList` method, 510, 516
- Listener interfaces, 588
- Listener objects, 588
- Listeners. *See* Action listeners, Event listeners, Window listeners
- ListIterator* interface, 479
 - `add` method, 470, 476–478, 483
 - `hasPrevious` method, 476, 483
 - `next/previousIndex` methods, 483
 - `previous` method, 476, 483
 - `remove` method, 478
 - `set` method, 478, 483
- `listIterator` method
 - of `LinkedList`, 476
 - of `List`, 482
- Lists, 470
 - modifiable/resizable, 520

- load method
 - of Properties, 531, 788, 793
 - of ServiceLoader, 802
- Local inner classes, 339
 - accessing final variables from outer methods in, 339–342
- Local variables
 - annotating, 430
 - vs. instance fields, 151–152, 173
- LocalDate class, 139–141
 - extending, 285
 - getXxx methods, 141, 145
 - minusDays method, 145
 - now, of methods, 140, 145
 - plusDays method, 141, 145
 - processing arrays of, 446
- Locales, 393
- Localization, 136, 393–394, 784–785
- Lock interface, 878
 - await method, 873–877
 - lock method, 871, 893–895
 - lockInterruptibly method, 893–895
 - newCondition method, 873, 877
 - signal method, 875–877
 - signalAll method, 874–877
 - tryLock method, 856, 893–895
 - unlock method, 869, 871
 - vs. synchronization methods, 880
- Lock objects, 868–872
 - client-side, 883
 - deadlocks, 874, 889–893, 896
 - fair, 872
 - hold count for, 870
 - inconsistent state and, 896
 - intrinsic, 878, 884
 - not with try-with-resources statement, 869
 - read/write, 895–896
 - reentrant, 870
 - testing and timeouts, 893–895
- Locks
 - condition objects for, 872–877
 - in synchronized blocks, 882–883
- log, log10 methods (`Math`), 58
- Logarithms, 58
- Logger class
 - add/removeHandler methods, 406
 - entering, exiting methods, 391, 405
- get/setFilter methods, 398, 406
- get/setParent methods, 406
- get/setUseParentHandlers methods, 406
- getGlobal method, 389, 410
- getHandlers method, 406
- getLevel method, 405
- getLogger method, 390, 404
- info method, 389
- log method, 390, 392, 405
- logp method, 391, 405
- logrb method, 405
- setLevel method, 389, 405
- severe, warning, info, config, fine, finer, finest methods, 390, 404
- throwing method, 392, 405
- Loggers
 - configuring, 392–393
 - default, 389, 391
 - hierarchical names of, 390
 - writing your own, 390–392
- Logging, 389–408
 - advanced, 390–392
 - basic, 389
 - file pattern variables for, 396
 - file rotation for, 397
 - filters for, 398
 - formatters for, 399
 - handlers for, 394–398
 - configuring, 396
 - including class names in, 344
 - levels of, 390–391
 - changing, 392–393
 - localization of, 393–394
 - messages for, 240
 - recipe for, 399–408
 - resource bundles and, 393–394
- Logging proxy, 410
- logging/LoggingImageViewer.java, 400
- logging.properties file, 392–393
- Logical conditions, 52
- Logical “and,” “or,” 62
- LogManager class, 393
 - readConfiguration method, 392
- LogRecord class
 - getLevel method, 407
 - getLoggerName method, 407
 - getMessage method, 407
 - getMillis method, 408
 - getParameters method, 407

getResourceBundle, getResourceBundleName methods, 407
getSequenceNumber method, 408
getSourceXxxName methods, 408
getThreadID method, 408
getThrown method, 408
Long class
 converting from long, 252
 hashCode method, 237
long type, 47
 platform-independence of, 48
LongAccumulator class, methods of, 888
LongAdder class, 888, 908
 add, increment, sum methods, 888
Look-and-feel, 539, 700
 appearance of buttons in, 632
 changing, 598–603
 pluggable, 756
LookAndFeelInfo class, methods of, 603
lookup method (*ServiceManager*), 836
Loops
 break statements in, 106–108
 continue statements in, 108
 determinate (for), 99–103
 “for each,” 113–114
 while, 94–99
LotteryArray/LotteryArray.java, 126
LotteryDrawing/LotteryDrawing.java, 118
LotteryOdds/LotteryOdds.java, 102
lower method (*NavigableSet*), 493
Low-level events, 626
Lu, Francis, 810

M

Mac OS X
 Eclipse versions for, 27
 executing JARs in, 783
 JDK versions for, 18
 running applets in, 34–35
 setting paths in, 20
 setting up JDK in, 20
main method, 161–164
 body of, 44
 declared public, 43
 declared static void, 44–45
 eliminating, for applets, 807
 loading classes from, 262
 not defined, 145, 179
 separate for each class, 409
 String[] args parameter of, 116
 tagged with throws, 88
make program (UNIX), 149
MANIFEST.MF (manifest file), 781–782
 editing, 782
 newline characters in, 782
 permissions in, 823
Map interface, 469
 compute, computeIfPresent/Absent methods, 501
 containsKey/Value methods, 499
 entrySet, keySet methods, 502–503
 forEach method, 499
 get, put methods, 469, 497, 499
 merge method, 501
 putAll method, 499
 remove method, 498
 replaceAll method, 502
 values method, 502–503
map/MapTest.java, 498
Map.Entry interface, 502
 getKey, get/setValue methods, 503
mappingCount method (*ConcurrentHashMap*), 906
Maps, 497–509
 adding/retrieving objects to/from, 497
 concurrent, 905–907
 garbage collecting, 504
 hash vs. tree, 497
 implementations for, 497
 keys for, 498
 enumerating, 502
 subranges of, 511
Marker interfaces, 309
Math class, 57–59
 E, PI static constants, 58, 159
 floorMod method, 57
 logarithms, 58
 pow method, 57, 160
 round method, 60
 sqrt method, 57
 trigonometric functions, 58
Matisse, 701, 713–723
max method (*Collections*), 523
Maximum value, computing, 419
menu/MenuFrame.java, 690
MouseListener interface, 689
 menuXxx methods, 689–690
Menus, 678–699
 accelerators for, 687–688
 checkboxes and radio buttons in, 683–684

- Menus (*continued*)
 - icons in, 682–683
 - keyboard mnemonics for, 686–688
 - menu bar in, 679
 - menu items in, 679–684
 - enabling/disabling, 689–693
 - pop-up, 684–686
 - submenus in, 679
- `merge` method
 - of `ConcurrentHashMap`, 909
 - of `Map`, 501
- Merge sort algorithm, 519
- META-INF directory, 781
- Metal look-and-feel, 541, 598
- Method class, 265
 - `getDeclaringClass` method, 270
 - `getExceptionTypes` method, 270
 - `getGenericXxx` methods, 457
 - `getModifiers` method, 265, 270
 - `getName` method, 265, 270
 - `getParameterTypes`, `getReturnType` methods, 270
 - `getTypeParameters` method, 457
 - `invoke` method, 279–283
 - `toString` method, 266
- Method parameters. *See* Parameters
- Method pointers, 279–281
- Method references, 319–321
 - `this`, `super` parameters in, 320
- Method tables, 216
- Methods, 131
 - abstract, 222
 - in functional interfaces, 318
 - accessor, 141–145, 153–154, 444
 - adding, in subclasses, 207
 - applying to objects, 137
 - asynchronous, 915
 - body of, 44–45
 - bridge, 428–429, 440
 - calling by reference vs. by value, 164–171
 - casting, 219–221
 - concrete, 222
 - consistent, 231
 - default, 298–300
 - deprecated, 141, 197, 412
 - destructor, 181–182
 - documentation comments for, 194–198
 - do-nothing, 604
 - dynamic binding for, 209, 214–217
 - exception specification in, 362
 - factory, 161
 - final, 215, 217–218, 265, 295
 - generic, 421–422, 427–429, 466–469
 - helper, 156, 448
 - inline, 7, 218
 - invoking, 45
 - arbitrary, 279–283
 - mutator, 141–145, 154, 444
 - names of, 201
 - overloading, 172
 - overriding, 206–207, 234, 285
 - exceptions and, 364
 - return type and, 427
 - package scope of, 189
 - parameters of, 45–46
 - passing objects to, 136
 - private, 156–157, 215, 265
 - protected, 194, 228, 311
 - public, 194, 265, 290
 - reflexive, 231
 - resolving conflicts in, 300–302
 - return type of, 172, 215
 - signature of, 172, 215
 - static, 160–161, 185, 215, 436
 - adding to interfaces, 298
 - symmetric, 231
 - tracing, 351
 - transitive, 231
 - varargs, 256–257
 - passing generic types to, 432–433
 - visibility of, in subclasses, 217
- `methods/MethodTableTest.java`, 282
- Micro Edition (Java ME), 3, 11, 18
- Microsoft
 - .NET platform, 6
 - ActiveX, 5, 15
 - C#, 8, 11, 218, 280
 - Internet Explorer, 9, 15, 803, 810
 - J#, J++, 8, 280
 - Visual Basic, 3, 136, 587, 638
 - Visual Studio, 23
- MIME types, 825
- `min` method (`Collections`), 523
- Minimum value, computing, 419
- `minimumLayoutSize` method (`LayoutManager`), 728
- `minusDays` method (`LocalDate`), 145
- `mod` method (`BigDecimal`, `BigInteger`), 110–111
- Modality, 730, 742

Model-view-controller design pattern, 632–636
 classes in, 632
 multiple views in, 634
Modifier class
 `isAbstract`, `isInterface`, `isNative`, `isProtected`,
 `isStatic`, `isStrict`, `isSynchronized`, `isVolatile`
 methods, 271
 `isFinal`, `isPrivate`, `isPublic`, `toString` methods,
 265, 271
Modulus, 56
Monitor concept, 884
Mosaic, 10
Mouse events, 616–624
 with keyboard modifiers, 616
 `mouse/MouseComponent.java`, 620
 `mouse/MouseFrame.java`, 619
MouseAdapter class, 619, 626
MouseEvent class, 626
 `getClickCount` method, 616, 623, 627
 `getPoint` method, 623, 627
 `getX/Y` methods, 616, 623, 627
 `isPopupTrigger` method, 686
 `translatePoint` method, 627
MouseHandler class, 619
MouseListener interface, 617, 626
 `mouseClicked` method, 616–617, 619, 627
 `mouseDragged` method, 619
 `mouseEntered/Exited` methods, 619, 627
 `mousePressed` method, 616–617, 627
 `mouseReleased` method, 616, 627
MouseMotionHandler class, 619
MouseMotionListener interface, 617, 619, 626
 `mouseDragged` method, 628
 `mouseMoved` method, 618–619, 628
MouseWheelEvent class, 626
 `getScrollAmount`, `getWheelRotation` methods, 628
MouseWheelListener interface, `mouseWheelMoved`
 method, 628
mouseXxx methods (`Robot`), 778
Mozilla Firefox, 34
Multidimensional arrays, 120–125
 printing, 240
 ragged, 124–127
Multiple inheritance, 297
 not supported in Java, 213
Multiple selections, 103–105
Multiplication operator, 56
multiply method (`BigDecimal`, `BigInteger`), 110–111

Multitasking, 839
Multithreading, 7, 839–952
 deadlocks in, 874, 889–892
 deferred execution in, 325
 performance and, 872, 888, 899, 920
 preemptive vs. cooperative scheduling
 for, 855
 synchronization in, 862–897
 using pools for, 920–926
Mutator methods, 444
 error checking in, 154

N

`\n` escape sequence, 50
`NaN` (not a number), 49
Napkin look-and-feel, 542
naturalOrder method (`Comparator`), 329
Naughton, Patrick, 10–11
NavigableMap interface, 471
 `subMap`, `headMap`, `tailMap` methods, 517
NavigableSet interface, 471, 490, 511
 `ceiling`, `floor` methods, 493
 `descendingIterator` method, 493
 `higher`, `lower` methods, 493
 `pollFirst/Last` methods, 493
 `subSet`, `headSet`, `tailSet` methods, 511, 516
nCopies method (`Collections`), 510, 515
Negation operator, 62
Negative infinity, 49
.NET platform, 6
NetBeans, 20, 24, 409
 Matisse, 701, 713–723
 specifying grid bag constraints in, 706
Netscape, 10
 IFC library, 538
 LiveScript/JavaScript, 15
 Navigator browser, 9, 803, 810
Networking, 4
 `new` operator, 64, 71, 136, 150
 return value of, 138
 with arrays, 111
 with generic classes, 245
 with threads, 855
 `new` keyword, in constructor references, 321
 `newCachedThreadPool` method (`Executors`), 921, 925
 `newCondition` method (`Lock`), 873, 877
 `newFixedThreadPool` method (`Executors`), 921, 925
 `newInstance` method
 of `Array`, 276, 279

- n**
- `newInstance` method (*continued*)
 - of `Class`, 263, 265, 451
 - of `Constructor`, 265, 452
 - `newKeySet` method (`ConcurrentHashMap`), 912
 - `newProxyInstance` method (`Proxy`), 350, 355–356
 - `newScheduledThreadPool` method (`Executors`), 921, 926
 - `newSingleThreadExecutor` method (`Executors`), 921, 925
 - `newSingleThreadScheduledExecutor` method (`Executors`), 921, 926
 - `next` method
 - of `Iterator`, 463, 465, 469
 - of `Scanner`, 81
 - `nextDouble` method (`Scanner`), 79, 81
 - `nextElement` method (`Enumeration`), 465, 528, 530
 - `nextIndex` method
 - of `LinkedList`, 480
 - of `ListIterator`, 483
 - `nextInt` method
 - of `Random`, 181
 - of `Scanner`, 79, 81
 - `nextLine` method (`Scanner`), 79, 81
 - Nimbus look-and-feel, 541
 - No-argument constructors, 173, 208, 801
 - `NoClassDefFoundError`, 26
 - `node` method (`Preferences`), 794, 799
 - `noneOf` method (`EnumSet`), 508
 - `NoSuchElementException`, 464, 469, 483, 494–495
 - Notepad text editor, 26
 - `notHelloWorld/NotHelloWorld.java`, 558
 - `notify`, `notifyAll` methods (`Objects`), 878, 881–882
 - `now` method (`LocalDate`), 140, 145
 - `null` value, 138
 - equality testing to, 231
 - `nullFirst/Last` methods (`Comparator`), 329
 - `NullPointerException`, 361, 383
 - `Number` class, 253
 - `NumberFormat` class
 - factory methods, 161
 - `parse` method, 256
 - `NumberFormatException`, 383
 - Numeric types
 - casting, 60–61
 - comparing, 62, 328
 - converting:
 - to other numeric types, 59–60
 - to strings, 254
 - default initialization of, 172
 - fixed sizes for, 6
 - precision of, 108
 - printing, 82
- o**
- Oak programming language, 10
 - `Object` class, 132, 228–244
 - `clone` method, 156, 306–313, 318
 - `equals` method, 229–235, 244, 302, 512
 - `getClass` method, 244
 - `hashCode` method, 235, 237, 489
 - no redefining for methods of, 302
 - `notify`, `notifyAll` methods, 878, 881–882
 - `toString` method, 238–244, 302, 318
 - `wait` method, 856, 878, 882
 - Object references
 - as method parameters, 165
 - converting, 219
 - default initialization of, 172
 - modifying, 166
 - Object traversal algorithms, 507
 - Object variables, 223
 - in predefined classes, 136–139
 - initializing, 137
 - setting to `null`, 138
 - vs. C++ object pointers, 139
 - vs. objects, 137
 - `objectAnalyzer/ObjectAnalyzer.java`, 273
 - `objectAnalyzer/ObjectAnalyzerTest.java`, 273
 - Object-oriented programming (OOP), 4, 130–135, 203
 - design principles of, 632
 - passing objects in, 302
 - separating time measurement from calendars in, 140
 - vs. procedural, 130–135
 - Objects, 130–133
 - analyzing at runtime, 271–276
 - applying methods to, 137
 - behavior of, 132
 - cloning, 306–313
 - comparing, 295
 - concatenating with strings, 239
 - constructing, 131, 171–182
 - damaged, 896
 - default hash codes of, 235
 - destruction of, 181–182
 - equality testing for, 229–235, 262

- finalize method of, 181–182
identity of, 132
implementing an interface, checks of, 295
in predefined classes, 136–139
initializing, 136
intrinsic locks of, 878
passing to methods, 136
references to, 138
runtime type identification of, 261
serializing, 507
sorting, 290
state of, 131–132, 331–334
vs. object variables, 137
- Objects class
 hash method, 237
 hashCode method, 236–237
- Ocean look-and-feel, 541
- Octal numbers
 formatting output for, 82
 prefix for, 48
- of method
 of EnumSet, 508
 of LocalDate, 140, 145
- offer method
 of BlockingQueue, 898–899, 904
 of Queue, 494
- offerFirst/Last methods
 of BlockingDeque, 905
 of Deque, 494
- offsetByCodePoints method (String), 70, 72
- Online documentation, 71, 74–77, 194, 199
- openFileDialog method (FileOpenService), 830, 837
- openMultiFileDialog method (FileOpenService), 837
- OpenType format, 575
- Operators
 arithmetic, 56–65
 bitwise, 63
 boolean, 62
 hierarchy of, 64–65
 increment/decrement, 61–62
 no overloading for, 109
 relational, 62
- Option dialogs, 731–741
- Optional operations, 514
- optionDialog/ButtonPanel.java, 738
- optionDialog/OptionDialogFrame.java, 734
- or method (BitSet), 533
- Oracle, 12, 18, 20
 Java Plug-in, 803
- JavaFX, 543
- Ordered collections, 470, 476
- ordinal method (Enum), 260
- org.omg.CORBA package, 255
- Originating host, 821
- OSGi platform, 800
- Output statements, 66
- Output, formatting, 82–87
- Overloading resolution, 172, 215
- @Override annotation, 234
- overview.html, 198
- Owner frame, 742
- P**
- p (exponent), in hexadecimal numbers, 49
- pack method (Window), 550, 557, 560
- pack200 compression, 780
- package statement, 183, 185
- package.html, 198
- package-info.java, 198
- Packages, 182–190
 adding classes into, 185–188
 default, 185
 documentation comments for, 194, 198
 importing, 183
 names of, 182, 261
 online documentation for, 71
 scope of, 189–190
 sealing, 787
- PackageTest/com/horstmann/corejava/Employee.java, 188
- PackageTest/PackageTest.java, 187
- paintComponent method (JComponent), 554–556, 560, 577, 583, 897
 overriding, 624
- pair1/PairTest1.java, 420
- pair2/PairTest2.java, 423
- pair3/PairTest3.java, 449
- ParallelGroup class, 714, 723
- Parallelism threshold, 910
- param element (HTML), 810–816
- Parameterized types. *See* Type parameters
- ParameterizedType interface, 453
 getXxx methods, 458
- Parameters, 45–46, 164–171
 checking, with assertions, 386–387
 documentation comments for, 196
 explicit, 152–153
 implicit, 152–153, 160, 409
 modifying, 165–167, 169

- Parameters (*continued*)
 - names of, 175
 - string, 45
 - using collection interfaces in, 527
 - variable number of, 256–257
 - passing generic types to, 432–433
- `ParamTest/ParamTest.java`, 170
- Parent classes. *See* Superclasses
- `parse` method (`NumberFormat`), 256
- `parseInt` method (`Integer`), 254, 256, 811
- Pascal, 10
 - architecture-neutral object file format of, 5
 - passing parameters in, 167
- Password fields, 652–653
- `PasswordChooser` class, 746
- Passwords
 - dialog box for, 746
 - reading from console, 80
- PATH environment variable, 20
- `Path` interface, 298
- `Paths` class, 89, 298
- Payne, Jonathan, 11
- `peek` method
 - of `BlockingQueue`, 898–899
 - of `Queue`, 494
 - of `Stack`, 532
- `peekFirst/Last` methods (`Deque`), 495
- Performance, 7
 - collections and, 471, 486, 906
 - computations and, 56, 59
 - JAR files and, 190
 - measuring with the sieve of Eratosthenes, 533–536
 - multithreading and, 872, 888, 899, 920
 - of Java vs. C++, 14, 534
 - of tests vs. catching exceptions, 381
- Permits, 935
- `PersistenceService` class, 831
 - `create` method, 831, 837
 - `delete` method, 838
 - `get`, `getNames` methods, 838
- Persistent storage, 272
- `Phaser` class, 937
- Physical limitations, 359
- `PI` constant (`Math`), 58, 159
- `plaf/PlafFrame.java`, 601
- `play` method (`Applet`), 817
- Plug-ins, 800–802
- `plusDays` method (`LocalDate`), 141, 145
- `Point` class, 564
- Point size (in typesetting), 574–575
- `Point2D` class, 563–564
- `Point2D.Double` class, 563, 569
- `Point2D.Float` class, 563
- `poll` method
 - of `BlockingQueue`, 898–899, 904
 - of `ExecutorCompletionService`, 928
 - of `Queue`, 494
- `pollFirst/Last` methods
 - of `Deque`, 495, 905
 - of `NavigableSet`, 493
- Polymorphism, 209, 213–214, 285
- `pop` method (`Stack`), 532
- Pop-up menus, 684–686
 - triggers for, 685
- Pop-up windows, 821
- Positive infinity, 49
- PostScript Type 1 format, 575
- `pow` method (`Math`), 57, 160
- Precision, of numbers, 82
- Preconditions, 387
- Predefined action table names, 609
- Predefined classes, 135–145
 - mutator and accessor methods in, 141–145
 - objects and object variables in, 136–139
- `Predicate` interface, 319, 326
- Preemptive scheduling, 855
- Preferences, 788–800
 - accessing, 794
 - enumerating keys in, 795
 - importing/exporting, 795
- `Preferences` class, 794–800
 - `exportXxx` methods, 795, 800
 - `get`, `getDataType` methods, 795, 800
 - `importPreferences` method, 795, 800
 - `keys` method, 795, 799
 - `node` method, 794, 799
 - platform-independency of, 794
 - `put`, `putDataType` methods, 795, 800
 - `system/userNodeForPackage` methods, 794, 799
 - `system/userRoot` methods, 794, 799
- `preferences/PreferencesTest.java`, 796
- `preferredLayoutSize` method (`LayoutManager`), 728
- `previous` method (`ListIterator`), 476, 483
- `previousIndex` method
 - of `LinkedList`, 480

- of *ListIterator*, 483
Prime numbers, 533
Primitive types, 47–53
 as method parameters, 165
 comparing, 328
 converting to objects, 252
 final fields of, 157
 not for type parameters, 430–431
 transforming hash map values to, 911
 values of, not object, 229
Princeton University, 5
`print` method (`System.out`), 46, 82
`printf` method (`System.out`), 82–86
 conversion characters for, 82
 flags for, 83–84
 for date and time, 84–85
 parameters of, 256
`println` method (`System.out`), 45–46, 79, 319, 389
`printStackTrace` method (`Throwable`), 264–265, 377, 410
`PrintStream` class, 830
`PrintWriter` class, 87, 89
Priority queues, 495
`PriorityBlockingQueue` class, 899, 904
`PriorityQueue` class, 496
 as a concrete collection type, 472
`priorityQueue/PriorityQueueTest.java`, 496
private access modifier, 150, 189–190, 333
 checking, 265
 for fields, in superclasses, 206
 for methods, 156–157
Procedures, 130
`process` method (`SwingWorker`), 944–946, 950
Processes, vs. threads, 840
Producer threads, 898
Profilers, 413
Programs. *See* Applications
Properties, 549, 788–793
`Properties` class, 528–531, 788–793
 `getProperty` method, 531, 789, 792
 `load`, `store` methods, 531, 788, 793
 `setProperty` method, 792
`properties/PropertiesTest.java`, 790
Property maps, 530–531, 788–793
 comments in, 599
 names of, 788
 reading/writing, 788
`PropertyChangeListener` interface, 758
protected access modifier, 227–228, 311
 for fields, 283
Proxies, 350–356
 properties of, 355–356
 purposes of, 351
Proxy class, 355–356
 `get/isProxyClass` methods, 355–356
 `newProxyInstance` method, 350, 355–356
`proxy/ProxyTest.java`, 353
public access modifier, 42, 56, 147–150, 189–190, 290
 checking, 265
 for fields in interfaces, 296
 for `main` method, 43
 for only one class in source file, 147
 not specified for interfaces, 289
publish method
 of `Handler`, 398, 406
 of `SwingWorker`, 944–945, 950
Pure virtual functions (C++), 224
push method (`Stack`), 532
put method, 908
 of `BlockingQueue`, 898–899, 904
 of `Map`, 469, 497, 499
 of `Preferences`, 795, 800
putAll method (`Map`), 499
putData Type methods (`Preferences`), 795, 800
putFirst/Last methods (`BlockingDeque`), 904
putIfAbsent method (`ConcurrentHashMap`), 908
putValue method (`Action`), 608, 615
- Q**
- `Queue` interface, 460, 462, 494–495
 methods of, 494
Queues, 460–463, 494–495
 blocking, 898–905
 concurrent, 905–907
 double-ended. *See* Deques
QuickSort algorithm, 117, 519
- R**
- `\r` escape sequence, 50
Race conditions, 862–868
 and atomic operations, 887
Radio buttons, 660–663
 in menus, 683–684
`radioButton/RadioButtonFrame.java`, 662
Ragged arrays, 124–127

- Random class, 181
 - nextInt method, 181
 - thread-safe, 892
- Random number generation, 181, 892
- RandomAccess interface, 471, 520, 522
- range method (*EnumSet*), 508
- Raw types, 425–426
 - converting type parameters to, 441
 - type inquiring at runtime, 431
- Read/write locks, 895–896
- readConfiguration method (*LogManager*), 392
- readLine/Password methods (*Console*), 81
- Rectangle class, 490, 564
- Rectangle2D class, 560, 562–565
 - getWidth, setRect methods, 563
- Rectangle2D.Double class, 562–563, 568
- Rectangle2D.Float class, 562–563, 569
- Rectangles, 560
 - comparing, 490
 - constructing, 564
 - drawing, 561
 - filling with color, 569
- RectangularShape class, 563
 - getHeight/Width, getCenterX/Y methods, 563, 568
 - getX/Y, getMinX/Y, getMaxX/Y methods, 568
- Recursive computations, 929
- RecursiveAction class, 929
- RecursiveTask class, 929
- Red-black trees, 489
- reduce, reduceXxx methods (*ConcurrentHashMap*), 910–911
- Redundant keywords, 296
- Reentrant locks, 870
- ReentrantLock class, 868–872
- ReentrantReadWriteLock class, 895–896
- Reflection, 204, 260–283
 - analyzing:
 - classes, 265–271
 - objects, at runtime, 271–276
 - generics and, 276–279, 450–458
 - overusing, 286
 - reflection/ReflectionTest.java, 267
- Relational operators, 62, 64
- Relative resource names, 784
- remove method
 - of *ArrayList*, 249, 251
 - of *BlockingQueue*, 898–899
 - of *Collection*, 467–468
 - of *Iterator*, 463, 465, 469
 - of *JMenu*, 681
 - of *List*, 470, 483
 - of *ListIterator*, 478
 - of *Map*, 498
 - of *Queue*, 494
 - of *ThreadLocal*, 893
- removeAll method
 - of *Collection*, 467–468
 - of *LinkedList*, 480
- removeEldestEntry method (*LinkedHashMap*), 506, 508
- removeFirst/Last methods
 - of *Deque*, 495
 - of *LinkedList*, 484
- removeHandler method (*Logger*), 406
- removeIf method
 - of *ArrayList*, 319
 - of *Collection*, 468, 524
- removeLayoutComponent method (*LayoutManager*), 728
- removePropertyChangeListener method (*Action*), 608–609
- removeXxx methods (*JComboBox*), 669, 671
- repaint method
 - of *Component*, 556
 - of *JComponent*, 559, 841, 951
- replace method
 - of *ConcurrentHashMap*, 908
 - of *String*, 73
- replaceAll method
 - of *Collections*, 523
 - of *List*, 524
 - of *Map*, 502
- replaceRange method (*JTextArea*), 951
- Reserved words, 43
 - forbidden for variable names, 53
 - not used, 56
- resetChoosableFilters method (*JFileChooser*), 756, 763
- resize method (*Applet*), 808
- Resource bundles, 393–394
- resource/ResourceTest.java, 786
- ResourceBundle class, 394
- Resources, 783–787
 - closing, 373
 - exhaustion of, 360
 - localization of, 784
 - names of, 784–785
- Restricted views, 514
- resume method (*Thread*), 858
- retain method (*Collection*), 467

retainAll method (*Collection*), 469
Retirement/Retirement.java, 97
Retirement2/Retirement2.java, 98
return statement
 in finally blocks, 374
 in lambda expressions, 316
Return types, 215
 covariant, 429
 documentation comments for, 196
 for overridden methods, 427
Return values, 138
@return comment (javadoc), 196
revalidate method (*JComponent*), 649–650, 951
reverse method (*Collections*), 524
reversed, reverseOrder methods (*Comparator*), 329, 519, 521
RoadApplet/RoadApplet.html, 36
RoadApplet/RoadApplet.java, 38
Robot class, 774–778
 methods of, 778
robot/RobotTest.java, 775
rotate method (*Collections*), 524
round method (*Math*), 60
Rounding mode, 111
RoundingMode class, 111
rt.jar file, 780
run method (*Thread*), 849, 851
runAfterXxx methods (*CompletableFuture*), 934
runFinalizersOnExit method (*System*), 182
Runnable interface, 326, 847
 lambdas and, 318
 run method, 325, 851
Runtime
 adding shutdown hooks at, 182
 analyzing objects at, 271–276
 creating classes at, 350
 setting the size of an array at, 244
 type identification at, 220, 261, 431
RuntimeException, 360, 380, 383

S

@SafeVarargs annotation, 432
Sandbox, 820–822
saveAsFileDialog method (*FileSaveService*), 837
saveFileDialog method (*FileSaveService*), 830, 837
Scala programming language, default
 methods in, 300
Scanner class, 79–81, 87, 89
 next, hasNext, hasNextType methods, 81
 nextXxx methods, 79, 81
Scheduled execution, 926
ScheduledExecutorService class, methods of, 926
Scroll panes, 654–656
Scrollbars, 654–656
Sealing, 787
search, searchXxx methods (*ConcurrentHashMap*), 910–911
Secure certificates, 822
Security, 4–5, 14, 820–822
@see comment (javadoc), 197–198
Semantic events, 626
Semaphore class, 935
Semaphores, 935
SequentialGroup class, 714, 723
Serialization, 507
 of applet objects, 809
Service loaders, 800–802
ServiceLoader class, 801
 iterator, load methods, 802
ServiceManager interface, 830
 getServiceNames, lookup methods, 836
ServletException, 370
Servlets, 370
Set interface, methods of, 471
set method
 of *Array*, 279
 of *ArrayList*, 247, 251
 of *BitSet*, 533
 of *Field*, 276
 of *List*, 483
 of *ListIterator*, 478, 483
 of *ThreadLocal*, 893
 of *Vector*, 883
set/SetTest.java, 487
setAccelerator method (*JMenuItem*), 687–688
setAcceptAllFileFilterUsed method (*JFileChooser*), 756, 763
setAccessible method (*AccessibleObject*), 272, 275
setAccessory method (*JFileChooser*), 763
setAction method (*AbstractButton*), 681
setActionCommand method (*AbstractButton*), 663
setAutoCreateXxx methods (*GroupLayout*), 722
setBackground method (*Component*), 570, 573
setBoolean, setByte, setChar methods (*Array*), 279
setBorder method (*JComponent*), 664, 668
setBounds method (*Component*), 546, 552, 724
 coordinates in, 548
setCharAt method (*StringBuilder*), 78

setClassAssertionStatus method (`ClassLoader`), 388
setColor method
 of `Graphics`, 570, 572
 of `JColorChooser`, 770
setColumns method
 of `JTextArea`, 654, 656
 of `JTextField`, 649–650
setComponentPopupMenu method (`JComponent`), 685–686
setCurrentDirectory method (`JFileChooser`), 754,
 762
setCursor method (`Component`), 624
setDaemon method (`Thread`), 859–860
setDebugGraphicsOptions method (`JComponent`), 771
setDefaultAssertionStatus method (`ClassLoader`), 388
setDefaultButton method (`JRootPane`), 748, 752
setDefaultCloseOperation method (`JDialog`), 743,
 807
setDefaultUncaughtExceptionHandler method (`Thread`),
 411, 860–861
setDisplayedMnemonicIndex method (`AbstractButton`),
 686, 688
setDouble method (`Array`), 279
setEchoChar method (`JPasswordField`), 653
setEditable method
 of `JComboBox`, 669, 671
 of `JTextComponent`, 648
setEnabled method
 of `Action`, 608, 615
 of `JMenuItem`, 689–690
setExtendedState method (`Frame`), 553
setFileFilter method (`JFileChooser`), 755, 763
setFileSelectionMode method (`JFileChooser`), 754,
 763
setFileView method (`JFileChooser`), 756–757, 763
setFilter method
 of `Handler`, 406
 of `Logger`, 398, 406
setFloat method (`Array`), 279
setFont method
 of `Graphics`, 581
 of `JComponent`, 650
setForeground method (`Component`), 570, 573
setFormatter method (`Handler`), 399, 406
setFrameFromCenter method (`Ellipse2D`), 565
setFrameFromDiagonal method (`Ellipse2D`), 564
setHonorsVisibility, setHorizontalGroup methods
 (`GridLayout`), 722
setHorizontalTextPosition method (`AbstractButton`),
 682–683
setIcon method
 of `JLabel`, 651–652
 of `JMenuItem`, 682
setIconImage method (`Frame`), 546, 553
setInheritsPopupMenu method (`JComponent`), 685–686
setInt method (`Array`), 279
setInverted method (`JSlider`), 674, 678
setMenuBar method (`JFrame`), 679, 682
setLabelTable method (`JSlider`), 429, 673, 678
setLayout method (`Container`), 641
setLevel method
 of `Handler`, 406
 of `Logger`, 389, 405
setLineWrap method (`JTextArea`), 654, 656
 setLocation method (`Component`), 546, 552
 coordinates in, 548
setLocationByPlatform method (`Window`), 552
setLong method (`Array`), 279
setLookAndFeel method (`UIManager`), 599, 602
setMnemonic method (`AbstractButton`), 687–688
setModel method (`JComboBox`), 669
setMultiSelectionEnabled method (`JFileChooser`),
 754, 763
setOut method (`System`), 159
setPackageAssertionStatus method (`ClassLoader`), 388
setPaint method (`Graphics2D`), 569, 573
setPaintLabels method (`JSlider`), 673, 678
setPaintTicks method (`JSlider`), 673–674, 678
setPaintTrack method (`JSlider`), 678
setParent method (`Logger`), 406
setPriority method (`Thread`), 859
setProperty method
 of `Properties`, 792
 of `System`, 392
setRect method (`Rectangle2D`), 563
setResizable method (`Frame`), 546, 553
setRows method (`JTextArea`), 654, 656
Sets, 487
 concurrent, 905–907
 intersecting, 525
 mutating elements of, 487
 subranges of, 511
 thread-safe, 912
setSelected method
 of `AbstractButton`, 684
 of `JCheckBox`, 657, 659
setSelectedFile/Files methods (`JFileChooser`), 754,
 763
setSelectionStart/End methods (`JComponent`), 952

setShort method (*Array*), 279
setSize method (*Component*), 552
setSnapToTicks method (*JSlider*), 673, 678
setTabSize method (*JTextArea*), 656
setText method
 of *JLabel*, 651–652
 of *JTextComponent*, 648, 650, 951
setTime method (*Calendar*), 218
setTitle method (*JFrame*), 546, 553
setToolTipText method (*JComponent*), 699
setUncaughtExceptionHandler method (*Thread*), 861
setUndecorated method (*Frame*), 546, 553
setUseParentHandlers method (*Logger*), 406
setValue method (*Map.Entry*), 503
setVerticalGroup method (*GroupLayout*), 722
setVisible method
 of *Component*, 546, 552, 951
 of *JDialog*, 743, 746, 807
setWrapStyleWord method (*JTextArea*), 656
setXxxTickSpacing methods (*JSlider*), 678
severe method (*Logger*), 390, 404
Shallow copies, 308–310
Shape interface, 560–561
Shell
 redirection syntax of, 88
 scripts in, 193
Shift operators, 63
short type, 47
Short class
 converting from *short*, 252
 hashCode method, 237
show method (*JPopupMenu*), 685
showConfirmDialog method (*JOptionPane*), 731–732, 739
showDialog method
 of *JColorChooser*, 770
 of *JFileChooser*, 747, 752, 754, 763
showDocument method
 of *AppletContext*, 819–820
 of *BasicService*, 836
showInputDialog method (*JOptionPane*), 731–732, 740
showInternalConfirmDialog, showInternalMessageDialog methods (*JOptionPane*), 739
showInternalInputDialog method (*JOptionPane*), 741
showInternalOptionDialog method (*JOptionPane*), 740
showMessageDialog method (*JOptionPane*), 304, 731–732, 739
showOptionDialog method (*JOptionPane*), 731–732, 739–740
showStatus method (*Applet*), 819–820
showXxxDialog methods (*JFileChooser*), 747, 752, 754, 763
shuffle method (*Collections*), 520–521
shuffle/ShuffleTest.java, 520
Shuffling, 520
Shutdown hooks, 182
shutdown method (*ExecutorService*), 922, 925
shutdownNow method (*ExecutorService*), 922, 927
Sieve of Eratosthenes benchmark, 533–536
sieve/sieve.cpp, 535
sieve/Sieve.java, 534
signal method (*Condition*), 875–877, 890
signalAll method (*Condition*), 874–877, 890
Signatures (of methods), 172, 215
simpleframe/SimpleFrameTest.java, 544
sin method (*Math*), 58
Single-thread rule (Swing), 939, 951–952
singleton, singletonCollection methods
 (*Collections*), 510, 515
size method
 of *ArrayList*, 246–247
 of *Collection*, 467–468
 of concurrent collections, 905
sizedFrame/SizedFrameTest.java, 551
sleep method (*Thread*), 841, 846–847, 852
slider/SliderFrame.java, 674
Sliders, 672–678
 ticks on, 673–674
 vertical, 672
SoftBevelBorder class, 665, 667
Software Development Kit (SDK), 18
Solaris
 Eclipse versions for, 27
 executing JARs in, 783
 JDK versions for, 18
sort method
 of *Arrays*, 117–119, 290, 292, 294, 314, 318
 of *Collections*, 518–521
 of *List*, 521
SortedMap interface, 471
 comparator, first/lastKey methods, 500
 subMap, headMap, tailMap methods, 511, 516
SortedSet interface, 471, 511
 comparator, first, last methods, 493
 subSet, headSet, tailSet methods, 511, 516

Sorting
algorithms for, 117, 518–521
arrays, 117–120, 292
assertions for, 387
in reverse order, 519
people, by name, 328–329
strings by length, 305–306, 314, 316
Source files, 192
editing in Eclipse, 29
installing, 22–23
Special characters, 50
Splash screen, 262
Spring layout, 700
`sqr` method (`Math`), 57
`src.zip` file, 22
`Stack` interface, 460, 528, 531
 peek, pop, push methods, 532
`Stack` trace, 377–381, 889
Stacks, 531
`StackTrace`/`StackTraceTest.java`, 378
`StackTraceElement` class
 `getLineNumber` method, 380
 `getXxxName` methods, 380
 `isNativeMethod` method, 381
 `toString` method, 378, 381
Standard Edition (Java SE), 11, 18
Standard Java library
 companion classes in, 298
 online API documentation for, 71, 74–77, 194, 199
 packages in, 182
Standard Template Library (STL), 460, 465
`start` method
 of `Applet`, 808
 of `Thread`, 849, 851, 855
 of `Timer`, 305
`startsWith` method (`String`), 72
`stateChanged` method (`ChangeListener`), 672–673
Statements, 45
 compound. *See* Blocks
`static` access modifier, 158–164
 for fields in interfaces, 296
 for `main` method, 44–45
Static binding, 215
Static constants, 159
 documentation comments for, 196
Static fields, 158–159
 accessing, in static methods, 160
 importing, 185
 initializing, 178
 no type variables in, 436
static final access modifier, 55
Static imports, 185
Static inner classes, 331, 346–349
Static methods, 160–161
 accessing static fields in, 160
 adding to interfaces, 298
 importing, 185
 no type variables in, 436
Static variables, 159
`staticInnerClass/StaticInnerClassTest.java`, 348
`StaticTest/StaticTest.java`, 163
`stop` method
 of `Applet`, 808
 of `Thread` (deprecated), 851, 858, 896–897
 of `Timer`, 305
`store` method (`Properties`), 531, 788, 793
Strategy design pattern, 631
`Stream` interface, `toArray` method, 321
`StreamHandler` class, 397
`strictfp` keyword, 57
`StrictMath` class, 57, 59
`String` class, 65–78
 `charAt` method, 70, 72
 `codePointAt`, `codePoints` methods, 72
 `codePointCount` method, 70, 73
 `compareTo` method, 72
 `endsWith` method, 72
 `equals`, `equalsIgnoreCase` methods, 68, 72
 `format`, `formatTo` methods, 83
 `hashCode` method, 235, 485
 immutability of, 67, 157, 218
 `indexOf` method, 73, 172
 `join` method, 73
 `lastIndexOf` method, 73
 `length` method, 69–70, 73
 `offsetByCodePoints` method, 70, 72
 `replace` method, 73
 `startsWith` method, 72
 `substring` method, 66, 73, 510
 `toLowerCase`, `toUpperCase` methods, 73
 `trim` method, 73, 650
`StringBuilder` class, 77–78
 `append` method, 77–78
 `appendCodePoint` method, 78
 `delete` method, 78
 `insert` method, 78

- length method, 78
setCharAt method, 78
toString method, 77–78
- Strings, 65–78
building, 77–78
code points/code units of, 70
comparing, 305–306
concatenating, 66–67
with objects, 239
converting to numbers, 254
empty, 69
equality of, 68
formatting output for, 82–87
immutability of, 67
length of, 66, 69
null, 69
shared, in compiler, 67, 69
sorting by length, 305–306, 314, 316
substrings of, 66
using ". . ." for, 45
- Strongly typed languages, 47, 291
- Subclasses, 204–228
adding fields/methods to, 207
anonymous, 344
cloning, 311
comparing objects from, 295
constructors for, 207
defining, 204
method visibility in, 217
no access to private fields of superclass, 227
overriding superclass methods in, 207
- subList method (*List*), 510, 516
- subMap method
of *NavigableMap*, 517
of *SortedMap*, 511, 516
- Submenus, 679
- submit method
of *ExecutorCompletionService*, 925, 928
of *ExecutorService*, 921
- Subranges, 510–511
- subSet method (*NavigableSet*, *SortedSet*), 511, 516
- Substitution principle, 213
- substring method (*String*), 66, 73, 510
- subtract method (*BigDecimal*, *BigInteger*), 110–111
- Subtraction operator, 56
- sum method (*LongAdder*), 888
- Sun Microsystems, 2, 5–12, 14, 539
HotJava browser, 11, 802
Java Plug-in, 803
- super keyword, 207, 444
capturing in method references, 320
vs. this, 207–208
- Superclass wins rule, 300
- Superclasses, 204–228
accessing private fields of, 206
common fields and methods in, 223, 283
overriding methods of, 234
throws specifiers in, 364, 369
- Supertype bounds, 444–447
- Supplementary characters, 52
- Supplier* interface, 326
- @SuppressWarnings annotation, 105, 252, 430, 432, 437–439
- Surrogates area (Unicode), 52
- suspend method (*Thread*, deprecated), 858, 896–897
- swap method (*Collections*), 524
- Swing, 537–586, 629–778
advantages of, 539
debugging, 770–778
double buffering in, 771
implementing applets with, 803–808
in full-screen, 550
model-view-controller analysis of, 636–638
starting, 545
threads and, 937–943
single-thread rule, 939, 951–952
- Swing graphics debugger, 771
- swing/SwingThreadTest.java, 940
- swing.properties file, 598
- SwingConstants* interface, 296, 651
- SwingUtilities class
getAncestorOfClass method, 747, 752
updateComponentTreeUI method, 599
- SwingWorker class, 943–950
doInBackground method, 944–945, 950
execute method, 945, 950
getState method, 950
process method, 944–946, 950
publish method, 944–945, 950
- swingWorker/SwingWorkerTest.java, 947
- switch statement, 103–105
enumerated constants in, 105
missing break statements in, 412

- SWT toolkit, 543
`synch/Bank.java`, 875
`synch2/Bank.java`, 880
Synchronization, 862–897
 condition objects, 872–877
 final variables, 886
 in `Vector`, 484
 lock objects, 868–872
 lock testing and timeouts, 893–895
 monitor concept, 884
 race conditions, 862–868, 887
 read/write locks, 895
 volatile fields, 885–886
Synchronization primitives, 935
Synchronization wrappers, 914–915
Synchronized blocks, 882–883
 `synchronized` keyword, 868, 878–882, 884
Synchronized views, 512–513
 `synchronizedCollection` methods (`Collections`), 512–513, 515, 915
Synchronizers, 934–937
 barriers, 936–937
 countdown latches, 936
 exchangers, 937
 semaphores, 935
 synchronous queues, 937
SynchronousQueue class, 935–937
Synth look-and-feel, 542
System class
 `console` method, 81
 `exit` method, 45
 `getProperties` method, 789, 793
 `getProperty` method, 793
 `identityHashCode` method, 507, 509
 `runFinalizersOnExit` method, 182
 `setOut` method, 159
 `setProperty` method, 392
System of Patterns, A (Buschmann et al.), 632
System.err class, 411
System.in class, 79
System.out class, 45–46, 159, 411
 `print` method, 82
 `printf` method, 82–86, 256
 `println` method, 79, 389
SystemColor class, 571–572
systemNodeForPackage method (`Preferences`), 794, 799
systemRoot method (`Preferences`), 794, 799
- T**
T type variable, 419
\`t escape sequence, 50
Tab key
 escape sequence for, 50
 navigating GUI controls with, 729
Tagging interfaces, 309, 426, 471
tailMap method
 of `NavigableMap`, 517
 of `SortedMap`, 511, 516
tailSet method (`NavigableSet`, `SortedSet`), 511, 516
take method
 of `BlockingQueue`, 898–899, 904
 of `ExecutorCompletionService`, 928
takeFirst/Last methods (`BlockingDeque`), 904
tan method (`Math`), 58
tar command, 780
target attribute (HTML), 820
Tasks
 controlling groups of, 927–928
 decoupling from mechanism of running, 848
 interrupting, 842
 multiple, 839
 running asynchronously, 915
 scheduled, 926
 time-consuming, 939–943
 work stealing for, 930
Template code bloat, 426
Terminal window, 25
Text
 centering, 576
 displaying, 557
 fonts for, 573–582
 typesetting properties of, 576
Text areas, 653–654
 formatted text in, 654
 preferred size of, 654
 scrollbars in, 654–656
Text fields, 649–651
 columns in, 649
 creating blank, 650
 preferred size of, 649
Text input, 648–656
 labels for, 651–652
 password fields, 652–653
 scroll panes, 654
`text/TextComponentFrame.java`, 655

thenAccept, thenApply, thenApplyAsync, thenRun
 methods (`CompletableFuture`), 933
thenAcceptBoth, thenCombine methods
 (`CompletableFuture`), 934
thenComparing method (`Comparator`), 328–329
thenCompose method (`CompletableFuture`),
 932–933
this keyword, 152, 176
 capturing in method references, 320
 in first statement of constructor, 176
 in inner classes, 335
 in lambda expressions, 324
 vs. super, 207–208
Thread class
 `currentThread` method, 851–854
 extending, 848
 `get/setUncaughtExceptionHandler` methods, 861
 `getDefaultUncaughtExceptionHandler` method,
 861
 `getState` method, 858
 `interrupt`, `isInterrupted` methods, 851–854
 `interrupted` method, 853–854
 `join` method, 856–858
 methods with timeout, 856
 `resumes` method, 858
 `run` method, 849, 851
 `setDaemon` method, 859–860
 `setDefaultUncaughtExceptionHandler` method, 411,
 860–861
 `setPriority` method, 859
 `sleep` method, 841, 846–847, 852
 `start` method, 849, 851, 855
 `stop` method (deprecated), 851, 858,
 896–897
 `suspend` method (deprecated), 858,
 896–897
 `yield` method, 859
Thread dump, 889
Thread groups, 860
Thread pools, 920–926
 of fixed size, 921
`Thread.UncaughtExceptionHandler` interface,
 860–862
ThreadDeath error, 857, 862, 896
ThreadGroup class, 861
 `uncaughtException` method, 861–862
ThreadLocal class, methods of, 893
ThreadLocalRandom class, `current` method,
 893
threadPool/`ThreadPoolTest.java`, 922
ThreadPoolExecutor class, 921–922
 `getLargestPoolSize` method, 926
Threads
 accessing collections from, 512–513,
 905–915
 blocked, 852, 856–857
 condition objects for, 872–877
 daemon, 859
 defined, 840–851
 executing code in, 325
 handlers for uncaught exceptions in,
 860–862
 idle, 928
 interrupting, 851–854
 listing all, 889
 locking, 882–883
 new, 855
 preemptive vs. cooperative scheduling
 for, 855
 priorities of, 858
 producer/customer, 898
 purposes of, 846–851
 `runnable`, 855–856
 simple procedure for, 846–851
 states of, 855–858
 Swing and, 937–943, 951–952
 synchronizing, 862–897, 934–937
 terminated, 847, 851, 857
 thread-local variables in, 892–893
 timed waiting, 856–857
 unblocking, 875
 vs. processes, 840
 waiting, 856–857, 873
 work stealing for, 930
Thread-safe collections, 905–915
 callables and futures, 915–920
 concurrent, 905–907
 copy on write arrays, 912
 synchronization wrappers, 914–915
throw keyword, 364–365
Throwable class, 360, 383
 `add/getSuppressed` methods, 377, 380
 `get/initCause` methods, 379
 `getMessage` method, 366
 `getStackTrace` method, 377, 379
 `printStackTrace` method, 264–265, 377,
 410
 `toString` method, 366

throwing method (*Logger*), 392, 405
throws keyword, 361–364
 for *main* method, 88
@throws comment (javadoc), 196
Ticks, 673
 icons for, 674
 labeling, 673
 snapping to, 673
Time measurement vs. calendars, 140
Timed waiting threads, 856–857
Timeless Way of Building, *The* (Alexander), 630
TimeoutException, 915
Timer class, 302, 314, 627
 start, stop methods, 305
timer/TimerTest.java, 304
title element (HTML), 807
toArray method
 of *ArrayList*, 435
 of *Collection*, 249, 467, 469
 of *Stream*, 321
toBack/Front methods (*Window*), 552
toLowerCase method (*String*), 73
Tomcat, 824–838
toolBar/ToolBarFrame.java, 697
Toolbars, 694–696
 detaching, 695
 dragging, 694
 title of, 696
 vertical, 696
Toolkit class
 beep method, 305
 createCustomCursor method, 618, 623
 getDefaultToolkit method, 305, 549, 553
 getScreenSize method, 549, 553
Toolkit-modal dialogs, 742
Tooltips, 696–699
toString method
 adding to all classes, 240
 Formattable and, 83
 of *Arrays*, 114, 119
 of *Date*, 137
 of *Enum*, 258, 260
 of *Integer*, 256
 of *Modifier*, 266, 271
 of *Object*, 238–244, 302
 of proxy classes, 355
 of *StackTraceElement*, 378, 381
 of *StringBuilder*, 77–78
 of *Throwable*, 366
 redeclaring, 318
 working with any class, 272
Total ordering, 490
toUpperCase method (*String*), 73
TraceHandler class, 351
Tracing execution flow, 391
TransferQueue interface, 900
 transfer, tryTransfer methods, 905
translatePoint method (*MouseEvent*), 627
Traversal order, 729–730
Tree maps, 497
Tree sets, 489–493
 adding elements to, 490
 red-black, 489
 total ordering of, 490
 vs. priority queues, 495
TreeMap class, 471, 497, 500
 as a concrete collection type, 472
 vs. *HashMap*, 497
TreeSet class, 471, 489–493
 as a concrete collection type, 472
treeSet/Item.java, 491
treeSet/TreeSetTest.java, 490
Trigonometric functions, 58
trim method (*String*), 73, 650
trimToSize method (*ArrayList*), 246–247
Troubleshooting. *See* Debugging
TrueType format, 575
Truncated computations, 56
try/catch statement, 264, 367–372
 decoupling, 374
 generics and, 436–437
 wrapping entire task in try block, 382
try/finally statement, 372–376
 decoupling, 374
tryLock method (*Lock*), 856, 893–895
Try-with-resources statement, 376–377
 no locks with, 869
Two-dimensional arrays, 120–125
Type interface, 453
Type erasure, 425–430
 clashes after, 439–440
Type parameters, 245
 converting to raw types, 441
 not for arrays, 431–432, 441
 not instantiated with primitive types, 430–431
 vs. inheritance, 416

Type variables

- bounds for, 422–424
- in exceptions, 437
- in static fields or methods, 436
- matching in generic methods, 452
- names of, 419
- no instantiating for, 433–434
- replacing with bound types, 425–426

Typesetting terms, 576**TypeVariable** interface, 453

- getBounds, getName methods, 457

U**UCSD Pascal system**, 5**UIManager** class

- getInstalledLookAndFeels, setLookAndFeel methods, 602
- setLookAndFeel method, 599

UML (Unified Modeling Language)

- notation, 134–135

UnaryOperator interface, 326**UnavailableServiceException**, 830**uncaughtException** method (*ThreadGroup*), 861–862**UncaughtExceptionHandler** interface, 860–862

- uncaughtException method, 861

Unchecked exceptions, 264, 361–363

- applicability of, 383

Unequality operator, 62**Unicode standard**, 6, 51–52, 65

- in char type, 50

Unit testing, 162**University of Illinois**, 10**UNIX**

- Eclipse versions for, 27
- JNLP configuration in, 828
- running applets in, 34
- setting paths in, 20, 191–193
- setting up JDK in, 20
- system directories, 788
- troubleshooting Java programs in, 26

unlock method (*Lock*), 869, 871**Unmodifiable views**, 511–512**unmodifiableCollection** methods (*Collections*), 511–512, 514**UnsupportedOperationException**, 503, 510, 512, 514**unsynch/Bank.java**, 865**unsynch/UnsynchBankTest.java**, 864**updateAndGet** method (*AtomicType*), 887**updateComponentTreeUI** method (*SwingUtilities*), 599**User input**, 650

- errors of, 359

User Interface. *See* **Graphical User Interface****userNodeForPackage** method (*Preferences*), 794, 799**userRoot** method (*Preferences*), 794, 799**“Uses-a” relationship**, 133–135**UTC** (Coordinated Universal Time), 139**UTF-8 standard**, 87**Utility classes**, 298–299**V****V** type variable, 419**validate** method (*Component*), 651, 951**valueOf** method

- of *BigDecimal*, *BigInteger*, 108, 110–111
- of *Enum*, 258, 260
- of *Integer*, 256

values method (*Map*), 502–503**Values**, captured by lambda expressions, 323**Varargs**, 256–257

- passing generic types to, 432–433

Variables, 53–56

- accessing in lambdas, 322–324

- copying, 306

- declarations of, 53

- deprecated, 197

- effectively final, 324

- final, accessing from outer methods, 339–342

- initializing, 54, 200

- local, 138, 430

- annotating, 430

- mutating in lambda expressions, 323

- names of, 53–56

- package scope of, 189

- printing/logging values of, 409

- static, 159

- thread-local, 892–893

Vector class, 460, 528, 883, 914–915

- elements method, 530

- for dynamic arrays, 245

- get, set methods, 883

- synchronization in, 484

@version comment (*javadoc*), 197, 199

- Views, 509, 633
 bulk operations for, 525
 checked, 513
 restricted, 514
 subranges of, 510–511
 synchronized, 512–513
 unmodifiable, 511–512
- Visual Basic programming language
 built-in date type in, 136
 event handling in, 587
 forms in, 638
 syntax of, 3
- Visual Studio, 23
- void keyword, 44–45
- Volatile fields, 885–886
- volatile keyword, 885–886
- von der Ahé, Peter, 422
- W**
- wait method (`Object`), 856, 878, 882
- Wait sets, 873
- warning method (`Logger`), 390, 404
- Warnings
 fallthrough behavior, 105
 generic types, 252, 430, 432,
 437–439
 suppressing, 432, 437–439
- Weak hash maps, 504
- Weak references, 504
- `WeakHashMap` class, 504, 507
 as a concrete collection type, 472
- Weakly consistent iterators, 906
- `WeakReference` object, 504
- Web pages
 dynamic, 9
 reading from URL, 932
 showing applets on, 802–824
 title of, 807
- `webstart/CalculatorFrame.java`, 832
- `Welcome/Welcome.java`, 25
- `whenComplete` method (`CompletableFuture`), 933
- while loop, 94–99
- Whitespace, irrelevant to Java compiler,
 44
- Wildcard types, 417, 442–450
 arrays of, 432
 capturing, 448–450
 supertype bounds for, 444–447
 unbounded, 447
- WildcardType* interface, 453
`getLowerBounds`, `getUpperBounds` methods, 458
- Window class, 628
`is setLocationByPlatform` methods, 552
`pack` method, 550, 557, 560
`toBack/Front` methods, 552
- Window listeners, 603–607
- Window place, 630–631
- WindowAdapter class, 626
- WindowClosing event, 688
- WindowEvent class, 588, 603, 626
`getNewState`, `getOldState` methods, 607,
 628
`getWindow`, `getOppositeWindow`, `getScrollAmount`
 methods, 628
- WindowFocusListener interface, 626
`windowGainedFocus`, `windowLostFocus` methods,
 628
- WindowListener interface, 626
`windowActivated/Deactivated` methods, 603, 607,
 628
`windowClosing/Closed` methods, 603–607, 628
`windowIconified/Deiconified` methods, 603, 607,
 628
`windowOpened` method, 603, 606, 628
- Windows. *See* Dialogs
- Windows look-and-feel, 539–540
- Windows operating system
 Alt+F4 in, 688
 debugging applets in, 807
 default location in, 395
 device context in, 556
 Eclipse versions for, 27
 executing JARs in, 783
 file separators in, 785
 fonts shipped with, 574
 JDK versions for, 18
 pop-up trigger in, 685
 registry in, 794–795
 resources in, 783
 running applets in, 34–35
 setting paths in, 20, 191, 193
 setting up JDK in, 20
 thread priority levels in, 859
- WindowStateListener interface, 603, 626
`windowStateChanged` method, 607,
 628
- Wirth, Niklaus, 5, 10, 130
- withInitial method (`ThreadLocal`), 893

Work stealing, 930

Wrappers, 252–256

 equality testing for, 254

 immutability of, 253

 lightweight collection, 509–510

X

X11 programming, 556

XML (Extensible Markup Language), 12–13

xor method (`BitSet`), 533

Y

yield method (`Thread`), 859

Z

ZIP format, 191, 780