

ORACLE  
PRESS



# CORE JAVA

Volume I: Fundamentals

---

TWELFTH EDITION

ORACLE

Cay S. Horstmann

FREE SAMPLE CHAPTER |



---

# Core Java



## Volume I: Fundamentals

Twelfth Edition

---

*This page intentionally left blank*

---

# Core Java

## Volume I: Fundamentals

Twelfth Edition

---

**Cay S. Horstmann**



Pearson

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town

Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City

São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover image: Jon Chica/Shutterstock

Figure 1.1: © Jmol

Figures 2.1-2.3, 2.9: © Microsoft 2021

Figures 2.5-2.8: © Eclipse Foundation

Figures 3.2-3.5, 4.9, 3.2-3.5, 4.9, 5.4, 7.2, 7.3, 10.1, 10.3, 10.8, 10.10, 10.11, 10.14-10.16, 11.4, 11.5, 11.8-11.34, 12.5, 12.6: © 2021 Oracle

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.

The views expressed in this book are those of the author and do not necessarily reflect the views of Oracle.

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](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the United States, please contact [international@pearsoned.com](mailto:international@pearsoned.com).

Visit us on the Web: [informit.com](http://informit.com)

Library of Congress Preassigned Control Number: 2021946079

Copyright © 2022 Pearson Education Inc.

Portions copyright © 1996-2013 Oracle and/or its affiliates. All Rights Reserved.

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.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services. The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Microsoft® Windows®, and Microsoft Office® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

All rights reserved. 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.pearson.com/permissions/](http://www.pearson.com/permissions/).

ISBN-13: 978-0-13-767362-9

ISBN-10: 0-13-767362-0

ScoutAutomatedPrintCode

## **Pearson's Commitment to Diversity, Equity, and Inclusion**

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at <https://www.pearson.com/report-bias.html>.

*This page intentionally left blank*

---

# Contents

---



<i>Preface</i> .....	<i>xxi</i>
<i>Acknowledgments</i> .....	<i>xxvii</i>
<b>Chapter 1: An Introduction to Java</b> .....	<b>1</b>
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 .....	5
1.2.6 Architecture-Neutral .....	6
1.2.7 Portable .....	6
1.2.8 Interpreted .....	7
1.2.9 High-Performance .....	7
1.2.10 Multithreaded .....	8
1.2.11 Dynamic .....	8
1.3 Java Applets and the Internet .....	9
1.4 A Short History of Java .....	10
1.5 Common Misconceptions about Java .....	13
<b>Chapter 2: The Java Programming Environment</b> .....	<b>17</b>
2.1 Installing the Java Development Kit .....	18
2.1.1 Downloading the JDK .....	18
2.1.2 Setting Up the JDK .....	18
2.1.3 Installing Source Files and Documentation .....	20
2.2 Using the Command-Line Tools .....	22
2.3 Using an Integrated Development Environment .....	27
2.4 JShell .....	30



---

<b>Chapter 3: Fundamental Programming Structures in Java .....</b>	<b>35</b>
3.1 A Simple Java Program .....	36
3.2 Comments .....	39
3.3 Data Types .....	40
3.3.1 Integer Types .....	40
3.3.2 Floating-Point Types .....	42
3.3.3 The char Type .....	43
3.3.4 Unicode and the char Type .....	45
3.3.5 The boolean Type .....	46
3.4 Variables and Constants .....	46
3.4.1 Declaring Variables .....	47
3.4.2 Initializing Variables .....	48
3.4.3 Constants .....	49
3.4.4 Enumerated Types .....	50
3.5 Operators .....	51
3.5.1 Arithmetic Operators .....	51
3.5.2 Mathematical Functions and Constants .....	51
3.5.3 Conversions between Numeric Types .....	53
3.5.4 Casts .....	54
3.5.5 Assignment .....	55
3.5.6 Increment and Decrement Operators .....	56
3.5.7 Relational and boolean Operators .....	57
3.5.8 The Conditional Operator .....	58
3.5.9 Switch Expressions .....	58
3.5.10 Bitwise Operators .....	59
3.5.11 Parentheses and Operator Hierarchy .....	60
3.6 Strings .....	61
3.6.1 Substrings .....	62
3.6.2 Concatenation .....	62
3.6.3 Strings Are Immutable .....	63
3.6.4 Testing Strings for Equality .....	64
3.6.5 Empty and Null Strings .....	65
3.6.6 Code Points and Code Units .....	66
3.6.7 The String API .....	68
3.6.8 Reading the Online API Documentation .....	70

3.6.9	Building Strings .....	73
3.6.10	Text Blocks .....	74
3.7	Input and Output .....	76
3.7.1	Reading Input .....	76
3.7.2	Formatting Output .....	79
3.7.3	File Input and Output .....	82
3.8	Control Flow .....	85
3.8.1	Block Scope .....	85
3.8.2	Conditional Statements .....	86
3.8.3	Loops .....	89
3.8.4	Determinate Loops .....	94
3.8.5	Multiple Selections with <code>switch</code> .....	98
3.8.6	Statements That Break Control Flow .....	103
3.9	Big Numbers .....	106
3.10	Arrays .....	109
3.10.1	Declaring Arrays .....	109
3.10.2	Accessing Array Elements .....	111
3.10.3	The “for each” Loop .....	112
3.10.4	Array Copying .....	113
3.10.5	Command-Line Parameters .....	114
3.10.6	Array Sorting .....	115
3.10.7	Multidimensional Arrays .....	118
3.10.8	Ragged Arrays .....	121
<b>Chapter 4: Objects and Classes .....</b>	<b>125</b>	
4.1	Introduction to Object-Oriented Programming .....	126
4.1.1	Classes .....	127
4.1.2	Objects .....	128
4.1.3	Identifying Classes .....	129
4.1.4	Relationships between Classes .....	130
4.2	Using Predefined Classes .....	132
4.2.1	Objects and Object Variables .....	132
4.2.2	The <code>LocalDate</code> Class of the Java Library .....	136
4.2.3	Mutator and Accessor Methods .....	138
4.3	Defining Your Own Classes .....	141
4.3.1	An <code>Employee</code> Class .....	142

4.3.2	Use of Multiple Source Files .....	145
4.3.3	Dissecting the <code>Employee</code> Class .....	146
4.3.4	First Steps with Constructors .....	146
4.3.5	Declaring Local Variables with <code>var</code> .....	148
4.3.6	Working with <code>null</code> References .....	149
4.3.7	Implicit and Explicit Parameters .....	150
4.3.8	Benefits of Encapsulation .....	151
4.3.9	Class-Based Access Privileges .....	154
4.3.10	Private Methods .....	155
4.3.11	Final Instance Fields .....	155
4.4	Static Fields and Methods .....	156
4.4.1	Static Fields .....	156
4.4.2	Static Constants .....	157
4.4.3	Static Methods .....	158
4.4.4	Factory Methods .....	159
4.4.5	The <code>main</code> Method .....	160
4.5	Method Parameters .....	163
4.6	Object Construction .....	170
4.6.1	Overloading .....	170
4.6.2	Default Field Initialization .....	171
4.6.3	The Constructor with No Arguments .....	172
4.6.4	Explicit Field Initialization .....	173
4.6.5	Parameter Names .....	174
4.6.6	Calling Another Constructor .....	175
4.6.7	Initialization Blocks .....	175
4.6.8	Object Destruction and the <code>finalize</code> Method .....	180
4.7	Records .....	181
4.7.1	The Record Concept .....	182
4.7.2	Constructors: Canonical, Custom, and Compact .....	184
4.8	Packages .....	186
4.8.1	Package Names .....	186
4.8.2	Class Importation .....	187
4.8.3	Static Imports .....	189
4.8.4	Addition of a Class into a Package .....	190
4.8.5	Package Access .....	193

4.8.6	The Class Path .....	195
4.8.7	Setting the Class Path .....	197
4.9	JAR Files .....	198
4.9.1	Creating JAR files .....	198
4.9.2	The Manifest .....	199
4.9.3	Executable JAR Files .....	200
4.9.4	Multi-Release JAR Files .....	201
4.9.5	A Note about Command-Line Options .....	202
4.10	Documentation Comments .....	204
4.10.1	Comment Insertion .....	204
4.10.2	Class Comments .....	205
4.10.3	Method Comments .....	206
4.10.4	Field Comments .....	207
4.10.5	General Comments .....	207
4.10.6	Package Comments .....	208
4.10.7	Comment Extraction .....	209
4.11	Class Design Hints .....	210
<b>Chapter 5: Inheritance</b>	<b>.....</b>	<b>213</b>
5.1	Classes, Superclasses, and Subclasses .....	214
5.1.1	Defining Subclasses .....	214
5.1.2	Overriding Methods .....	216
5.1.3	Subclass Constructors .....	218
5.1.4	Inheritance Hierarchies .....	222
5.1.5	Polymorphism .....	223
5.1.6	Understanding Method Calls .....	225
5.1.7	Preventing Inheritance: Final Classes and Methods .....	228
5.1.8	Casting .....	229
5.1.9	Pattern Matching for instanceof .....	232
5.1.10	Protected Access .....	234
5.2	Object: The Cosmic Superclass .....	235
5.2.1	Variables of Type Object .....	235
5.2.2	The equals Method .....	236
5.2.3	Equality Testing and Inheritance .....	238
5.2.4	The hashCode Method .....	241
5.2.5	The toString Method .....	244

---

5.3	Generic Array Lists .....	251
5.3.1	Declaring Array Lists .....	252
5.3.2	Accessing Array List Elements .....	254
5.3.3	Compatibility between Typed and Raw Array Lists .....	258
5.4	Object Wrappers and Autoboxing .....	259
5.5	Methods with a Variable Number of Parameters .....	263
5.6	Abstract Classes .....	265
5.7	Enumeration Classes .....	271
5.8	Sealed Classes .....	273
5.9	Reflection .....	279
5.9.1	The Class Class .....	280
5.9.2	A Primer on Declaring Exceptions .....	283
5.9.3	Resources .....	284
5.9.4	Using Reflection to Analyze the Capabilities of Classes .....	287
5.9.5	Using Reflection to Analyze Objects at Runtime .....	294
5.9.6	Using Reflection to Write Generic Array Code .....	300
5.9.7	Invoking Arbitrary Methods and Constructors .....	304
5.10	Design Hints for Inheritance .....	308
<b>Chapter 6: Interfaces, Lambda Expressions, and Inner Classes .....</b>		<b>311</b>
6.1	Interfaces .....	312
6.1.1	The Interface Concept .....	312
6.1.2	Properties of Interfaces .....	319
6.1.3	Interfaces and Abstract Classes .....	321
6.1.4	Static and Private Methods .....	322
6.1.5	Default Methods .....	323
6.1.6	Resolving Default Method Conflicts .....	324
6.1.7	Interfaces and Callbacks .....	326
6.1.8	The Comparator Interface .....	329
6.1.9	Object Cloning .....	330
6.2	Lambda Expressions .....	338
6.2.1	Why Lambdas? .....	338
6.2.2	The Syntax of Lambda Expressions .....	339
6.2.3	Functional Interfaces .....	342
6.2.4	Method References .....	344
6.2.5	Constructor References .....	348

6.2.6	Variable Scope .....	349
6.2.7	Processing Lambda Expressions .....	352
6.2.8	More about Comparators .....	356
6.3	Inner Classes .....	357
6.3.1	Use of an Inner Class to Access Object State .....	358
6.3.2	Special Syntax Rules for Inner Classes .....	362
6.3.3	Are Inner Classes Useful? Actually Necessary? Secure? .....	363
6.3.4	Local Inner Classes .....	365
6.3.5	Accessing Variables from Outer Methods .....	366
6.3.6	Anonymous Inner Classes .....	367
6.3.7	Static Inner Classes .....	372
6.4	Service Loaders .....	376
6.5	Proxies .....	378
6.5.1	When to Use Proxies .....	379
6.5.2	Creating Proxy Objects .....	379
6.5.3	Properties of Proxy Classes .....	383
<b>Chapter 7: Exceptions, Assertions, and Logging .....</b>	<b>387</b>	
7.1	Dealing with Errors .....	388
7.1.1	The Classification of Exceptions .....	389
7.1.2	Declaring Checked Exceptions .....	391
7.1.3	How to Throw an Exception .....	394
7.1.4	Creating Exception Classes .....	396
7.2	Catching Exceptions .....	397
7.2.1	Catching an Exception .....	397
7.2.2	Catching Multiple Exceptions .....	399
7.2.3	Rethrowing and Chaining Exceptions .....	400
7.2.4	The finally Clause .....	402
7.2.5	The try-with-Resources Statement .....	405
7.2.6	Analyzing Stack Trace Elements .....	407
7.3	Tips for Using Exceptions .....	411
7.4	Using Assertions .....	415
7.4.1	The Assertion Concept .....	415
7.4.2	Assertion Enabling and Disabling .....	416
7.4.3	Using Assertions for Parameter Checking .....	417
7.4.4	Using Assertions for Documenting Assumptions .....	419

---

7.5	Logging .....	420
7.5.1	Basic Logging .....	421
7.5.2	Advanced Logging .....	421
7.5.3	Changing the Log Manager Configuration .....	424
7.5.4	Localization .....	426
7.5.5	Handlers .....	427
7.5.6	Filters .....	431
7.5.7	Formatters .....	431
7.5.8	A Logging Recipe .....	432
7.6	Debugging Tips .....	441
<b>Chapter 8: Generic Programming .....</b>		<b>447</b>
8.1	Why Generic Programming? .....	448
8.1.1	The Advantage of Type Parameters .....	448
8.1.2	Who Wants to Be a Generic Programmer? .....	449
8.2	Defining a Simple Generic Class .....	450
8.3	Generic Methods .....	453
8.4	Bounds for Type Variables .....	454
8.5	Generic Code and the Virtual Machine .....	457
8.5.1	Type Erasure .....	457
8.5.2	Translating Generic Expressions .....	458
8.5.3	Translating Generic Methods .....	459
8.5.4	Calling Legacy Code .....	461
8.6	Restrictions and Limitations .....	462
8.6.1	Type Parameters Cannot Be Instantiated with Primitive Types .....	463
8.6.2	Runtime Type Inquiry Only Works with Raw Types .....	463
8.6.3	You Cannot Create Arrays of Parameterized Types .....	463
8.6.4	Varargs Warnings .....	464
8.6.5	You Cannot Instantiate Type Variables .....	465
8.6.6	You Cannot Construct a Generic Array .....	466
8.6.7	Type Variables Are Not Valid in Static Contexts of Generic Classes .....	468
8.6.8	You Cannot Throw or Catch Instances of a Generic Class .....	469
8.6.9	You Can Defeat Checked Exception Checking .....	469

8.6.10	Beware of Clashes after Erasure .....	471
8.7	Inheritance Rules for Generic Types .....	472
8.8	Wildcard Types .....	475
8.8.1	The Wildcard Concept .....	475
8.8.2	Supertype Bounds for Wildcards .....	476
8.8.3	Unbounded Wildcards .....	480
8.8.4	Wildcard Capture .....	480
8.9	Reflection and Generics .....	483
8.9.1	The Generic Class Class .....	483
8.9.2	Using <code>Class&lt;T&gt;</code> Parameters for Type Matching .....	484
8.9.3	Generic Type Information in the Virtual Machine .....	485
8.9.4	Type Literals .....	489
<b>Chapter 9: Collections</b>	<b>.....</b>	<b>497</b>
9.1	The Java Collections Framework .....	498
9.1.1	Separating Collection Interfaces and Implementation ....	498
9.1.2	The Collection Interface .....	501
9.1.3	Iterators .....	501
9.1.4	Generic Utility Methods .....	504
9.2	Interfaces in the Collections Framework .....	508
9.3	Concrete Collections .....	510
9.3.1	Linked Lists .....	512
9.3.2	Array Lists .....	523
9.3.3	Hash Sets .....	523
9.3.4	Tree Sets .....	527
9.3.5	Queues and Deques .....	532
9.3.6	Priority Queues .....	533
9.4	Maps .....	535
9.4.1	Basic Map Operations .....	535
9.4.2	Updating Map Entries .....	539
9.4.3	Map Views .....	540
9.4.4	Weak Hash Maps .....	542
9.4.5	Linked Hash Sets and Maps .....	543
9.4.6	Enumeration Sets and Maps .....	545
9.4.7	Identity Hash Maps .....	545
9.5	Copies and Views .....	548



---

9.5.1	Small Collections .....	548
9.5.2	Unmodifiable Copies and Views .....	550
9.5.3	Subranges .....	552
9.5.4	Checked Views .....	553
9.5.5	Synchronized Views .....	553
9.5.6	A Note on Optional Operations .....	554
9.6	Algorithms .....	558
9.6.1	Why Generic Algorithms? .....	558
9.6.2	Sorting and Shuffling .....	560
9.6.3	Binary Search .....	563
9.6.4	Simple Algorithms .....	564
9.6.5	Bulk Operations .....	566
9.6.6	Converting between Collections and Arrays .....	567
9.6.7	Writing Your Own Algorithms .....	568
9.7	Legacy Collections .....	569
9.7.1	The Hashtable Class .....	570
9.7.2	Enumerations .....	570
9.7.3	Property Maps .....	572
9.7.4	Stacks .....	575
9.7.5	Bit Sets .....	576
<b>Chapter 10: Graphical User Interface Programming .....</b>		<b>581</b>
10.1	A History of Java User Interface Toolkits .....	582
10.2	Displaying Frames .....	583
10.2.1	Creating a Frame .....	584
10.2.2	Frame Properties .....	586
10.3	Displaying Information in a Component .....	590
10.3.1	Working with 2D Shapes .....	595
10.3.2	Using Color .....	603
10.3.3	Using Fonts .....	605
10.3.4	Displaying Images .....	612
10.4	Event Handling .....	614
10.4.1	Basic Event Handling Concepts .....	614
10.4.2	Example: Handling a Button Click .....	616
10.4.3	Specifying Listeners Concisely .....	620
10.4.4	Adapter Classes .....	621

---

10.4.5	Actions .....	623
10.4.6	Mouse Events .....	629
10.4.7	The AWT Event Hierarchy .....	636
10.5	The Preferences API .....	639
<b>Chapter 11:</b>	<b>User Interface Components with Swing .....</b>	<b>647</b>
11.1	Swing and the Model-View-Controller Design Pattern .....	648
11.2	Introduction to Layout Management .....	652
11.2.1	Layout Managers .....	652
11.2.2	Border Layout .....	655
11.2.3	Grid Layout .....	657
11.3	Text Input .....	658
11.3.1	Text Fields .....	659
11.3.2	Labels and Labeling Components .....	661
11.3.3	Password Fields .....	662
11.3.4	Text Areas .....	663
11.3.5	Scroll Panes .....	664
11.4	Choice Components .....	667
11.4.1	Checkboxes .....	667
11.4.2	Radio Buttons .....	670
11.4.3	Borders .....	673
11.4.4	Combo Boxes .....	676
11.4.5	Sliders .....	680
11.5	Menus .....	686
11.5.1	Menu Building .....	687
11.5.2	Icons in Menu Items .....	690
11.5.3	Checkbox and Radio Button Menu Items .....	691
11.5.4	Pop-Up Menus .....	692
11.5.5	Keyboard Mnemonics and Accelerators .....	694
11.5.6	Enabling and Disabling Menu Items .....	696
11.5.7	Toolbars .....	701
11.5.8	Tooltips .....	704
11.6	Sophisticated Layout Management .....	705
11.6.1	The Grid Bag Layout .....	706
11.6.1.1	The gridx, gridy, gridwidth, and gridheight Parameters .....	708

11.6.1.2	Weight Fields .....	708
11.6.1.3	The fill and anchor Parameters .....	709
11.6.1.4	Padding .....	709
11.6.1.5	Alternative Method to Specify the gridx, gridy, gridwidth, and gridheight Parameters .....	709
11.6.1.6	A Grid Bag Layout Recipe .....	710
11.6.1.7	A Helper Class to Tame the Grid Bag Constraints .....	710
11.6.2	Custom Layout Managers .....	716
11.7	Dialog Boxes .....	721
11.7.1	Option Dialogs .....	722
11.7.2	Creating Dialogs .....	726
11.7.3	Data Exchange .....	730
11.7.4	File Dialogs .....	737
<b>Chapter 12:</b>	<b>Concurrency .....</b>	<b>747</b>
12.1	What Are Threads? .....	748
12.2	Thread States .....	753
12.2.1	New Threads .....	754
12.2.2	Runnable Threads .....	754
12.2.3	Blocked and Waiting Threads .....	755
12.2.4	Terminated Threads .....	756
12.3	Thread Properties .....	757
12.3.1	Interrupting Threads .....	757
12.3.2	Daemon Threads .....	761
12.3.3	Thread Names .....	761
12.3.4	Handlers for Uncaught Exceptions .....	761
12.3.5	Thread Priorities .....	763
12.4	Synchronization .....	764
12.4.1	An Example of a Race Condition .....	764
12.4.2	The Race Condition Explained .....	766
12.4.3	Lock Objects .....	769
12.4.4	Condition Objects .....	772
12.4.5	The synchronized Keyword .....	778
12.4.6	Synchronized Blocks .....	782
12.4.7	The Monitor Concept .....	784

---

12.4.8	Volatile Fields .....	785
12.4.9	Final Variables .....	787
12.4.10	Atomics .....	787
12.4.11	Deadlocks .....	789
12.4.12	Why the stop and suspend Methods Are Deprecated .....	793
12.4.13	On-Demand Initialization .....	794
12.4.14	Thread-Local Variables .....	795
12.5	Thread-Safe Collections .....	797
12.5.1	Blocking Queues .....	797
12.5.2	Efficient Maps, Sets, and Queues .....	805
12.5.3	Atomic Update of Map Entries .....	806
12.5.4	Bulk Operations on Concurrent Hash Maps .....	810
12.5.5	Concurrent Set Views .....	812
12.5.6	Copy on Write Arrays .....	813
12.5.7	Parallel Array Algorithms .....	813
12.5.8	Older Thread-Safe Collections .....	814
12.6	Tasks and Thread Pools .....	815
12.6.1	Callables and Futures .....	816
12.6.2	Executors .....	818
12.6.3	Controlling Groups of Tasks .....	821
12.6.4	The Fork-Join Framework .....	827
12.7	Asynchronous Computations .....	830
12.7.1	Completable Futures .....	830
12.7.2	Composing Completable Futures .....	832
12.7.3	Long-Running Tasks in User Interface Callbacks .....	839
12.8	Processes .....	847
12.8.1	Building a Process .....	847
12.8.2	Running a Process .....	849
12.8.3	Process Handles .....	850
<b>Appendix</b>	.....	<b>855</b>
<i>Index</i>	.....	<i>861</i>

*This page intentionally left blank*

---

# 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, twelve major revisions of the Java Development Kit have been released. Over the course of the last 25 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 that you are reading right now is the first volume of the twelfth edition of *Core Java*. Each edition closely followed a release of the Java Development Kit, and each time, I rewrote the book to take advantage of the newest Java features. This edition has been updated to reflect the features of Java 17.

As with the previous editions, *this book still targets serious programmers who want to put Java to work on real projects*. I think of you, the reader, as a programmer with a solid background in a programming language other than Java. I 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 the book. My 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. The sample programs are 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.

I assume you are willing, even eager, to learn about all the advanced features that Java puts at your disposal. For example, you will find 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
- 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, the book is broken up into two volumes. This first volume 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. I'd very much like to know about them. But, of course, I'd prefer to learn about each of them only once. You will find a list of frequently asked questions and bug fixes 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 I don't answer

every query or don't get back to you immediately. I 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. The chapter explains what the designers of the language set out to do and to what extent they succeeded. A short history of Java follows, detailing how Java came into being and how it has evolved.

In **Chapter 2**, you will see how to download and install the JDK and the program examples for this book. Then I'll guide you through compiling and running a console application and a graphical application. You will see how to use the plain JDK, a Java IDE, and the JShell tool.

**Chapter 3** starts the discussion of the Java language. In this chapter, I 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, you will also find advice on sound OOP design. Finally, I 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 covering interfaces, I move on to *lambda expressions*, a concise way for expressing a block of code that can be executed at a later point in time. I then explain 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, I 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. I 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** provides an introduction into GUI programming. I 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. Next, you'll see how to write code that responds to events, such as mouse clicks or key presses.

**Chapter 11** 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 12** 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, I 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, I 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, I add a short summary description at the end of the section. These descriptions are a bit more informal but, hopefully, 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* 9

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. See Chapter 2 for more information on installing the Java Development Kit and the sample code.

Register your copy of *Core Java, Volume I: Fundamentals, Twelfth Edition*, on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to [informit.com/register](http://informit.com/register) and log in or create an account. Enter the product ISBN (9780137673629) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

---

# 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 Pearson 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 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), Gail Anderson (Anderson Software Group), 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), Marc Hoffmann (mtrail), Vladimir Ivanovic (PointBase), Jerry Jackson (CA Technologies), Heinz Kabutz (Java Specialists), Stepan V. Kalinin (I-Teco/Servionica LTD), Tim Kimmet (Walmart), Chris Laffra, Charlie Lai (Apple), Angelika Langer, Jeff Langr (Langr Software Solutions), 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), José Paumard (Oracle), Hao Pham, Paul

Philion, Blake Ragsdell, Stuart Reges (University of Arizona), Simon Ritter (Azul Systems), 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*  
*Berlin, Germany*  
*October 2021*

---

# Interfaces, Lambda Expressions, and Inner Classes

---

## In this chapter

- 6.1 Interfaces, page 312
- 6.2 Lambda Expressions, page 338
- 6.3 Inner Classes, page 357
- 6.4 Service Loaders, page 376
- 6.5 Proxies, page 378

You have now learned about classes and inheritance, the key concepts of 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 discussing interfaces,

we move on to *lambda expressions*, a concise way to create blocks 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 recent versions of Java.

### 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);
}
```

In the interface, the `compareTo` method is *abstract*—it has no implementation. A class that implements the `Comparable` interface needs to have a `compareTo` method, and the method must take an `Object` parameter and return an integer. Otherwise, the class is also abstract—that is, you cannot construct any objects.



**NOTE:** As of Java 5, 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. I 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 8, all methods in an interface were abstract. As you will see in Section 6.1.4, “Static and Private Methods,” on p. 322 and Section 6.1.5, “Default Methods,” on p. 323, it is now possible to have other methods in interfaces. Of course, those methods cannot refer to instance fields—interfaces don’t have any.

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 access—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  $(\text{Integer.MAX\_VALUE} - 1) / 2$ , 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).

**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         var 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 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 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 I follow that recommendation.

---

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`. (This interface is discussed in detail in Section 6.1.9, "Object Cloning," on p. 330.) 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
```



---

**NOTE:** Records and enumeration classes cannot extend other classes (since they implicitly extend the `Record` and `Enum` class). However, they can implement interfaces.

---



---

**NOTE:** Interfaces can be sealed. As with sealed classes, the direct subtypes (which can be classes or interfaces) must be declared in a `permits` clause or be located in the same source file.

---

### 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.



**TIP:** You have seen the `CharSequence` interface in Chapter 3. Both `String` and `StringBuilder` (as well as a few more esoteric string-like classes) implement this interface. The interface contains methods that are common to all classes that manage sequences of characters. A common interface encourages programmers to write methods that use the `CharSequence` interface. Those methods work with instances of `String`, `StringBuilder`, and the other string-like classes.

Sadly, the `CharSequence` interface is rather paltry. You can get the length, iterate over the code points or code units, extract subsequences, and lexicographically compare two sequences. Java 17 adds an `isEmpty` method.

If you process strings, and those operations suffice for your tasks, accept `CharSequence` instances instead of strings.

### 6.1.4 Static and Private Methods

As of Java 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’ll find pairs of interfaces and utility classes such as `Collection/Collections` or `Path/Paths`.

You can construct a path to a file or directory from a URI, or from a sequence of strings, such as `Paths.get("jdk-17", "conf", "security")`. In Java 11, equivalent methods are provided in the `Path` interface:

```
public interface Path
{
    public static Path of(URI uri) { . . . }
    public static Path of(String first, String... more) { . . . }
    . . .
}
```

Then the `Paths` class is no longer necessary.

Similarly, when you implement your own interfaces, there is no longer a reason to provide a separate companion class for utility methods.

As of Java 9, methods in an interface can be `private`. A `private` method can be `static` or an instance method. Since `private` methods can only be used in the methods of the interface itself, their use is limited to being helper methods for the other methods of the interface.

### 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, in Chapter 9 you will see an `Iterator` interface for visiting elements in a data structure. It declares a `remove` method as follows:

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    default void remove() { throw new UnsupportedOperationException("remove"); }
    . . .
}
```

If you implement an iterator, you need to provide the `hasNext` and `next` methods. There are no defaults for these methods—they depend on the data structure that you are traversing. But if your iterator is read-only, you don't have to worry about the `remove` method.

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:** The `Collection` interface in the Java API does not actually do this. Instead, there is a class `AbstractCollection` that implements `Collection` and defines `isEmpty` in terms of `size`. Implementors of a collection are advised to extend `AbstractCollection`. That 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 8, a `stream` method was added to the interface.

Suppose the `stream` method was not a default method. Then the `Bag` class would no longer compile 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 an interface provides a default method, and another interface contains 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 two interfaces with a `getName` method:

```
interface Person
{
    default String getName() { return ""; };
}

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 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 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 "class wins" rule, such a method could never win against `Object.toString` or `Object.equals`.

---

### 6.1.7 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 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 second. 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 "
            + Instant.ofEpochMilli(event.getWhen()));
        Toolkit.getDefaultToolkit().beep();
    }
}
```

Note the `ActionEvent` parameter of the `actionPerformed` method. This parameter gives information about the event, such as the time when the event happened. The call `event.getWhen()` returns the event time, measured in milliseconds since the “epoch” (January 1, 1970). By passing it to the static `Instant.ofEpochMilli` method, we get a more readable description.

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

```
var listener = new TimePrinter();
Timer t = new Timer(1000, 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 second. The second parameter is the listener object.

Finally, start the timer.

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

Every second, a message like

```
At the tone, the time is 2017-12-16T05:01:49.550Z
```

is displayed, followed by a beep.



**CAUTION:** Be sure to import `javax.swing.Timer`. There is also a `java.util.Timer` class that is slightly different.

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 every second. (If you omit the dialog, the program would terminate as soon as the `main` method exits.)

**Listing 6.3** timer/TimerTest.java

```
1 package timer;
2
3 /**
4  * @version 1.02 2017-12-14
5  * @author Cay Horstmann
6  */
7
8 import java.awt.*;
9 import java.awt.event.*;
10 import java.time.*;
11 import javax.swing.*;
12
13 public class TimerTest
14 {
15     public static void main(String[] args)
16     {
17         var listener = new TimePrinter();
18
19         // construct a timer that calls the listener once every second
20         var timer = new Timer(1000, listener);
21         timer.start();
22
23         // keep program running until the user selects "OK"
24         JOptionPane.showMessageDialog(null, "Quit program?");
25         System.exit(0);
26     }
27 }
28
29 class TimePrinter implements ActionListener
30 {
31     public void actionPerformed(ActionEvent event)
32     {
33         System.out.println("At the tone, the time is " + Instant.ofEpochMilli(event.getWhen()));
34         Toolkit.getDefaultToolkit().beep();
35     }
36 }
```

**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.

### 6.1.8 The Comparator Interface

In Section 6.1.1, “The Interface Concept,” on p. 312, 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:

```
var 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"]`.

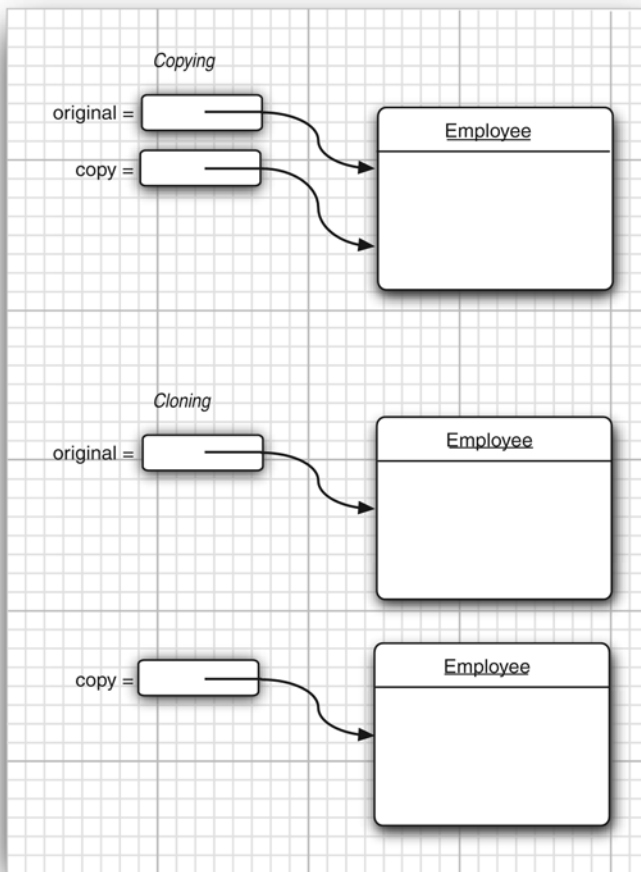
You will see in Section 6.2, “Lambda Expressions,” on p. 338 how to use a `Comparator` much more easily with a lambda expression.

### 6.1.9 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.

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



**Figure 6.1** Copying and cloning

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.

```
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 instance 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`.)

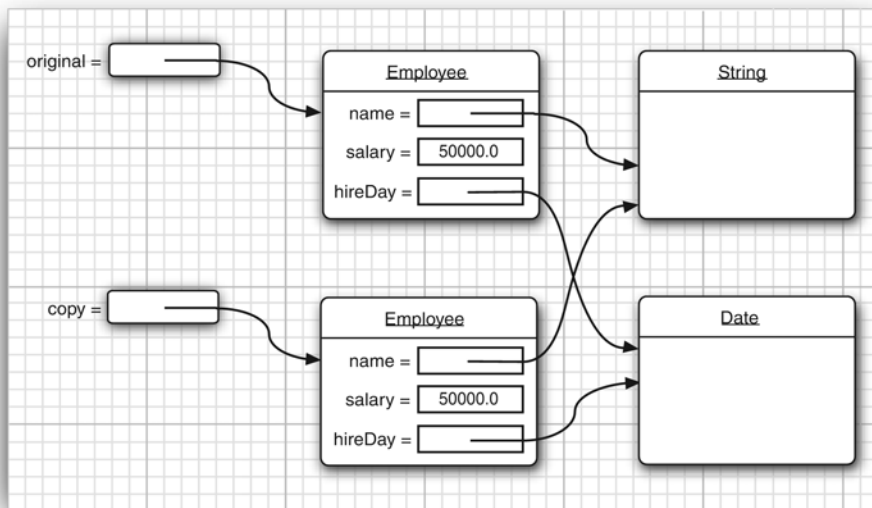


Figure 6.2 A shallow copy

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; or
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) . . .
```

I 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
{
    // public access, change return type
    public Employee clone() throws CloneNotSupportedException
    {
        return (Employee) super.clone();
    }
    . . .
}
```



**NOTE:** Up to Java 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
```



**NOTE:** Would it be better to catch the exception instead? (See Chapter 7 for details on catching exceptions.)

```

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 better 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 the 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. I 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 five 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 1.11 2018-03-16
6  * @author Cay Horstmann
7  */
8 public class CloneTest
9 {
10     public static void main(String[] args) throws CloneNotSupportedException
11     {
12         var original = new Employee("John Q. Public", 50000);
13         original.setHireDay(2000, 1, 1);
14         Employee copy = original.clone();
15         copy.raiseSalary(10);
16         copy.setHireDay(2002, 12, 31);
17         System.out.println("original=" + original);
18         System.out.println("copy=" + copy);
19     }
20 }
```

**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    {
```

(Continues)



**Listing 6.5** *(Continued)*

```
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 }
```

## 6.2 Lambda Expressions

In the following sections, you will learn how to use lambda expressions for defining blocks of code with a concise syntax, and how to write code that consumes lambda expressions.

### 6.2.1 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 6.1.7, "Interfaces and Callbacks," on p. 326, 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 taking objects of classes that implement a particular interface, but such APIs would be unpleasant to use.

For some time, 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.

## 6.2.2 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 ( $\lambda$ ) 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  $\lambda$ ? 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  $\Lambda$  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.

What you have just seen is a simple form of lambda expressions in Java: parameters, the `->` 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 "
        + Instant.ofEpochMilli(event.getWhen()));
    // 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.

Finally, you can use `var` to denote an inferred type. This isn't common. The syntax was invented for attaching annotations (see Chapter 8 of Volume II):

```
(@NonNull var first, @NonNull var second) -> first.length() - second.length()
```



**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 import javax.swing.*;
5 import javax.swing.Timer;
6
7 /**
8  * This program demonstrates the use of lambda expressions.
9  * @version 1.0 2015-05-12
10 * @author Cay Horstmann
11 */
```

(Continues)

**Listing 6.6** (Continued)

```
12 public class LambdaTest
13 {
14     public static void main(String[] args)
15     {
16         var planets = new String[] { "Mercury", "Venus", "Earth", "Mars",
17             "Jupiter", "Saturn", "Uranus", "Neptune" };
18         System.out.println(Arrays.toString(planets));
19         System.out.println("Sorted in dictionary order:");
20         Arrays.sort(planets);
21         System.out.println(Arrays.toString(planets));
22         System.out.println("Sorted by length:");
23         Arrays.sort(planets, (first, second) -> first.length() - second.length());
24         System.out.println(Arrays.toString(planets));
25
26         var timer = new Timer(1000, event ->
27             System.out.println("The time is " + new Date()));
28         timer.start();
29
30         // keep program running until user selects "OK"
31         JOptionPane.showMessageDialog(null, "Quit program?");
32         System.exit(0);
33     }
34 }
```

## 6.2.3 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. 323, 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:

```
var timer = new Timer(1000, event ->
{
    System.out.println("At the tone, the time is "
        + Instant.ofEpochMilli(event.getWhen()));
    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 your 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. 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);
```

Another useful functional interface is `Supplier<T>`:

```
public interface Supplier<T>
{
    T get();
}
```

A supplier has no arguments and yields a value of type `T` when it is called. Suppliers are used for *lazy evaluation*. For example, consider the call

```
LocalDate hireDay = Objects.requireNonNullElse(day,
    LocalDate.of(1970, 1, 1));
```

This is not optimal. We expect that `day` is rarely `null`, so we only want to construct the default `LocalDate` when necessary. By using the supplier, we can defer the computation:

```
LocalDate hireDay = Objects.requireNonNullElseGet(day,
    () -> LocalDate.of(1970, 1, 1));
```

The `requireNonNullElseGet` method only calls the supplier when the value is needed.

### 6.2.4 Method References

Sometimes, a lambda expression involves a single method. For example, suppose you simply want to print the event object whenever a timer event occurs. Of course, you could call

```
var timer = 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:

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

The expression `System.out::println` is a *method reference*. It directs the compiler to produce an instance of a functional interface, overriding the single abstract method of the interface to call the given method. In this example, an `ActionListener` is produced whose `actionPerformed(ActionEvent e)` method calls `System.out.println(e)`.



---

**NOTE:** Like a lambda expression, a method reference is not an object. It gives rise to an object when assigned to a variable whose type is a functional interface.

---



---

**NOTE:** There are ten overloaded `println` methods in the `PrintStream` class (of which `System.out` is an instance). The compiler needs to figure out which one to use, depending on context. In our example, the method reference `System.out::println` must be turned into an `ActionListener` instance with a method

```
void actionPerformed(ActionEvent e)
```

The `println(Object x)` method is selected from the ten overloaded `println` methods since `Object` is the best match for `ActionEvent`. When the `actionPerformed` method is called, the event object is printed.

Now suppose we assign the same method reference to a different functional interface:

```
Runnable task = System.out::println;
```

The `Runnable` functional interface has a single abstract method with no parameters

```
void run()
```

In this case, the `println()` method with no parameters is chosen. Calling `task.run()` prints a blank line to `System.out`.

---

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 variants:



1. `object::instanceMethod`
2. `Class::instanceMethod`
3. `Class::staticMethod`

In the first variant, the method reference is equivalent to a lambda expression whose parameters are passed to the method. In the case of `System.out::println`, the object is `System.out`, and the method expression is equivalent to `x -> System.out.println(x)`.

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

In the third variant, all parameters are passed to the static method: `Math::pow` is equivalent to `(x, y) -> Math.pow(x, y)`.

Table 6.1 walks you through additional examples.

Note that a lambda expression can only be rewritten as a method reference if the body of the lambda expression calls a single method and doesn't do anything else. Consider the lambda expression

```
s -> s.length() == 0
```

There is a single method call. But there is also a comparison, so you can't use a method reference here.



**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.



**NOTE:** Sometimes, the API contains methods that are specifically intended to be used as method references. For example, the `Objects` class has a method `isNull` to test whether an object reference is null. At first glance, this doesn't seem useful because the test `obj == null` is easier to read than `Objects.isNull(obj)`. But you can pass the method reference to any method with a `Predicate` parameter. For example, to remove all null references from a list, you can call

```
list.removeIf(Objects::isNull);  
// A bit easier to read than list.removeIf(e -> e == null);
```

**Table 6.1** Method Reference Examples

Method Reference	Equivalent Lambda Expression	Notes
<code>separator::equals</code>	<code>x -&gt; separator.equals(x)</code>	This is a method expression with an <i>object</i> and an instance method. The lambda parameter is passed as the explicit parameter of the method.
<code>String::trim</code>	<code>x -&gt; x.strip()</code>	This is a method expression with a <i>class</i> and an instance method. The lambda parameter becomes the implicit parameter.
<code>String::concat</code>	<code>(x, y) -&gt; x.concat(y)</code>	Again, we have an instance method, but this time, with an explicit parameter. As before, the <i>first</i> lambda parameter becomes the implicit parameter, and the remaining ones are passed to the method.
<code>Integer.valueOf</code>	<code>x -&gt; Integer.valueOf(x)</code>	This is a method expression with a <i>static</i> method. The lambda parameter is passed to the static method.
<code>Integer.sum</code>	<code>(x, y) -&gt; Integer.sum(x, y)</code>	This is another static method, but this time with two parameters. Both lambda parameters are passed to the static method. The <code>Integer.sum</code> method was specifically created to be used as a method reference. As a lambda, you could just write <code>(x, y) -&gt; x + y</code> .
<code>String::new</code>	<code>x -&gt; new String(x)</code>	This is a constructor reference—see Section 6.2.5. The lambda parameters are passed to the constructor.
<code>String[]::new</code>	<code>n -&gt; new String[n]</code>	This is an array constructor reference—see Section 6.2.5. The lambda parameter is the array length.



**NOTE:** There is a tiny difference between a method reference with an object and its equivalent lambda expression. Consider a method reference such as `separator::equals`. If `separator` is `null`, forming `separator::equals` immediately throws a `NullPointerException`. The lambda expression `x -> separator.equals(x)` only throws a `NullPointerException` if it is invoked.

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(ActionEvent event)
    {
        System.out.println("Hello, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
    }
}

class RepeatedGreeter extends Greeter
{
    public void greet(ActionEvent event)
    {
        var timer = new Timer(1000, super::greet);
        timer.start();
    }
}
```

When the `RepeatedGreeter.greet` method starts, a `Timer` is constructed that executes the `super::greet` method on every timer tick.

## 6.2.5 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.toList();
```

We will discuss the details of the `stream`, `map`, and `toList` 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 `n -> new int[n]`.

Array constructor references are useful to overcome a limitation of Java. As you will see in Chapter 8, 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.

### 6.2.6 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 12 for more information on this important issue.

It is also illegal to refer, in a lambda expression, to a variable 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 = Path.of("/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.

### 6.2.7 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 6.2 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();
}
```

**Table 6.2** 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, not
BiPredicate<T, U>	T, U	boolean	test	A boolean-valued function with two arguments	and, or, negate

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 6.3 lists the 34 available specializations for primitive types `int`, `long`, and `double`. As you will see in Chapter 8, it is more efficient to use these specializations than the generic interfaces. For that reason, I used an `IntConsumer` instead of a `Consumer<Integer>` in the example of the preceding section.

**Table 6.3** 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	<i>p</i>	<code>getAsP</code>
<code>PConsumer</code>	<i>p</i>	<code>void</code>	<code>accept</code>
<code>ObjPConsumer&lt;T&gt;</code>	<i>T, p</i>	<code>void</code>	<code>accept</code>
<code>PFunction&lt;T&gt;</code>	<i>p</i>	<i>T</i>	<code>apply</code>
<code>PToQFunction</code>	<i>p</i>	<i>q</i>	<code>applyAsQ</code>
<code>ToPFunction&lt;T&gt;</code>	<i>T</i>	<i>p</i>	<code>applyAsP</code>
<code>ToPBiFunction&lt;T, U&gt;</code>	<i>T, U</i>	<i>p</i>	<code>applyAsP</code>
<code>PUnaryOperator</code>	<i>p</i>	<i>p</i>	<code>applyAsP</code>
<code>PBinaryOperator</code>	<i>p, p</i>	<i>p</i>	<code>applyAsP</code>
<code>PPredicate</code>	<i>p</i>	<code>boolean</code>	<code>test</code>



**TIP:** It is a good idea to use an interface from Tables 6.2 or 6.3 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 abstract 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.



**NOTE:** Some programmers love chains of method calls, such as

```
String input = " 618970019642690137449562111 ";
boolean isPrime = input.strip().transform(BigInteger::new).isProbablePrime(20);
```

The `transform` method of the `String` class (added in Java 12) applies a `Function` to the string and yields the result. You could have equally well written

```
boolean prime = new BigInteger(input.strip()).isProbablePrime(20);
```

But then your eyes jump inside-out and left-to-right to find out what happens first and what happens next: Calling `strip`, then constructing the `BigInteger`, and finally testing if it is a probable prime.

I am not sure that the eyes-jumping-inside-out-and-left-to-right is a huge problem. But if you prefer the orderly left-to-right sequence of chained method calls, then `transform` is your friend.

Sadly, it only works for strings. Why isn't there a `transform(java.util.function.Function)` method in the `Object` class?

The Java API designers weren't fast enough. They had one chance to do this right—in Java 8, when the `java.util.function.Function` interface was added to the API. Up to that point, nobody could have added a `transform(java.util.function.Function)` method to their own classes. But in Java 12, it was too late. Someone somewhere could have defined `transform(java.util.function.Function)` in their class, with a different meaning. Admittedly, it is unlikely that this ever happened, but there is no way to know.

That is how Java works. It takes its commitments seriously, and won't renege on them for convenience.

---

### 6.2.8 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()`.

## 6.3 Inner Classes

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

- Inner classes can be hidden from other classes in the same package.
- 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 used to be very important for concisely implementing callbacks, but nowadays lambda expressions do a much better job. Still, inner classes can be very useful for structuring your code. The following sections walk you through all the details.



**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:
        Link* current;
        LinkedList* owner;
    };
    . . .
private:
    Link* head;
    Link* tail;
};
```

Nested classes are similar to inner classes in Java. 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. For example, in Java, the `Iterator` class would not need an explicit pointer to the `LinkedList` into which it points.

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

### 6.3.1 Use of an Inner Class to Access Object State

The syntax for inner classes is rather complex. For that reason, I present a simple but somewhat artificial example to demonstrate the use of inner classes. Let's 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 "
            + Instant.ofEpochMilli(event.getWhen()));
        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`. As you can see, an inner class method gets to access both its own instance 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).

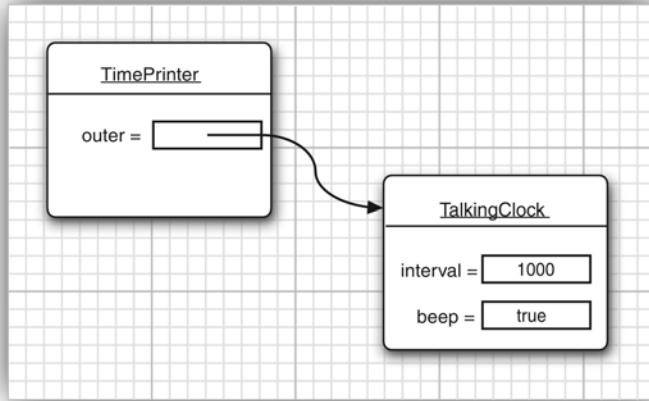
This reference is invisible in the definition of the inner class. However, to illuminate the concept, let us call the reference to the outer object *outer*. Then the `actionPerformed` method is equivalent to the following:

```

public void actionPerformed(ActionEvent event)
{
    System.out.println("At the tone, the time is "
        + Instant.ofEpochMilli(event.getWhen()));
    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:



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

```
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:

```
var 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 access.

**Listing 6.7** innerClass/InnerClassTest.java

```
1 package innerClass;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.time.*;
6
7 import javax.swing.*;
8
9 /**
10  * This program demonstrates the use of inner classes.
11  * @version 1.11 2017-12-14
12  * @author Cay Horstmann
13  */
14 public class InnerClassTest
15 {
16     public static void main(String[] args)
17     {
18         var clock = new TalkingClock(1000, true);
19         clock.start();
20
21         // keep program running until the 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     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 }
```

(Continues)



**Listing 6.7** (Continued)

```

46  /**
47   * Starts the clock.
48   */
49  public void start()
50  {
51      var listener = new TimePrinter();
52      var timer = new Timer(interval, listener);
53      timer.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 "
61              + Instant.ofEpochMilli(event.getWhen()));
62          if (beep) Toolkit.getDefaultToolkit().beep();
63      }
64  }
65 }

```

**6.3.2 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:

```
var 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 and initialized with a compile-time constant. If the field was not a constant, 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.

### 6.3.3 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.

Inner classes are translated into regular class files with \$ (dollar signs) separating the outer and inner class names. 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 --classpath ../v1ch05 reflection.ReflectionTest \
    innerClass.TalkingClock$TimePrinter
```

or

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

You will get the following printout:

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

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()
    {
        var listener = new TimePrinter(this);
        var timer = new Timer(interval, listener);
        timer.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. Before Java 11, inner classes were purely a phenomenon of the *compiler*, and the virtual machine did not have any special knowledge about them. In those days, spying on the `TalkingClock` class with the `ReflectionTest` program or with `javap` and the `-private` option showed:

```
class TalkingClock  
{  
    private int interval;  
    private boolean beep;  
  
    public TalkingClock(int, boolean);  
  
    static boolean access$0(TalkingClock); // Prior to Java 11  
    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 the compiler.)

That was a potential security risk, and it made life complicated for tools that analyze class files. As of Java 11, the virtual machine understands nesting relationships between classes, and the access methods are no longer generated.

### 6.3.4 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 "
                + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }

    var listener = new TimePrinter();
    var timer = new Timer(interval, listener);
    timer.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.

### 6.3.5 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 "
                + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    }

    var listener = new TimePrinter();
    var timer = new Timer(interval, listener);
    timer.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 or the javap utility again to spy on the TalkingClock\$1TimePrinter class, you will get the following output:

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

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

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

When an object is created, the current value of the beep variable is stored in the val\$beep field. As of Java 11, this happens with “nest mate” access. Previously, the inner class constructor had an additional parameter to set the field. Either way, the inner class field persists even if the local variable goes out of scope.

### 6.3.6 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)
{
    var listener = new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            System.out.println("At the tone, the time is "
                + Instant.ofEpochMilli(event.getWhen()));
            if (beep) Toolkit.getDefaultToolkit().beep();
        }
    };
    var timer = new Timer(interval, listener);
    timer.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.

```

var queen = new Person("Mary");
// a Person object
var 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.



**NOTE:** Even though an anonymous class cannot have constructors, you can provide an object initialization block:

```
var count = new Person("Dracula")
    {
        { initialization }
        . . .
    };
```

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)
{
    var timer = new Timer(interval, event ->
    {
        System.out.println(
            "At the tone, the time is " + Instant.ofEpochMilli(event.getWhen()));
        if (beep) Toolkit.getDefaultToolkit().beep();
    });
    timer.start();
}
```



**NOTE:** If you store an anonymous class instance in a variable defined with `var`, the variable knows about added methods or fields:

```
var bob = new Object() { String name = "Bob"; }
System.out.println(bob.name);
```

If you declare `bob` as having type `Object`, then `bob.name` does not compile.

The object constructed with `new Object() { String name = "Bob"; }` has type “Object with a String name field.” This is a *nondenotable* type—a type that you cannot express with Java syntax. Nevertheless, the compiler understands the type, and it can set it as the type for the `bob` variable.





**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:

```
var friends = new ArrayList<String>();
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 initialization block (see Chapter 4).

In practice, this trick is rarely useful. More likely than not, the `invite` method is willing to accept any `List<String>`, and you can simply pass `List.of("Harry", "Tony")`.



**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, I 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.time.*;
6
7 import javax.swing.*;
8
9 /**
10  * This program demonstrates anonymous inner classes.
11  * @version 1.12 2017-12-14
12  * @author Cay Horstmann
13  */
14 public class AnonymousInnerClassTest
15 {
16     public static void main(String[] args)
17     {
18         var clock = new TalkingClock();
19         clock.start(1000, true);
20
21         // keep program running until the 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         var listener = new ActionListener()
40         {
41             public void actionPerformed(ActionEvent event)
42             {
43                 System.out.println("At the tone, the time is "
44                     + Instant.ofEpochMilli(event.getWhen()));
45                 if (beep) Toolkit.getDefaultToolkit().beep();
46             }
47         };
48         var timer = new Timer(interval, listener);
49         timer.start();
50     }
51 }
```

### 6.3.7 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 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:** Classes that are declared inside an interface are automatically static and public.



**NOTE:** Interfaces, records, and enumerations that are declared inside a class are automatically static.

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         var values = new double[20];
13         for (int i = 0; i < values.length; i++)
14             values[i] = 100 * Math.random();
15         ArrayAlg.Pair p = ArrayAlg.minmax(values);
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 }
```

## 6.4 Service Loaders

Sometimes, you develop an application with a service architecture. There are platforms that encourage this approach, such as OSGi (<http://osgi.org>), which are used in development environments, application servers, and other complex applications. Such platforms go well beyond the scope of this book, but the JDK also offers a simple mechanism for loading services, described here. This mechanism is well supported by the Java Platform Module System—see Chapter 9 of Volume II.

Often, when providing a service, a program wants to give the service designer some freedom of how to implement the service's features. It can also be desirable to have multiple implementations to choose from. The `ServiceLoader` class makes it easy to load services that conform to a common interface.

Define an interface (or, if you prefer, a superclass) with the methods that each instance of the service should provide. For example, suppose your service provides encryption.

```
package serviceLoader;

public interface Cipher
{
    byte[] encrypt(byte[] source, byte[] key);
    byte[] decrypt(byte[] source, byte[] key);
    int strength();
}
```

The service provider supplies one or more classes that implement this service, for example

```
package serviceLoader.impl;

public class CaesarCipher implements Cipher
{
    public byte[] encrypt(byte[] source, byte[] key)
    {
        var result = new byte[source.length];
        for (int i = 0; i < source.length; i++)
            result[i] = (byte)(source[i] + key[0]);
        return result;
    }

    public byte[] decrypt(byte[] source, byte[] key)
    {
        return encrypt(source, new byte[] { (byte) -key[0] });
    }
}
```

```
    public int strength() { return 1; }  
}
```

The implementing classes can be in any package, not necessarily the same package as the service interface. Each of them must have a no-argument constructor.

Now add the names of the classes to a UTF-8 encoded text file in a file in the META-INF/services directory whose name matches the fully qualified interface name. In our example, the file META-INF/services/serviceLoader.Cipher would contain the line

```
serviceLoader.impl.CaesarCipher
```

In this example, we provide a single implementing class. You could also provide multiple classes and later pick among them.

With this preparation done, the program initializes a service loader as follows:

```
public static ServiceLoader<Cipher> cipherLoader = ServiceLoader.load(Cipher.class);
```

This should be done just once in the program.

The iterator method of the service loader returns an iterator through all provided implementations of the service. (See Chapter 9 for more information about iterators.) It is easiest to use an enhanced for loop to traverse them. In the loop, pick an appropriate object to carry out the service.

```
public static Cipher getCipher(int minStrength)  
{  
    for (Cipher cipher : cipherLoader) // implicitly calls cipherLoader.iterator()  
    {  
        if (cipher.strength() >= minStrength) return cipher;  
    }  
    return null;  
}
```

Alternatively, you can use streams (see Chapter 1 of Volume II) to locate the desired service. The stream method yields a stream of `ServiceLoader.Provider` instances. That interface has methods `type` and `get` for getting the provider class and the provider instance. If you select a provider by `type`, then you just call `type` and no service instances are unnecessarily instantiated.

```
public static Optional<Cipher> getCipher2(int minStrength)  
{  
    return cipherLoader.stream()  
        .filter(descr -> descr.type() == serviceLoader.impl.CaesarCipher.class)  
        .findFirst()  
        .map(ServiceLoader.Provider::get);  
}
```



Finally, if you are willing to take any service instance, simply call `findFirst`:

```
Optional<Cipher> cipher = cipherLoader.findFirst();
```

The `Optional` class is explained in Chapter 1 of Volume II.

#### **`java.util.ServiceLoader<S>` 1.6**

- `static <S> ServiceLoader<S> load(Class<S> service)`  
creates a service loader for loading the classes that implement the given service interface.
- `Iterator<S> iterator()`  
yields an iterator that lazily loads the service classes. That is, a class is loaded whenever the iterator advances.
- `Stream<ServiceLoader.Provider<S>> stream() 9`  
returns a stream of provider descriptors, so that a provider of a desired class can be loaded lazily.
- `Optional<S> findFirst() 9`  
finds the first available service provider, if any.

#### **`java.util.ServiceLoader.Provider<S>` 9**

- `Class<? extends S> type()`  
gets the type of this provider.
- `S get()`  
gets an instance of this provider.

## 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, so 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 platform and application classes, classes that are downloaded from the Internet, and so on. We will discuss class loaders in Chapter 9 of Volume II. In this example, we specify the “system class loader” that loads platform and application classes.
- 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
var handler = new TraceHandler(value);
// construct proxy for one or more interfaces
var interfaces = new Class[] { Comparable.class};
Object proxy = Proxy.newProxyInstance(
    ClassLoader.getSystemClassLoader(),
    new Class[] { Comparable.class } , 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.

```
var 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 = (int) (Math.random() * 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 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 1.02 2021-06-16
9  * @author Cay Horstmann
10 */
11 public class ProxyTest
12 {
13     public static void main(String[] args)
14     {
15         var 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             var handler = new TraceHandler(value);
22             Object proxy = Proxy.newProxyInstance(
23                 ClassLoader.getSystemClassLoader(),
24                 new Class[] { Comparable.class }, handler);
25             elements[i] = proxy;
26         }
27
28         // construct a random integer
29         Integer key = (int) (Math.random() * elements.length) + 1;
30
31         // search for the key
32         int result = Arrays.binarySearch(elements, key);
33
```

```
34     // print match if found
35     if (result >= 0) System.out.println(elements[result]);
36 }
37 }
38
39 /**
40  * An invocation handler that prints out the method name and parameters, then
41  * invokes the original method
42  */
43 class TraceHandler implements InvocationHandler
44 {
45     private Object target;
46
47     /**
48      * Constructs a TraceHandler
49      * @param t the implicit parameter of the method call
50      */
51     public TraceHandler(Object t)
52     {
53         target = t;
54     }
55
56     public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
57     {
58         // print implicit argument
59         System.out.print(target);
60         // print method name
61         System.out.print("." + m.getName() + "(");
62         // print explicit arguments
63         if (args != null)
64         {
65             for (int i = 0; i < args.length; i++)
66             {
67                 System.out.print(args[i]);
68                 if (i < args.length - 1) System.out.print(", ");
69             }
70         }
71         System.out.println(")");
72
73         // invoke actual method
74         return m.invoke(target, args);
75     }
76 }
```

### 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.



**NOTE:** Calling a default method of a proxy triggers the invocation handler.

To actually invoke the method, use the static `invokeDefault` method of the `InvocationHandler` interface. For example, here is an invocation handler that calls the default methods and passes the abstract methods to another target:

```
InvocationHandler handler = (proxy, method, args) ->
{
    if (method.isDefault())
        return InvocationHandler.invokeDefault(proxy, method, args)
    else
        return method.invoke(target, args);
}
```

**`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.
- `static Object invokeDefault(Object proxy, Method method, Object... args)` 16  
invokes a default method of the proxy instance with the given arguments, bypassing the invocation handler.

**`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 the final chapter on the object-oriented features of the Java programming language. Interfaces, lambda expressions, and inner classes are concepts that you will encounter frequently, whereas cloning, service loaders, 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

## Symbols

- (minus sign)
  - arithmetic operator, 51, 61
  - printf flag, 81
- operator, 56, 61
- >
  - in lambda expressions, 339–342
  - in switch expressions, 101–103
- \_ (underscore)
  - as a reserved word, 47, 859
  - delimiter, in number literals, 41
  - in instance field names (C++), 174
- , (comma)
  - operator (C++), 61
  - printf flag, 81
- ;(semicolon)
  - in class path (Windows), 195
  - in statements, 38, 47
- : (colon)
  - in assertions, 416
  - in class path (UNIX), 195
  - inheritance token (C++), 214
- :: operator (C++), 151, 159, 218, 345
- ! operator, 57, 61
- != operator, 57, 61, 96
- ? operator, 58, 61
  - with pattern matching, 233
- / (slash), arithmetic operator, 51, 61
- // comments, 39
- /\* . . . \*/ comments, 39
- /\*\* . . . \*/ comments, 40, 204–205
- . (period), in class paths, 195–196
- ... (ellipsis), in varargs, 264
- ^ operator, 59, 61, 340
- ~ operator, 59, 61
- ' , " (single, double quote), escape sequences for, 44
- ". . ." (double quotes), for strings, 39
- "" . . . "" (triple quotes), for text blocks, 74
- ( (left parenthesis), printf flag, 81
- (. . .) (parentheses)
  - empty, in method calls, 39
  - for casts, 54, 61, 230
  - for operator hierarchy, 60–61
- [. . .] (square brackets)
  - empty, in generics, 453
  - for arrays, 109, 114
- {. . .} (curly braces)
  - double, in inner classes, 370
  - for blocks, 38, 85
  - for enumerated type, 50
  - in lambda expressions, 340
- @ (at), in javadoc comments, 205–206
- \$ (dollar sign)
  - delimiter, for inner classes, 363
  - in variable names, 47
  - printf flag, 81
- \* (asterisk)
  - arithmetic operator, 51, 61
  - echo character, 662
  - in class path, 195
  - in imports, 188
- \ (backslash)
  - escape sequence for, 44
  - in file names, 82
  - in text blocks, 75
- & (ampersand)
  - bitwise operator, 59, 61
  - in bounding types, 455
  - in reference parameters (C++), 168
- && operator, 57, 61
- # (number sign)
  - in javadoc hyperlinks, 208
  - printf flag, 81
- % (percent sign)
  - arithmetic operator, 51, 61
  - conversion character, 80
- + (plus sign)
  - arithmetic operator, 51, 54, 61
  - for objects and strings, 62–63, 245–246
  - printf flag, 81

- ++ operator, 56, 61
  - < (left angle bracket)
    - in shell syntax, 84
    - printf flag, 81
    - relational operator, 57, 61
  - <? (in wildcard types), 475
  - << operator, 60–61
  - <= operator, 57, 61
  - <. . .> (angle brackets), for type
    - parameters, 252, 451
  - > (right angle bracket)
    - in shell syntax, 84, 444
    - relational operator, 57, 61
  - >& (in shell syntax), 444
  - >= operator, 57, 61
  - >>, >>> operators, 60–61
  - = operator, 48, 55
  - = operator, 57, 61
    - for class objects, 281
    - for enumerated types, 271
    - for floating-point numbers, 96
    - for identity hash maps, 545
    - for strings, 65
    - wrappers and, 261
  - | operator, 59, 61
  - || operator, 57, 61
  - 0, 0b, 0B, 0x, 0X prefixes (in integers), 41
  - 0, printf flag, 81
  - > (in shell syntax), 444
  - 2D shapes, 595–603
  - ☉ (beer mug), 67
  - ⓪ (octonions), 46, 67
- A**
- A, a conversion characters, 80
  - Abstract classes, 265–271
    - extending, 267
    - interfaces and, 312, 321–322
    - no instantiating for, 267
    - object variables of, 267
  - abstract keyword, 265–271, 855
  - Abstract methods, 266
    - in functional interfaces, 342
  - AbstractAction class, 624, 628, 688, 691
  - AbstractButton class, 638, 689–692
    - is/setSelected methods, 692
    - setAction method, 689
    - setActionCommand method, 673
    - setDisplayMnemonicIndex method, 695–696
    - setHorizontalTextPosition method, 690–691
    - setMnemonic method, 696
  - abstractClasses/Employee.java, 270
  - abstractClasses/Person.java, 269
  - abstractClasses/PersonTest.java, 269
  - abstractClasses/Student.java, 270
  - AbstractCollection class, 324, 505, 518
  - AbstractQueue class, 501
  - Accelerators (in menus), 695–696
  - accept method
    - of FileFilter (Swing), 740, 745
    - of *java.io.FileFilter*, 740
  - acceptEither method (CompletableFuture), 834, 836
  - Access modifiers
    - checking, 287
    - final, 49, 155, 228–229, 319, 366–367, 787
    - private, 146, 193–194, 360
    - protected, 234–235, 308, 335
    - public, 36–37, 50, 143–146, 193–194, 313–314
    - public static final, 320
    - redundant, 320
    - static, 38, 156–163
    - static final, 49
    - void, 38
  - AccessibleObject class
    - canAccess, trySetAccessible methods, 299
    - setAccessible method, 295, 299
  - Accessor methods, 138–141, 152–153, 476
  - Accessory components, 742
  - accumulate method (LongAccumulator), 789
  - accumulateAndGet method (AtomicType), 788
  - Action interface, 623–629, 688
    - actionPerformed method, 624
    - add/removePropertyChangeListener methods, 624
    - get/putValue methods, 624, 629
    - is/setEnabled methods, 624, 629
    - predefined action table names in, 625
  - Action listeners, 623–629
  - ActionEvent class, 614, 637
    - getActionCommand, getModifiers methods, 638
  - ActionListener interface
    - actionPerformed method, 327, 338, 359, 365, 368, 615, 623–624, 638, 793
    - overriding, 688
    - implementing, 343, 615

- ActionMap class, 628
- Actions, 623–629
  - associating with keystrokes, 626
  - names of, 628
  - predefined, 625
- ActiveX, 5
- Adapter classes, 621–623
- add method
  - of ArrayList, 253–258
  - of BigDecimal, 109
  - of BigInteger, 108
  - of *BlockingQueue*, 798
  - of ButtonGroup, 673
  - of *Collection*, 501, 505–506, 508
  - of Container, 617, 620, 655
  - of GregorianCalendar, 138
  - of HashSet, 525
  - of JFrame, 595
  - of JMenu, 687–689
  - of JToolBar, 703–705
  - of List, 509, 521
  - of *ListIterator*, 509, 515–517, 522
  - of LongAdder, 788
  - of *Queue*, 532
  - of Set, 510
- addAll method
  - of ArrayList, 450
  - of *Collection*, 505–506
  - of Collections, 565
  - of List, 521
- addChoosableFileFilter method (*JFileChooser*), 745
- addExact method (*Math*), 53
- addFirst method
  - of *Deque*, 533
  - of *LinkedList*, 522
- addHandler method (*Logger*), 438
- addItem method (*JComboBox*), 677–679
- Addition operator, 51, 61
  - for different numeric types, 54
  - for objects and strings, 62–63, 245–246
- addLast method
  - of *Deque*, 533
  - of *LinkedList*, 522
- addLayoutComponent method (*LayoutManager*), 720
- addPropertyChangeListener method (*Action*), 624
- addSeparator method
  - of JMenu, 687, 689
  - of JToolBar, 703–705
- addShutdownHook method (*Runtime*), 181
- addSuppressed method (*Throwable*), 406, 409
- AdjustmentEvent class, 637
- AdjustmentListener* interface, 638
- Adobe Flash, 10
- Aggregation, 130–131
- Algorithms, 126
  - for binary search, 563–564
  - for shuffling, 561
  - for sorting, 560–563
  - QuickSort, 115, 561
  - simple, in the standard library, 564–566
  - writing, 568–569
- Algorithms + Data Structures = Programs* (Wirth), 126
- Algorithms in C++* (Sedgewick), 561
- Alice in Wonderland* (Carroll), 526, 528
- allOf method
  - of *CompletableFuture*, 834, 836
  - of *EnumSet*, 547
- allProcesses method (*ProcessHandle*), 850, 853
- Alt+F4, in Windows, 695
- Amazon, 17
- and, andNot methods (*BitSet*), 577
- Andreessen, Mark, 11
- Android platform, 16, 840
- Annotations, 462
- Anonymous arrays, 110
- Anonymous inner classes, 367–371
- anonymousInnerClass/AnonymousInnerClassTest.java, 370
- Antisymmetry rule, 319
- anyOf method (*CompletableFuture*), 834, 836
- append method
  - of *JTextArea*, 666
  - of *StringBuilder*, 73–74
- appendCodePoint method (*StringBuilder*), 74
- Applets, 9–10, 15
  - changing warning string in, 194
  - running in a browser, 9
- Application Programming Interfaces (APIs),
  - online documentation for, 68, 70–73
- Applications
  - closing by user, 586
  - compiling/launching from the command line, 22–24, 37
  - debugging, 24, 388–396
  - extensible, 227
  - for different Java releases, 201–202

- localizing, 132, 285, 426–427
- managing in JVM, 445
- responsive, 839
- terminating, 38, 149
- testing, 415–420
- `applyToEither` method (`CompletableFuture`), 834, 836
- Arguments. *See* Parameters
- `arguments` method (`ProcessHandle.Info`), 854
- Arithmetic operators, 51
  - accuracy of, 51
  - autoboxing with, 260
  - combining with assignment, 55
  - precedence of, 61
- Array class, 300–303
  - `get`, `getXxx`, `set`, `setXxx` methods, 303
  - `getLength` method, 301, 303
  - `newInstance` method, 300, 303
- Array lists, 110, 523
  - anonymous, 370
  - capacity of, 253
  - elements of:
    - accessing, 254–258
    - adding, 253–256
    - removing, 256
    - traversing, 256
  - generic, 251–259
  - raw vs. typed, 258–259
- Array variables, 109
- `ArrayBlockingQueue` class, 799, 803
- `ArrayDeque` class, 500, 532–533
  - as a concrete collection type, 511
- `ArrayIndexOutOfBoundsException`, 111, 391–393
- `ArrayList` class, 112, 251–259, 448–450, 512
  - `add` method, 253–258
  - `addAll` method, 450
  - as a concrete collection type, 511
  - declaring with `var`, 252
  - `ensureCapacity` method, 253–254
  - `get`, `set` methods, 255, 258
  - iterating over, 502
  - `remove` method, 256, 258
  - `removeIf` method, 344
  - `size`, `trimToSize` methods, 253–254
  - synchronized, 815
  - `toArray` method, 468
- `arrayList/ArrayListTest.java`, 257
- Arrays, 109–124
  - anonymous, 110
  - circular, 500
  - cloning, 336
  - converting to collections, 567–568
  - copying, 113–114
    - on write, 813
  - creating, 109
  - elements of:
    - computing in parallel, 814
    - numbering, 111
    - remembering types of, 225
    - removing from the middle, 512–513
    - traversing, 110, 112, 120
  - equality testing for, 240–241
  - generic methods for, 300–303
  - hash codes of, 243–244
  - in command-line parameters, 114–115
  - initializing, 110–111
  - length of, 111
    - equal to 0, 111
    - increasing, 113
  - multidimensional, 118–123, 240, 246
  - not of generic types, 349, 463–464, 466–468, 473
  - of integers, 246
  - of subclass/superclass references, 224
  - of wildcard types, 464
  - out-of-bounds access in, 390
  - parallel operations on, 813
  - printing, 120, 246
  - ragged, 121–124
  - size of, 253, 301
    - setting at runtime, 251
  - sorting, 115–118, 316, 813
- Arrays class
  - `asList` method, 550, 557
  - `binarySearch` method, 118, 381
  - `copyOf` method, 113, 117, 300
  - `copyOfRange` method, 117
  - `deepEquals` method, 240
  - `deepToString` method, 120, 246
  - `equals` method, 118, 240–241
  - `fill` method, 118
  - `hashCode` method, 243–244
  - `parallelXxx` methods, 813–814
  - `sort` method, 115–117, 313, 316, 318, 339, 343
  - `toString` method, 113, 117
- `arrays/CopyOfTest.java`, 302
- `ArrayStoreException`, 225, 463, 465, 473

- arrayType method (Class), 301, 303
- Ascender, ascent (in typesetting), 607
- ASCII standard, 45
- asIterator method (*Enumeration*), 571
- asList method (*Arrays*), 550, 557
- assert keyword, 415–420, 855
- Assertions, 415–420
  - checking parameters with, 417–419
  - defined, 415
  - documenting assumptions with, 419–420
  - enabling/disabling, 415–417
- Assignment operator, 48, 55
- Asynchronous computations, 830–846
- Asynchronous methods, 816
- AsyncTask class, 840
- atan, atan2 methods (*Math*), 52
- Atomic operations, 787–789
  - client-side locking for, 783
  - in concurrent hash maps, 806–810
  - performance of, 788
- AtomicType classes, 788
- @author comment (*javadoc*), 207, 209
- Autoboxing, 259–263
- AutoCloseable interface, 405
  - close method, 405–406
- await method (*Condition*), 755, 773–777
- AWT (Abstract Window Toolkit), 582
  - event hierarchy in, 636–639
  - preferred field sizes in, 659
- AWTEvent class, 636
- Azul, 17
- B**
- B, b conversion characters, 80
- \b escape sequence, 44
- Background color
  - default, 586
  - setting, 604–605, 617
- BadCastException, 484
- Base classes. *See* Superclasses
- Baseline (in typesetting), 607
- Basic multilingual planes, 45
- BasicButtonUI class, 652
- Batch files, 197
- Beans, 198
- beep method (*Toolkit*), 329
- BiConsumer interface, 353
- BiFunction interface, 343, 353
- BIG-5 standard, 45
- BigDecimal class, 106–109
  - add, compareTo, subtract, multiply, divide, mod methods, 109
- BigInteger class, 106–108
  - add, compareTo, subtract, multiply, divide, mod, sqrt methods, 108
  - valueOf method, 106–108
- BigIntegerTest/BigIntegerTest.java, 107
- Binary search, 563–564
- BinaryOperator interface, 353
- binarySearch method
  - of *Arrays*, 118, 381
  - of *Collections*, 563–564
- BiPredicate interface, 353
- Bit masks, 60
- Bit sets, 576–580
- Bitcode files, 37
- BitSet interface, 498, 576–580
  - methods of, 577
- Bitwise operators, 59–61
- Blank lines, printing, 39
- Blocking queues, 797–804
- BlockingDeque interface, methods of, 804
- BlockingQueue interface
  - add, element, peek, remove methods, 798
  - offer, poll, put, take methods, 798, 804
- blockingQueue/BlockingQueueTest.java, 800
- Blocks, 38, 85–86
  - nested, 85
  - synchronized, 782–784
- Boolean class
  - converting from boolean, 259
  - hashCode method, 244
- boolean operators, 57, 61
- boolean type, 46, 855
  - default initialization of, 171
  - formatting output for, 80
  - no casting to numeric types for, 55
- Border layout manager, 655–657
- BorderFactory class, methods of, 674–676
- BorderLayout class, 655–657
- Borders, 673–676
- Bounded collections, 500
- Bounding rectangle, 598–599
- Bounds checking, 114
- Box layout, 705
- break statement, 100–106, 855
  - labeled/unlabeled, 103–104
  - not allowed in switch expressions, 102

- Bridge methods, 460–461, 472
  - Buckets (of hash tables), 524
  - Bulk operations, 566–567
  - button/ButtonFrame.java, 618
  - ButtonGroup class, 670
    - add method, 673
    - getSelection method, 671, 673
  - ButtonModel* interface, 650
    - getActionCommand method, 671, 673
    - getSelectedObjects method, 671
    - properties of, 651
  - Buttons
    - appearance of, 648
    - associating actions with, 626
    - clicking, 617
    - creating, 616
    - event handling for, 616–620
    - model-view-controller analysis of, 650, 652
    - rearranging automatically, 653
  - ButtonUIListener class, 652
  - Byte class
    - converting from byte, 259
    - hashCode method, 244
    - toUnsignedInt method, 42
  - byte type, 40, 855
- C**
- C, c conversion characters, 80
  - C programming language
    - assert macro in, 416
    - function pointers in, 304
    - integer types in, 6, 41
  - C# programming language, 8
    - foreach loop in, 85
    - polymorphism in, 229
    - useful features of, 12
  - C++ programming language
    - , (comma) operator in, 61
    - :: operator in, 151, 218
    - >> operator in, 60
    - access privileges in, 155
    - algorithms in, 560
    - arrays in, 114, 123
    - bitset template in, 576
    - boolean values in, 46
    - classes in, 38, 358
    - code units and code points in, 62
    - copy constructors in, 135
    - dynamic binding in, 220
    - dynamic casts in, 231
    - exceptions in, 391, 394–395, 399
    - fields in:
      - instance, 173–174
      - static, 159
    - for loop in, 85, 94
    - function pointers in, 304
    - #include in, 189
    - inheritance in, 214, 223, 321
    - integer types in, 6, 41
    - iterators as parameters in, 571
    - methods in:
      - accessor, 138
      - default, 324
      - destructor, 180
      - static, 159
    - namespace directive in, 189
    - new operator in, 147
    - NULL pointer in, 135
    - object pointers in, 135
    - operator overloading in, 107
    - passing parameters in, 165, 168
    - performance of, compared to Java, 578
    - polymorphism in, 229
    - protected modifier in, 235
    - pure virtual functions (= 0) in, 267
    - references in, 135
    - Standard Template Library in, 498, 503
    - static member functions in, 38
    - strings in, 64–65
    - superclasses in, 219
    - syntax of, 3
    - templates in, 12, 452, 455, 457
    - this pointer in, 175
    - type parameters in, 454
    - using directive in, 189
    - variables in, 49
      - redefining in nested blocks, 86
    - vector template in, 254
    - virtual constructors in, 282
    - void\* pointer in, 236
  - Calendar class, 136
    - get/setTime methods, 228
  - Calendars
    - displaying, 139–140
    - vs. time measurement, 136
  - CalendarTest/CalendarTest.java, 140
  - Call by reference, 163

- Call by value, 163–170
- Callable* interface, 821
  - call method, 816–817
  - wrapper for, 817
- Callables, 816–818
- Callbacks, 326–329
- CamelCase, 36
- canAccess* method (*AccessibleObject*), 299
- cancel method (*Future*), 816–817, 819, 841
- CancellationException*, 841
- cardinality method (*BitSet*), 577
- Carriage return character, 44
- case statement, 58, 98–103, 855
- cast method (*Class*), 484
- Casts, 54–55, 229–232
  - bad, 390
  - checking before attempting, 230
- catch statement, 397–411, 855
- ceiling method (*NavigableSet*), 531
- ChangeListener* interface, *stateChanged* method, 680
- char type, 43–45, 855
- Character class
  - converting from char, 259
  - hashCode* method, 244
  - isJavaIdentifierXXX* methods, 47
- Characters
  - escape sequences for, 44
  - exotic, 67
  - formatting output for, 80
- charAt* method (*String*), 66, 68
- CharSequence* interface, 70, 322
- checkBox/CheckBoxFrame.java*, 668
- Checkboxes, 667–669, 691–692
- Checked exceptions, 281–283
  - applicability of, 413
  - declaring, 391–394
  - suppressing with generics, 469–471
- Checked views, 553
- checkedCollection* methods (*Collections*), 556
- checkFromIndexSize*, *checkFromToIndex*, *checkIndex* methods (*Objects*), 414
- Child classes. *See* Subclasses
- children method (*ProcessHandle*), 850, 853
- Choice components, 667–686
  - checkboxes, 667–669, 691–692
  - combo boxes, 676–680
  - radio buttons, 670–673, 691–692
  - sliders, 680–686
- ChronoLocalDate* interface, 479
- Church, Alonzo, 340
- circleLayout/CircleLayout.java*, 717
- circleLayout/CircleLayoutFrame.java*, 720
- Circular arrays, 500
- Clark, Jim, 11
- Class* class, 280–283
  - arrayType* method, 301, 303
  - cast* method, 484
  - componentType* method, 303
  - forName* method, 281–282
  - generic, 466, 483–486
  - getClass* method, 280
  - getComponentType* method, 301, 303
  - getConstructor* method, 282, 484
  - getConstructors* method, 287, 292
  - getDeclaredConstructor* method, 484
  - getDeclaredConstructors* method, 287, 292
  - getDeclaredMethods* method, 287, 292, 304
  - getEnumConstants* method, 484
  - getField*, *getDeclaredField* methods, 299
  - getFields*, *getDeclaredFields* methods, 287, 292, 296, 299
  - getGenericXXX* methods, 493
  - getImage* method, 285
  - getMethod* method, 304
  - getMethods* method, 287, 292
  - getName* method, 251, 280–281
  - getPackageName* method, 293
  - getRecordComponents* method, 293
  - getResource*, *getResourceAsStream* methods, 285–286
  - getSuperclass* method, 251, 484
  - getTypeParameters* method, 493
  - isArray* method, 303
  - isEnum*, *isInterface*, *isRecord* methods, 293
  - newInstance* method, 282, 484
- Class constants, 49
- Class diagrams, 130–131
- .class* file extension, 37
- Class files, 190, 195
  - compiling, 37
  - locating, 196–197
  - names of, 36, 143
- class keyword, 36, 855
- Class loaders, 379, 415–416
- Class path, 195–198
- Class wins rule, 326
- Class<T>* parameters, 484–485



- ClassCastException, 230, 300, 319, 467, 474, 553
- Classes, 36, 127–128, 214–235
  - abstract, 265–271, 312, 321–322
  - access privileges for, 154
  - adapter, 621–623
  - adding to packages, 190–193
  - capabilities of, 287–294
  - companion, 322, 324
  - constructors for, 146
  - defining, 141–156
    - at runtime, 379
  - designing, 129, 210–212
  - documentation comments for, 204–208
  - encapsulation of, 127–128, 151–154
  - extending, 128
  - final, 228–229, 335
  - generic, 251–252, 450–453, 474, 676
  - helper, 710
  - immutable, 155, 182, 309
  - implementing multiple interfaces, 320–321
  - importing, 187–189
  - inner, 357–375
  - instances of, 127, 132
  - legacy, 182
  - loading, 444
  - multiple source files for, 145
  - names of, 24, 36, 186, 211
    - full package, 187
  - number of basic types in, 210
  - objects of, at runtime, 294–300
  - package scope of, 193
  - parameters in, 150–151
  - predefined, 132–141
  - private methods in, 155
  - protected, 234–235
  - public, 187, 204
  - relationships between, 130–131
  - sealed, 273–279
  - sharing, among programs, 195
  - unit testing, 160
  - wrapper, 259–263
- ClassLoader class, 420
- CLASSPATH environment variable, 24, 197
- Cleaner class, 181
- clear method
  - of BitSet, 577
  - of *Collection*, 505, 507
  - clearAssertionStatus method (ClassLoader), 420
- Client-side locking, 782–783
- clone method
  - of array types, 336
  - of Object, 154, 330–338, 342
- clone/CloneTest.java, 336
- clone/Employee.java, 337
- Cloneable interface, 330–338
- CloneNotSupportedException, 334–335
- close method
  - of *AutoCloseable*, 405–406
  - of *Closeable*, 405
  - of Handler, 439
- Closures, 350
- Code errors, 389
- Code planes, 45
- Code points, code units, 45, 66
- codePointAt method (String), 68
- codePointCount method (String), 66, 69
- codePoints method (String), 67–68
- Collection* interface, 501, 508, 518
  - add method, 501, 505–506, 508
  - addAll method, 505–506
  - clear method, 505, 507
  - contains, containsAll methods, 505–506, 518
  - equals method, 505
  - generic, 504–507
  - implementing, 324
  - isEmpty method, 323, 505–506
  - iterator method, 501, 506
  - remove method, 505–506
  - removeAll method, 505, 507
  - removeIf method, 507, 566
  - retain method, 505
  - retainAll method, 507
  - size method, 505–506
  - stream method, 324
  - toArray method, 256, 505, 507, 567
- Collections, 497–580
  - algorithms for, 558–560
  - bounded, 500
  - bulk operations in, 566–567
  - concrete, 510–535
  - concurrent modifications of, 517
  - converting to arrays, 567–568
  - debugging, 518

- elements of:
    - inserting, 508
    - maximum, 558
    - removing, 504
    - traversing, 502–503
  - interfaces for, 498–507
  - legacy, 569–580
  - mutable, 549
  - ordered, 509, 514
  - performance of, 509, 525
  - searching in, 563–564
  - sorted, 527
  - thread-safe, 553–554, 797–815
  - type parameters for, 450
  - using for method parameters, 569
- Collections class, 561
- addAll method, 565
  - binarySearch method, 563–564
  - checkedCollection methods, 556
  - copy method, 565
  - disjoint method, 566
  - emptyCollection methods, 550, 557
  - enumeration method, 571
  - fill method, 565
  - frequency method, 566
  - indexOfSubList method, 565
  - lastIndexOfSubList method, 565
  - list method, 571
  - max, min methods, 565
  - nCopies method, 549, 556
  - replaceAll method, 565
  - reverse method, 565
  - rotate method, 566
  - shuffle method, 561–562
  - singleton, singletonXxx methods, 550, 557
  - sort method, 560–563
  - swap method, 565
  - synchronizedCollection methods, 553–554, 556, 815
  - unmodifiableCollection methods, 551–552, 556
- Collections framework. *See* Java collections framework (JCF)
- Color class, 603–605
- Colors
- background/foreground, 604
  - changing, 625
  - predefined/custom, 604
- Columns (of a text field), 659
- Combo boxes, 676–680
- adding items to, 677
  - comboBox/ComboBoxFrame.java, 678
  - command method (*ProcessHandle.Info*), 854
- Command line
- compiling/launching from, 22–24, 37
  - parameters in, 114–115
- commandLine method (*ProcessHandle.Info*), 854
- Comments, 39–40
- automatic documentation and, 40, 204–209
  - blocks of, 39
  - not nesting, 40
  - to the end of line, 39
- Companion classes, 322, 324
- Comparable interface, 312, 381, 455, 525, 560
- compareTo method, 313–317, 455, 478
- Comparator interface, 329–330, 338, 356–357, 560
- chaining comparators in, 356
  - comparing method, 356–357
  - lambda expressions and, 342
  - naturalOrder method, 357
  - nullFirst/Last methods, 357
  - reversed, reverseOrder methods, 357, 560, 563
  - thenComparing method, 356–357
- comparator method (SortedMap), 531, 539
- compare method (integer types), 318, 343
- compareAndSet method (*AtomicType*), 788
- compareTo method
- in subclasses, 319
  - of BigDecimal, 109
  - of BigInteger, 108
  - of Comparable, 313–317, 455, 478
  - of Enum, 273
  - of String, 68
- Compilation errors, 29
- Compiler
- autoboxing in, 261
  - bridge methods in, 460
  - command-line options of, 445
  - creating bytecode files in, 37
  - deducting method types in, 454
  - enforcing throws specifiers in, 398
  - error messages in, 29, 393
  - just-in-time, 6–7, 15, 151, 229, 578
  - launching, 23
  - optimizing method calls in, 7, 229

- overloading resolution in, 225
- shared strings in, 63, 65
- translating typed array lists in, 259
- type parameters in, 449
- warnings in, 101, 259
- whitespace in, 38
- `CompletableFuture` class, 832–839
  - `acceptEither` method, 834, 836
  - `allOf`, `anyOf` methods, 834, 836
  - `applyToEither` method, 834, 836
  - exceptionally, `exceptionallyCompose` methods, 834–835
  - `handle` method, 835
  - `orTimeout` method, 835
  - `runAfterXXX` methods, 834, 836
  - `thenAccept`, `thenAcceptBoth`, `thenCombine`, `thenRun` methods, 835
  - `thenApply`, `thenApplyAsync`, `thenCompose` methods, 833, 835
  - `whenComplete` method, 835
- `completableFutures/CompletableFutureDemo.java`, 836
- `CompletionStage` interface, 835
- `Component` class, 638
  - `getBackground/Foreground` methods, 605
  - `getFont` method, 661
  - `getPreferredSize` method, 593, 595
  - `getSize` method, 589
  - inheritance hierarchies of, 654
  - `isVisible` method, 589
  - `repaint` method, 592, 595
  - `setBackground/Foreground` methods, 604–605
  - `setBounds`, `setLocation` methods, 586, 588–589
  - `setCursor` method, 635
  - `setSize` method, 589
  - `setVisible` method, 586, 589
  - `validate` method, 661
- Components (in layout), 653
  - classes for, 584
  - displaying information in, 590–613
  - labeling, 661–662
  - visibility of, 586, 589
- Components (of records), 182
- `componentType` method (Class), 303
- `CompoundInterest/CompoundInterest.java`, 120
- Computations
  - asynchronous, 830–846
  - performance of, 51, 53
  - truncated, 51
- `compute`, `computeIfXXX` methods
  - of `ConcurrentHashMap`, 807
  - of `Map`, 540
- Concrete collections, 510–535
- Concrete methods, 266
- Concurrent hash maps
  - atomic updates in, 806–810
  - bulk operations on, 810–812
  - efficiency of, 805
  - size of, 805
  - vs. synchronization wrappers, 815
- Concurrent modification detection, 517
- Concurrent programming, 8, 747–850
  - records in, 184
  - synchronization in, 764–797
- Concurrent sets, 812–813
- `ConcurrentHashMap` class, 805–806
  - atomic updates in, 806–810
  - `compute`, `computeIfXXX` methods, 807–808
  - `forEach` method, 810–812
  - `get` method, 807
  - `keySet`, `newKeySet` methods, 812
  - `mappingCount` method, 805
  - `merge` method, 808
  - organizing buckets as trees in, 805
  - `put`, `putIfAbsent` methods, 807
  - `reduce`, `reduceXXX` methods, 810–812
  - `replace` method, 807
  - `search`, `searchXXX` methods, 810–812
- `concurrentHashMap/CHMDemo.java`, 808
- `ConcurrentLinkedQueue` class, 805–806
- `ConcurrentModificationException`, 517, 805, 815
- `ConcurrentSkipListMap/Set` classes, 805–806
- `Condition` interface, 778
  - `await` method, 755
  - `signal`, `signalAll` methods, 791
  - vs. synchronization methods, 780
- Condition objects, 772–777
- Condition variables, 772
- Conditional operator, 58
  - with pattern matching, 233
- Conditional statements, 86–89
- `config` method (Logger), 422, 437
- Configuration files, 639–645
- Confirmation dialogs, 724
- Console
  - printing output to, 36–39, 79
  - reading input from, 76–79

- Console class, 78
    - readLine/Password methods, 79
  - console method (System), 79
  - ConsoleHandler class, 427–431, 439
  - const keyword, 50, 856
  - Constants, 49–50
    - documentation comments for, 207
    - names of, 49
    - public, 50, 158
    - static, 157–158
  - Constructor class, 287
    - getDeclaringClass method, 293
    - getModifiers, getName methods, 287, 293
    - getXxxTypes methods, 293
    - newInstance method, 283, 484
  - Constructor expressions, 465
  - Constructor references, 348–349
  - Constructors, 146–148, 170–181
    - calling another constructor in, 175
    - canonical, compact, custom, 184
    - defined, 132
    - documentation comments for, 204
    - field initialization in, 171, 173
    - final, 287
    - initialization blocks in, 175–180
    - names of, 132, 147
    - no-argument, 172, 218, 377
    - overloading, 170–171
    - parameter names in, 174
    - private, 287
    - protected, 204
    - public, 204, 287
    - with super keyword, 218
  - ConstructorTest/ConstructorTest.java, 178
  - Consumer interface, 353
  - Consumer threads, 797
  - Container class, 653
    - add method, 617, 620, 655
    - setLayout method, 655
  - Containers, 653
  - contains method
    - of *Collection*, 505–506, 518
    - of *HashSet*, 525
  - containsAll method (*Collection*), 505–506, 518
  - containsKey/Value methods (*Map*), 538
  - Content pane, 591
  - continue statement, 105–106, 856
    - not allowed in switch expressions, 102
  - Control flow, 85–106
    - block scope, 85–86
    - breaking, 103–106
    - conditional statements, 86–89
    - loops, 89–94
      - determinate, 94–98
      - “for each”, 112–113
      - multiple selections, 98–103
  - Controllers, 648
  - Conversion characters, 80
  - Cooperative scheduling, 754
  - Coordinated Universal Time (UTC), 136
  - Copies, 548–558
    - unmodifiable, 550–552, 555
  - copy method (*Collections*), 565
  - copyArea method (*Graphics*), 613
  - copyOf method
    - of *Arrays*, 113, 117, 300
    - of *EnumSet*, 547
    - of *List*, *Map*, *Set*, 550, 555
    - of *Map.Entry*, 542
  - copyOfRange method (*Arrays*), 117
  - CopyOnWriteArrayList class, 813, 815
  - CopyOnWriteArraySet class, 813
  - Cornell, Gary, 1
  - Corruption of data, 764–768
  - cos method (*Math*), 52
  - Count of Monte Cristo, The* (Dumas), 528, 840–842
  - Covariant return types, 461
  - createTypeBorder methods (*BorderFactory*), 674–676
  - Ctrl+\, for thread dump, 791
  - Ctrl+C, for program termination, 765, 775
  - Ctrl+O, Ctrl+S accelerators, 695
  - current method
    - of *ProcessHandle*, 850, 853
    - of *ThreadLocalRandom*, 797
  - Current user, 640
  - currentThread method (*Thread*), 757–760
  - Cursor class, getPredefinedCursor method, 631
  - Cursor shapes, 631
  - Custom layout managers, 716–721
  - Customizations. *See* Preferences
- ## D
- d conversion character, 80
  - D suffix (double numbers), 42
  - Daemon threads, 761

- Data exchange, 730–737
- Data types, 40–46
  - boolean type, 46
  - casting between, 54–55
  - char type, 43–45
  - conversions between, 53–54, 229–232
  - floating-point, 42–43
  - integer, 40–42
- dataExchange/DataExchangeFrame.java, 733
- dataExchange/PasswordChooser.java, 734
- Date and time
  - formatting output for, 80
  - hash codes for, 244
  - no built-in types for, 132
- Date class, 136
  - getDay/Month/Year methods (deprecated), 137
  - toString method, 133
- DateInterval class, 460
- Deadlocks, 774, 789–793
- Debugging, 8, 441–446
  - collections, 518
  - debuggers for, 442
  - generic types, 553
  - GUI programs, 397
  - including class names in, 370
  - intermittent bugs, 65, 586
  - messages for, 396
  - reflection for, 295
  - trapping program errors in a file for, 444
  - when running applications in terminal window, 24
- Decrement operators, 56–57
- decrementExact method (Math), 53
- Deep copies, 333
- deepEquals method (Arrays), 240
- deepToString method (Arrays), 120, 246
- Default methods, 323–324
  - conflicts in, 324–326
- default statement, 100, 323–324, 856
  - sealed classes and, 275
- DefaultButtonModel class, 650, 652
- DefaultComboBoxModel class, 678
- Deferred execution, 352
- Delayed interface, 799
  - getDelay method, 799, 803
- DelayQueue class, 799, 803
- delete method (StringBuilder), 74
- Dependence, 130–131
- Deprecated methods, 137–138
- Deque interface, 532–533
  - methods of, 533
- Dequeues, 532–533
- Derived classes. *See* Subclasses
- deriveFont method (Font), 606, 611
- descendants method (ProcessHandle), 850, 853
- Descender, descent (in typesetting), 607
- descendingIterator method (NavigableSet), 532
- destroy, destroyForcibly methods (Process), 850, 853
- Determinate loops, 94–98
- Development environments
  - choosing, 22–27
  - in terminal window, 24
  - integrated, 27–30
- Device errors, 389
- dialog/AboutDialog.java, 729
- dialog/DialogFrame.java, 728
- Dialogs, 721–746
  - accepting/canceling, 731
  - centering, 329
  - closing, 621–622, 695, 728, 731
  - confirmation, 724
  - creating, 726–730
  - data exchange in, 730–737
  - default button in, 732
  - displaying, 728
  - modal, 721–726
  - modeless, 721, 727–728, 732
  - root pane of, 733
- Diamond syntax, 252
  - with anonymous subclasses, 449
- Digital signatures, 5
- Directories
  - starting, for a launched program, 83
  - working, for a process, 847
- directory method (ProcessBuilder), 847, 851
- disjoint method (Collections), 566
- divide method
  - of BigDecimal, 109
  - of BigInteger, 108
- Division operator, 51
- do/while loop, 91–92, 856
- Documentation comments, 40, 204–209
  - extracting, 209
  - for fields, 207
  - for methods, 206

- for packages, 208
- general, 207
- HTML markup in, 205
- hyperlinks in, 208
- inserting, 204–205
- links to other files in, 205
- overview, 209
- `doInBackground` method (`SwingWorker`), 841–842, 846
- Do-nothing methods, 622
- Double brace initialization, 370
- Double class
  - compare method, 318
  - converting from double, 259
  - `hashCode` method, 244
  - `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `NaN` constants, 43
- double type, 42, 856
  - arithmetic computations with, 51
  - converting to other types, 53–54
- `DoubleAccumulator`, `DoubleAdder` classes, 789
- Double-precision numbers, 42–43
- Doubly linked lists, 513
- `draw` method (`Graphics2D`), 596
- `draw/DrawTest.java`, 600
- `drawImage` method (`Graphics`), 612–613
- Drawing with mouse, 629–635
- `drawString` method (`Graphics/Graphics2D`), 612
- Drop-down lists, 676
- Dynamic binding, 220, 225–228
- Dynamic languages, 8

## E

### E

- as type variable, 451
- constant (`Math`), 53

E, e conversion characters, 42, 80

Echo character, 662–663

Eclipse, 22, 27–30, 441

- Adoptium, 17
- configuring projects in, 28
- editing source files in, 29
- error messages in, 29–30
- imports in, 188

Effectively final variables, 406

Eiffel programming language, 321

element method

- of `BlockingQueue`, 798
- of `Queue`, 532

`Ellipse2D` class, 596, 599

- `setFrameFromCenter` method, 599

`Ellipse2D.Double` class, 603

Ellipses, 596, 599

- bounding rectangles of, 598–599
- constructing, 599
- filling with color, 603

else statement, 86–87, 856

else if statement, 87, 89

Emoji characters, 67

`EmployeeTest/EmployeeTest.java`, 144

`emptyCollection` methods (`Collections`), 550, 557

`EmptyStackException`, 411–414

Encapsulation, 127–128

- benefits of, 151–154
- protected instance fields and, 308

`endsWith` method (`String`), 69

`ensureCapacity` method (`ArrayList`), 253–254

`entering` method (`Logger`), 437

Enterprise Edition (Java EE), 12

`entry` method (`Map`), 549, 555

`entrySet` method (`Map`), 540, 542

`Enum` class, 271–273

- `compareTo`, `ordinal` methods, 273
- `toString`, `valueOf` methods, 272–273

`enum` keyword, 50, 856

Enumerated types, 50

- equality testing for, 271
- in switch statement, 59

enumeration method (`Collections`), 571

`Enumeration` interface, 498, 570–571

- `asIterator` method, 571
- `hasMoreElements`, `nextElement` methods, 503, 570–571

Enumeration maps/sets, 545

Enumerations, 271–273

- always final, 229
- declared inside a class, 374
- implementing interfaces, 321
- legacy, 570–571

`EnumMap` class, 545, 547

- as a concrete collection type, 511

`enums/EnumTest.java`, 272

`EnumSet` class, 545

- `allOf` method, 547
- as a concrete collection type, 511
- `copyOf`, `noneOf`, `of`, `range` methods, 547

`environment` method (`ProcessBuilder`), 852

- environment variables, modifying, 848
- EOFException, 394–395
- Epoch, 136
- equals method, 326
  - hashCode method and, 242–243
  - implementing, 239
  - inheritance and, 238–241
  - of Arrays, 118, 240–241
  - of *Collection*, 505
  - of Object, 236–241, 251, 552
  - of proxy classes, 384
  - of records, 182, 237
  - of Set, 510
  - of String, 64, 69
  - redefining, 242–243
  - wrappers and, 261
- equals/Employee.java, 248
- equals/EqualsTest.java, 247
- equals/Manager.java, 249
- equalsIgnoreCase method (String), 65, 69
- Error class, 390
- Errors
  - checking, in mutator methods, 153
  - code, 389
  - compilation, 29
  - device, 389
  - internal, 390, 393, 418
  - messages for, 400
  - NoClassDefFoundError, 24
  - physical limitations, 389
  - ThreadDeath, 756, 763, 793
  - user input, 389
- Escape sequences, 44
- Event delegation model, 614
- Event dispatch thread, 585, 794
- Event handling, 614, 636–639
  - semantic vs. low-level events, 637
- Event listeners, 614–615
  - with lambda expressions, 620
- Event objects, 614
- Event procedures, 614
- Event sources, 614–615
- EventObject class, 614
  - getActionCommand, getSource methods, 637
- Exception class, 390, 409
- Exception handlers, 283, 389
- Exception specification, 392
- exceptionally, exceptionallyCompose methods (CompletableFuture), 834–835
- Exceptions, 389–391
  - ArrayIndexOutOfBoundsException, 111, 391–393
  - ArrayStoreException, 225, 463, 465, 473
  - BadCastException, 484
  - CancellationException, 841
  - catching, 149, 283–284, 335, 394, 397–411
  - changing type of, 400
  - checked, 281–283, 391–394, 411, 413
  - ClassCastException, 230, 300, 319, 467, 474, 553
  - CloneNotSupportedException, 334–335
  - ConcurrentModificationException, 517, 805, 815
  - creating classes for, 395–396
  - documentation comments for, 206
  - EmptyStackException, 411–414
  - EOFException, 394–395
  - FileNotFoundException, 392–394
  - finally clause in, 402–405
  - generics in, 469–471
  - hierarchy of, 389, 413
  - IllegalAccessException, 295, 299
  - IllegalStateException, 504, 507, 522, 532–533, 798
  - InaccessibleObjectException, 296
  - InterruptedException, 749, 757–760, 816
  - InvocationTargetException, 282
  - IOException, 84, 392–394, 398, 405
  - logging, 424, 433
  - micromanaging, 412
  - NoSuchElementException, 502, 507, 522, 532–533
  - NullPointerException, 149–150, 163, 261, 348, 391, 414
  - NumberFormatException, 413
  - out-of-bounds, 414
  - propagating, 398, 414
  - rethrowing and chaining, 400, 443
  - RuntimeException, 390–391, 413
  - ServletException, 400
  - sqlclching, 413
  - stack trace for, 407–411
  - “throw early, catch late”, 415
  - throwing, 283–284, 394–395
  - TimeoutException, 816
  - tips for using, 411–415
  - type variables in, 469
  - uncaught, 444, 756, 761–763
  - unchecked, 283, 391–393, 413

unexpected, 424, 433  
UnsupportedOperationException, 541, 551, 554, 556  
variables for, implicitly final, 400  
vs. simple tests, 411  
wrapping, 401  
.exe file extension, 201  
exec method (Runtime), 847  
Executable class, 305  
Executable JAR files, 200–201  
Executable path, 19  
execute method (SwingWorker), 841, 846  
Execution flow, tracing, 423  
ExecutorCompletionService class, 822  
poll, submit, take methods, 826  
Executors, 815–829  
groups of tasks, controlling, 821–826  
scheduled, 820  
Executors class, newXxx methods, 818–820  
executors/ExecutorDemo.java, 823  
ExecutorService interface, 818–820  
invokeAny/All methods, 821, 826  
shutdown method, 819–820  
shutdownNow method, 819, 821  
submit method, 819–820  
Exit codes, 38  
exit method (System), 38  
exiting method (Logger), 423, 437  
exitValue method (Process), 850, 853  
exp method (Math), 53  
Explicit parameters, 150–151  
exploratory programming, 7  
exports keyword, 856  
exportXxx methods (Preferences), 641, 645  
Expressions, 56  
extends keyword, 214–235, 455, 856  
External padding, 709

## F

F suffix (float numbers), 42  
F, f conversion characters, 80  
\f escape sequence, 44  
Factorial functions, 407  
Factory methods, 159  
Fair locks, 772  
Fallthrough behavior, 101  
false value, 856  
fdlibm (Freely Distributable Math Library), 53

Field class, 287  
get method, 294, 300  
getDeclaringClass method, 293  
getModifiers, getName methods, 287, 293  
getType method, 287  
set method, 300

## Fields

adding, in subclasses, 218  
default initialization of, 171  
documentation comments for, 204, 207  
final, 158, 228  
instance, 127, 146–152, 155, 173, 210  
private, 210, 216–217  
protected, 204, 234, 308  
public, 204, 207  
public static final, 320  
static, 156–157, 177, 189, 468  
volatile, 785–787  
with the null value, 149

File dialogs, 737–746  
adding accessory components to, 742

FileFilter class (Swing), methods of, 740, 745

FileFilter interface (java.io package), 740

FileHandler class, 427–431, 439  
configuration parameters of, 429

FileNameExtensionFilter interface, 745

FileNotFoundException, 392–394

## Files

filters for, 740–742  
locating, 83  
names of, 24, 82  
opening/saving in GUI, 737–746  
reading, 82  
all words from, 405  
in a separate thread, 840  
writing, 83

FileView class, methods of, 741, 746

fill method  
of Arrays, 118  
of Collections, 565  
of Graphics2D, 603–605

Filter interface, 431  
isLoggable method, 431, 441

final access modifier, 49, 228–229, 856  
checking, 287  
for fields in interfaces, 320  
for instance fields, 155  
for methods in superclass, 319



- for shared fields, 787
  - inner classes and, 366–367
- finalize method, 180–181
- finally clause, 402–405, 856
  - return statements in, 404
  - unlock operation in, 769
  - without catch, 403
- Financial calculations, 43
- findFirst method (ServiceLoader), 378
- fine, finer, finest methods (Logger), 422, 437
- first method (SortedSet), 531
- First Person, Inc., 11
- firstKey method (SortedMap), 539
- FirstSample/FirstSample.java, 40
- Flags, for formatted output, 81
- Flash, 583
- Float class
  - converting from float, 259
  - hashCode method, 244
  - POSITIVE\_INFINITY, NEGATIVE\_INFINITY, NaN constants, 43
- float type, 42, 856
  - converting to other numeric types, 53–54
- Floating-point numbers, 42–43
  - arithmetic computations with, 51
  - converting from/to integers, 229
  - equality of, 96
  - formatting output for, 80
  - rounding, 43, 55
- floor method (NavigableSet), 531
- floorMod method (Math), 52
- Flow layout manager, 653
- FlowLayout class, 655
- flush method (Handler), 439
- FocusEvent class, 637
  - isTemporary method, 638
- FocusListener interface, methods of, 638
- Font class, 606–611
  - deriveFont method, 606, 611
  - getFamily, getFontName, getName methods, 611
  - getLineMetrics method, 608, 611
  - getStringBounds method, 607–608, 611
- font/FontTest.java, 609
- FontMetrics class, getFontRenderContext method, 612
- Fonts, 605–612
  - checking availability of, 605
  - names of, 605–606
  - size of, 606
  - styles of, 606
  - typesetting properties of, 607
- “for each” loop, 110–113
  - for array lists, 256
  - for collections, 502, 815
  - for multidimensional arrays, 120
- for loop, 94–98, 856
  - comma-separated expressions in, 61
  - defining variables inside, 96
  - for collections, 502
- forEach method
  - of ConcurrentHashMap, 810–812
  - of Map, 538
  - of StackWalker, 410
- forEachRemaining method (Iterator), 501, 507
- Foreground color, 604
- Fork-join framework, 827
- forkJoin/ForkJoinTest.java, 828
- Form feed character, 44
- Format specifiers (printf), 80–82
- format, formatted, formatTo methods (String), 82
- formatMessage method (Formatter), 441
- Formattable interface, 81
- Formatter class, methods of, 431, 441
- forName method (Class), 281–282
- Frame class, 583
  - getIconImage method, 590
  - getTitle method, 590
  - isResizable method, 589
  - setIconImage method, 586, 590
  - setResizable method, 586, 589
  - setTitle method, 586, 590
- Frames
  - closing by user, 586
  - creating, 583
  - displaying information in, 590–613
  - positioning, 586–590
  - properties of, 586–590
- frequency method (Collections), 566
- Function interface, 353, 356
- Functional interfaces, 342–344
  - abstract methods in, 342
  - annotating, 355
  - conversion to, 343
  - generic, 343
  - using supertype bounds in, 479
- @FunctionalInterface annotation, 355

Functions. *See* Methods

*Future* interface, 821

- cancel, get methods, 816–817, 819, 841
- isCancelled, isDone methods, 816–817, 819

Futures, 816–818

- combining, 834, 836
- completable, 832–839

*FutureTask* class, 816–818

## G

G, g conversion characters, 80

Garbage collection, 64, 135

- hash maps and, 542–543

GB18030 standard, 45

General Public License (GPL), 15

Generic programming, 447–495

- arrays and, 349, 466–468
- classes in, 251–252, 450–453, 676
  - extending/implementing other generic classes, 474
  - no throwing or catching instances of, 469

collection interfaces in, 567

converting to raw types, 473

debugging, 553

expressions in, 458

in JVM, 457, 485–489

inheritance rules for, 472–474

legacy code and, 461

methods in, 453–454, 459–461, 504–507

reflection and, 483–495

required skill levels for, 449

static fields or methods and, 468

type erasure in, 457–463, 466

- clashes after, 471–472

type matching in, 484–485

vs. inheritance, 448–450

wildcard types in, 475–483

*GenericArrayType* interface, 485–486

getGenericComponentType method, 495

genericReflection/GenericReflectionTest.java, 486

genericReflection/TypeLiterals.java, 490

get method

- of Array, 303
- of ArrayList, 255, 258
- of BitSet, 577
- of ConcurrentHashMap, 807
- of Field, 294, 300

of *Future*, 816–817, 819, 841

of *LinkedList*, 518

of *List*, 509, 521

of *LongAccumulator*, 789

of *Map*, 508, 536–537

of Paths, 322

of Preferences, 640, 645

of *ServiceLoader.Provider*, 377–378

of *ThreadLocal*, 797

of *Vector*, 784

getAccessor method (*RecordComponent*), 294

getActionCommand method

- of *ActionEvent*, 638
- of *ButtonModel*, 671, 673
- of *EventObject*, 637

getActionMap method (*JComponent*), 629

getActualTypeArguments method  
(*ParameterizedType*), 494

getAdjustable, getAdjustmentType methods  
(*AdjustmentEvent*), 638

getAncestorOfClass method (*SwingUtilities*), 732, 737

getAndType methods (*AtomicType*), 788

getAscent method (*LineMetrics*), 611

getAvailableFontFamilyNames method  
(*GraphicsEnvironment*), 605

getBackground method (*Component*), 605

getBoolean method

- of Array, 303
- of Preferences, 640, 645

getBounds method (*TypeVariable*), 494

getBytes method (*Array*), 303

getBytesArray method (*Preferences*), 640, 645

getCause method (*Throwable*), 409

getCenterX/Y methods (*RectangularShape*), 598, 602

getChar method (*Array*), 303

getClass method

- always returning raw types, 463
- of *Class*, 280
- of *Object*, 250

getClassName method

- of *StackFrame*, 410
- of *StackTraceElement*, 411

getClickCount method (*MouseEvent*), 630, 635, 638

getColumns method (*JTextField*), 661

getComponentPopupMenu method (*JComponent*), 694

getComponentType method (*Class*), 301, 303

- getConstructor method (Class), 282, 484
- getConstructors method (Class), 287, 292
- getDay method (Date, deprecated), 137
- getDayXxx methods (LocalDate), 137, 141
- getDeclaredConstructor method (Class), 484
- getDeclaredConstructors method (Class), 287, 292
- getDeclaredField method (Class), 299
- getDeclaredFields method (Class), 287, 292, 296, 299
- getDeclaredMethods method (Class), 287, 292, 304
- getDeclaringClass method
  - of java.lang.reflect, 293
  - of StackFrame, 410
- getDefaultToolkit method (Toolkit), 329, 588, 590
- getDefaultUncaughtExceptionHandler method (Thread), 762
- getDelay method (*Delayed*), 799, 803
- getDescent method (LineMetrics), 611
- getDescription method
  - of FileFilter, 740, 745
  - of FileView, 741, 746
- getDouble method
  - of Array, 303
  - of Preferences, 640, 645
- getEnumConstants method (Class), 484
- getErrorStream method (Process), 847–848, 852
- getExceptionTypes method (Constructor), 293
- getFamily method (Font), 611
- getField method (Class), 299
- getFields method (Class), 287, 292, 299
- getFileName method
  - of StackFrame, 410
  - of StackTraceElement, 411
- getFilter method
  - of Handler, 439
  - of Logger, 438
- getFirst method (*Deque*), 533
- getFloat method
  - of Array, 303
  - of Preferences, 640, 645
- getFont method (Component), 661
- getFontMetrics method (JComponent), 608, 612
- getFontName method (Font), 611
- getFontRenderContext method
  - of FontMetrics, 612
  - of Graphics2D, 607, 612
- getForeground method (Component), 605
- getFormatter method (Handler), 439
- getGenericComponentType method (GenericArrayType), 495
- getGenericXxx methods (Class), 493
- getGenericXxx methods (Method), 494
- getGlobal method (Logger), 421, 443
- getHandlers method (Logger), 438
- getHead method (Formatter), 432, 441
- getHeight method
  - of LineMetrics, 612
  - of RectangularShape, 598, 602
- getIcon method
  - of FileView, 741, 746
  - of JLabel, 662
- getIconImage method (Frame), 590
- getImage method
  - of Class, 285
  - of ImageIcon, 590, 612
- getInheritsPopupMenu method (JComponent), 694
- getInputMap method (JComponent), 627, 629
- getInputStream method (Process), 847, 852
- getInstance method (StackWalker), 407, 410
- getInstant method (LogRecord), 440
- getInt method
  - of Array, 303
  - of Preferences, 640, 645
- getItem, getItemSelectable methods (ItemEvent), 638
- getItemAt method (JComboBox), 677
- getKey method (*Map.Entry*), 542
- getKeyStroke method (KeyStroke), 626, 629
- getKeyXxx methods (KeyEvent), 638
- getLargestPoolSize method (ThreadPoolExecutor), 820
- getLast method (*Deque*), 533
- getLeading method (LineMetrics), 612
- getLength method (Array), 301, 303
- getLevel method
  - of Handler, 439
  - of Logger, 438
  - of LogRecord, 440
- getLineMetrics method (Font), 608, 611
- getLineNumber method
  - of StackFrame, 410
  - of StackTraceElement, 411
- getLogger method (Logger), 422, 437
- getLoggerName method (LogRecord), 440
- getLogManager method (LogManager), 441

- getLong method
  - of Array, 303
  - of Preferences, 640, 645
- getLongThreadID method (LogRecord), 440
- getLowerBounds method (WildcardType), 494
- getMaxX/Y methods (RectangularShape), 602
- getMessage method
  - of LogRecord, 440
  - of Throwable, 396
- getMethod method (Class), 304
- getMethodName method
  - of StackFrame, 410
  - of StackTraceElement, 411
- getMethods method (Class), 287, 292
- getMillis method (LogRecord), 440
- getMinX/Y methods (RectangularShape), 602
- getModifiers method
  - of ActionEvent, 638
  - of java.lang.reflect, 287, 293
- getMonth method (Date, deprecated), 137
- getMonthXXX methods (LocalDate), 137, 141
- getName method
  - of Class, 251, 280–281
  - of FileView, 741, 746
  - of Font, 611
  - of java.lang.reflect, 287, 293
  - of RecordComponent, 294
  - of TypeVariable, 494
- getNewState, getOldState methods (WindowEvent), 639
- getOppositeWindow method (WindowEvent), 639
- getOrDefault method (Map), 537
- getOutputStream method (Process), 847, 852
- getOwnerType method (ParameterizedType), 494
- getPackageName method (Class), 293
- getPaint method (Graphics2D), 604
- getParameters method (LogRecord), 440
- getParameterTypes method (Method), 293
- getParent method (Logger), 438
- getPassword method (JPasswordField), 663
- getPoint method (MouseEvent), 635, 638
- getPredefinedCursor method (Cursor), 631
- getPreferredSize method (Component), 593, 595
- getProperties method (System), 573, 575
- getProperty method
  - of Properties, 573–574
  - of System, 83, 575
- getProxyClass method (Proxy), 384–385
- getRawType method (ParameterizedType), 494
- getRecordComponents method (Class), 293
- getResource, getResourceAsStream methods (Class), 285–286
- getResourceBundle, getResourceBundleName methods (LogRecord), 440
- getReturnType method (Method), 293
- getRootPane method (JComponent), 733, 737
- getScreenSize method (Toolkit), 588, 590
- getScrollAmount method (MouseWheelEvent), 638
- getSelectedFile/Files methods (JFileChooser), 740, 744
- getSelectedItem method (JComboBox), 677–680
- getSelectedObjects method (ItemSelectable), 671
- getSelection method (ButtonGroup), 671, 673
- getSequenceNumber method (LogRecord), 440
- getShort method (Array), 303
- getSize method (Component), 589
- getSource method (EventObject), 637
- getSourceXXXName methods (LogRecord), 440
- getStackTrace method (Throwable), 407, 409
- getState method
  - of SwingWorker, 846
  - of Thread, 575
- getStateChange method (ItemEvent), 638
- getStringBounds method (Font), 607–608, 611
- getSuperclass method (Class), 251, 484
- getSuppressed method (Throwable), 406, 409
- getTail method (Formatter), 432, 441
- Getter/setter pairs. *See* Properties
- getText method
  - of JLabel, 662
  - of JTextComponent, 660
- getThrown method (LogRecord), 440
- getTime method (Calendar), 228
- getTitle method (Frame), 590
- getType method
  - of java.lang.reflect, 287
  - of RecordComponent, 294
- getTypeDescription method (FileView), 741, 746
- getTypeParameters method
  - of Class, 493
  - of Method, 494
- getUncaughtExceptionHandler method (Thread), 762
- getUpperBounds method (WildcardType), 494
- getUseParentHandlers method (Logger), 438

- getValue method
    - of *Action*, 624, 629
    - of *AdjustmentEvent*, 638
    - of *Map.Entry*, 542
  - getWheelRotation method (*MouseEvent*), 638
  - getWidth method
    - of *Rectangle2D*, 598
    - of *RectangularShape*, 598, 602
  - getWindow method (*WindowEvent*), 639
  - getX/Y methods
    - of *MouseEvent*, 630, 635, 638
    - of *RectangularShape*, 602
  - getYear method
    - of *Date* (deprecated), 137
    - of *LocalDate*, 137, 141
  - GMT (Greenwich Mean Time), 136
  - Goetz, Brian, 748, 786
  - Gosling, James, 10–11
  - goto statement, 85, 103, 856
  - Graphical User Interface (GUI), 581–645
    - components of, 647–746
      - choice components, 667–686
      - dialog boxes, 721–746
      - menus, 686–705
      - text input, 658–667
      - toolbars, 701–704
      - tooltips, 704–705
    - deadlocks in, 793
    - debugging, 397
    - events in, 614
    - keyboard focus in, 626
    - layout of, 652–658, 705–721
    - long-running tasks in, 839–846
  - Graphics* class, 595, 612–613
    - copyArea method, 613
    - drawImage method, 612–613
  - Graphics editor applications, 629–635
  - Graphics2D* class, 595–603
    - draw method, 596
    - drawString method, 612
    - fill method, 603–605
    - getFontRenderContext method, 607, 612
    - getPaint method, 604
    - setPaint method, 603–604
  - GraphicsEnvironment* class, 605
  - Green project, 10–11
  - GregorianCalendar* class, 138
    - add method, 138
    - constructors for, 136, 170–171
  - Grid bag layout, 705–716
  - Grid layout, 657–658
  - gridbag/FontFrame.java*, 711
  - gridbag/GBC.java*, 713
  - GridBagConstraints* class, 707
    - anchor, fill parameters, 709, 716
    - gridx/y, gridwidth/height parameters, 708, 715
    - helper class for, 710
    - insets parameter, 709, 716
    - ipadx/y parameters, 716
    - weightx/y parameters, 708, 715
  - GridLayout* class, 654, 657–658
  - Group layout, 706
  - GUI. *See* Graphical User Interface
- ## H
- H, h conversion characters, 80
  - handle method (*CompletableFuture*), 835
  - Handler class, 430
    - close method, 439
    - flush method, 439
    - get/setFilter methods, 439
    - get/setFormatter methods, 439
    - get/setLevel methods, 439
    - publish method, 431, 439
    - setFormatter method, 432
  - Handlers, 427–431
  - Hansen, Per Brinch, 784–785
  - “Has-a” relationship, 130–131
  - hash method (*Objects*), 243–244
  - Hash codes, 241–244, 523
    - default, 242
    - formatting output for, 80
  - Hash collisions, 244, 524
  - Hash maps, 535
    - concurrent, 805–806
    - identity, 545–548
    - linked, 543–545
    - setting, 535
    - vs. tree maps, 535
    - weak, 542–543
  - Hash sets, 523–527
    - adding elements to, 528
    - linked, 543–545
  - Hash tables, 523–524
    - legacy, 570
    - load factor of, 525
    - rehashing, 525

- hashCode method, 241–244
    - equals method and, 242–243
    - null-safe, 243
    - of Arrays, 243–244
    - of Boolean, Byte, Character, Double, Float, Integer, Long, Short, 244
    - of LocalDate, 244
    - of Object, 244, 527
    - of Objects, 243–244
    - of proxy classes, 384
    - of records, 182, 243
    - of Set, 510
    - of String, 523
  - HashMap class, 535, 538
    - as a concrete collection type, 511
  - HashSet class, 525–527
    - add, contains methods, 525
    - as a concrete collection type, 511
    - iterating over, 502
  - Hashtable interface, 498, 569–570, 814–815
    - as a concrete collection type, 511
    - synchronized methods, 570
  - hasMoreElements method (*Enumeration*), 503, 570–571
  - hasNext method
    - of *Iterator*, 501–503, 507
    - of *Scanner*, 78
  - hasNextXxx methods (*Scanner*), 79
  - hasPrevious method (*ListIterator*), 515, 522
  - headMap method
    - of *NavigableMap*, 558
    - of *SortedMap*, 552, 558
  - headSet method
    - of *NavigableSet*, 553, 558
    - of *SortedSet*, 552, 557
  - Heap, 533
  - Height (in typesetting), 607
  - Helper classes, 710
  - Helper methods, 155, 323, 481
  - Hexadecimal numbers
    - formatting output for, 80
    - prefix for, 41
  - HexFormat class, 80
  - higher method (*NavigableSet*), 531
  - Hoare, Tony, 784
  - Hold count, 770
  - HotJava browser, 11
  - Hotspot just-in-time compiler, 18, 578
  - HTML (HyperText Markup Language), 12, 14
    - in javadoc comments, 205
    - in labels, 662
    - tables in, 706
  - HTML editors, 649
- ## I
- Icons
    - in menu items, 690–691
    - in sliders, 682
  - Identifiers, 855
  - Identity hash maps, 545–548
  - identityHashCode method (*System*), 545, 548
  - IdentityHashMap class, 545–548
    - as a concrete collection type, 511
  - IEEE 754 specification, 43, 53
  - if statement, 86–89, 856
  - IFC (Internet Foundation Classes), 582
  - IllegalAccessException, 295, 299
  - IllegalStateException, 504, 507, 522, 532–533, 798
  - ImageIcon class, 589
    - getImage method, 590, 612
  - Images, displaying, 612–613
  - ImageViewer/ImageViewer.java, 26
  - Immutable classes, 155, 309
  - Implementations, 498–501
  - implements keyword, 314, 856
  - Implicit parameters, 150–151
    - none, in static methods, 158
    - state of, 442
  - import statement, 187–189, 856
  - importPreferences method (*Preferences*), 641, 645
  - InaccessibleObjectException, 296
  - Inconsistent state, 793
  - increment method (*LongAdder*), 788
  - Increment operators, 56–57
  - Incremental linking, 7
  - incrementAndGet method (*AtomicType*), 788
  - incrementExact method (*Math*), 53
  - Indentation, in text blocks, 76
  - Index (in arrays), 109
  - @index comment (javadoc), 208
  - indexOf method
    - of *List*, 521
    - of *String*, 69
  - indexOfSubList method (*Collections*), 565

- Inferred types, 341
- info method
  - of `Logger`, 421–422, 437
  - of `ProcessHandle`, 853
- Information hiding. *See* Encapsulation
- Inheritance, 130–131, 213–310
  - design hints for, 308–310
  - equality testing and, 238–241
  - hierarchies of, 222–223
  - multiple, 223, 321
  - preventing, 228–229
  - private fields and, 216
  - vs. type parameters, 448, 472–474
- `inheritance/Employee.java`, 221
- `inheritance/Manager.java`, 222
- `inheritance/ManagerTest.java`, 220
- `inheritIO` method (`ProcessBuilder`), 851
- `initCause` method (`Throwable`), 409
- Initialization blocks, 175–180
  - static, 177
- Inlining, 7, 229
- Inner classes, 357–375
  - accessing object state with, 358–362
  - anonymous, 367–371
  - applicability of, 363–365
  - defined, 357
  - local, 365
  - private, 360
  - static, 358, 372–375
  - syntax of, 362–363
  - translated into regular classes, 363
  - vs. lambda expressions, 343
- `innerClass/InnerClassTest.java`, 361
- Input dialogs, 724
- Input maps, 627–628
- Input, reading, 76–79
- `InputTest/InputTest.java`, 77
- `insert` method
  - of `JMenu`, 689
  - of `StringBuilder`, 74
- `insertItemAt` method (`JComboBox`), 677, 679
- `insertSeparator` method (`JMenu`), 689
- Instance fields, 127
  - final, 155
  - initializing, 175–180, 210
    - explicit, 173
  - names of, 182
  - not present in interfaces, 313, 320
  - private, 146, 210
  - protected, 308
  - public, 146
  - shadowing, 148, 174
  - values of, 152
  - volatile, 785–787
  - vs. local variables, 148, 151, 171
- `instanceof` operator, 61, 230, 232, 240, 319, 856
  - pattern matching for, 232–234
- Instances, 127
  - creating on the fly, 282
- `int` type, 40, 856
  - converting to other numeric types, 53–54
  - fixed size for, 6
  - platform-independent, 41
- `Integer` class
  - `compare` method, 318, 343
  - converting from `int`, 259
  - `hashCode` method, 244
  - `intValue` method, 263
  - `parseInt` method, 262–263
  - `toString` method, 263
  - `valueOf` method, 263
- `Integer` types, 40–42
  - arithmetic computations with, 51
  - arrays of, 246
  - computations of, 53
  - converting from/to floating-point, 229
  - formatting output for, 80
  - no unsigned types in Java, 41
- Integrated Development Environment (IDE), 27–30
- IntelliJ IDEA, 27
- interface keyword, 312, 857
- Interface types, 500
- Interface variables, 319
- Interfaces, 312–338
  - abstract classes and, 321–322
  - binary- vs. source-compatible, 324
  - callbacks and, 326–329
  - constants in, 320
  - declared inside a class, 374
  - documentation comments for, 204
  - evolution of, 324
  - extending, 319
  - for custom algorithms, 568–569
  - functional, 342–344
  - implementing, 314, 319–323

- methods in:
  - clashes between, 324–326
  - do-nothing, 622
  - nonabstract, 342
  - private, 323
  - static, 322
- no instance fields in, 313, 320
- properties of, 319–321
- public, 204
- sealed, 321
- tagging, 334, 458, 509
- vs. implementations, 498–501
- interfaces/*Employee.java*, 317
- interfaces/*EmployeeSortTest.java*, 316
- Intermittent bugs, 65, 586
- Internal errors, 390, 393, 418
- Internal padding, 709
- Internationalization. *See* Localization
- Internet Explorer browser, 10
- Interpreted languages, 15
- Interpreter, 7
- interrupt method (Thread), 757–760
- interrupted method (Thread), 759–760
- InterruptedException, 749, 757–760, 816
- Intrinsic locks, 778, 785–786, 793
- Introduction to Algorithms* (Cormen et al.), 528
- intValue method (Integer), 263
- Invocation handlers, 379
- InvocationHandler* interface, 379, 384–385
- InvocationTargetException*, 282
- invoke method
  - of *InvocationHandler*, 379, 384–385
  - of Method, 304–307
- invokeAny/All methods (*ExecutorService*), 821, 826
- invokeDefault method (*InvocationHandler*), 385
- IOException, 84, 392–394, 398, 405
- “Is-a” relationship, 130–131, 223, 308
- isAbstract method (Modifier), 294
- isActionKey method (KeyEvent), 638
- isAlive method (Process), 850, 853
- isArray method (Class), 303
- isBlank method (String), 69
- isCancelled, isDone methods (*Future*), 816–819
- isDefaultButton method (JButton), 737
- isEditable method
  - of JComboBox, 679
  - of JTextComponent, 659
- isEmpty method
  - of Collection, 323, 505–506
  - of String, 69
- isEnabled method (Action), 624, 629
- isEnum method (Class), 293
- isFinal method (Modifier), 287, 294
- isInterface method
  - of Class, 293
  - of Modifier, 294
- isInterrupted method (Thread), 757–760
- isJavaIdentifierXXX methods (Character), 47
- isLocationByPlatform method (Window), 589
- isLoggable method (Filter), 431, 441
- isNaN method (Double), 43
- isNative method (Modifier), 294
- isNativeMethod method
  - of StackFrame, 410
  - of StackTraceElement, 411
- ISO 8859-1 standard, 45, 572
- isPopupTrigger method (JPopupMenu, MouseEvent), 693
- isPrivate, isProtected, isPublic methods (Modifier), 287, 294
- isProxyClass method (Proxy), 384–385
- isRecord method (Class), 293
- isResizable method (Frame), 589
- isSelected method
  - of AbstractButton, 692
  - of JCheckBox, 668–669
- isStatic, isStrict, isSynchronized methods (Modifier), 294
- isTemporary method (FocusEvent), 638
- isTraversable method (FileView), 741, 746
- isVisible method (Component), 589
- isVolatile method (Modifier), 294
- ItemEvent class, 637
  - getXXX methods, 638
- ItemListener* interface, *itemStateChanged* method, 638
- ItemSelectable* interface, *getSelectedObjects* method, 671
- Iterable* interface, 112
- Iterator* interface, 501–504
  - “for each” loop, 502
  - forEachRemaining* method, 501, 507
  - generic, 504
  - hasNext* method, 501–503, 507
  - next* method, 501–504, 507
  - remove* method, 501, 503–504, 507



- iterator method
  - of *Collection*, 501, 506
  - of *ServiceLoader*, 378
- Iterators, 501–504
  - being between elements, 503
  - weakly consistent, 805
- IzPack utility, 201
- J**
- J#, J++ programming languages, 8
- Jar Bundler utility, 201
- JAR files, 195, 198–204
  - creating, 198–199
  - executable, 200–201
  - in *jre/lib/ext* directory, 198
  - manifest of, 199–200
  - multi-release, 201–202
  - resources and, 284–286
- jar program, 198–199
  - command-line options of, 199–200, 203–204
- Java 2D library, 595–603
  - floating-point coordinates in, 596
- Java bug parade, 37
- Java collections framework (JCF), 497–580
  - algorithms in, 558–560
  - converting to/from arrays in, 567–568
  - copies and views in, 548–558
  - interfaces in, 508–510
    - vs. implementations, 498–501
  - legacy classes in, 569–580
  - operations in:
    - bulk, 566–567
    - optional, 554
  - vs. traditional collections libraries, 503
- Java Concurrency in Practice* (Goetz), 748
- Java Development Kit (JDK), 6, 17–33
  - documentation in, 70–73, 628
  - downloading, 18
  - fonts shipped with, 606
  - installation of, 17–22, 198
  - setting up, 18–20
- .java file extension, 36
- Java Language Specification, 37
- Java look-and-feel, 626
- Java Memory Model and Thread Specification, 786
- java program, 23
  - command-line options of, 202, 416–417
- Java programming language
  - architecture-neutral object file format of, 6
  - as a programming platform, 1–2
  - available under GPL, 15
  - backward compatibility of, 201, 233, 356, 448
  - basic syntax of, 36–39, 142
  - case-sensitiveness of, 24, 36, 47–48, 570
  - design of, 2–8
  - documentation for, 21
  - dynamic, 8
  - history of, 10–13
  - interpreter in, 7
  - libraries in, 4, 12, 14
    - installing, 20–22
  - misconceptions about, 13–16
  - networking capabilities of, 4
  - no multiple inheritance in, 321
  - no operator overloading in, 107
  - no unsigned types in, 41
  - reliability of, 4
  - security of, 5, 15, 365
  - simplicity of, 3–4, 339
  - strongly typed, 40, 315
  - versions of, 11–13, 582, 705
  - vs. C++, 3, 578
- Java Runtime Environment (JRE), 18
- Java virtual machine (JVM), 6
  - generics in, 457, 485–489
  - launching, 23
  - managing applications in, 445
  - method tables in, 226
  - optimizing execution in, 423
  - thread priority levels in, 763
  - truncating computations in, 51
  - watching class loading in, 444
- Java Virtual Machine Specification, 37
  - java.awt.BorderLayout* API, 657
  - java.awt.Color* API, 604
  - java.awt.Component* API, 589, 595, 605, 635, 661
  - java.awt.Container* API, 620, 655
  - java.awt.event.MouseEvent* API, 635, 693
  - java.awt.event.WindowListener* API, 622–623
  - java.awt.event.WindowStateListener* API, 623
  - java.awt.FlowLayout* API, 655
  - java.awt.Font* API, 611
  - java.awt.font.LineMetrics* API, 611–612

- java.awt.FontMetrics API, 612
- java.awt.Frame API, 589–590
- java.awt.geom.RectangularShape API, 602
- java.awt.geom.Xxx2D.Double APIs, 603
- java.awt.Graphics API, 613
- java.awt.Graphics2D API, 604–605, 612
- java.awt.GridBagConstraints API, 715–716
- java.awt.GridLayout API, 658
- java.awt.LayoutManager API, 720–721
- java.awt.Toolkit API, 329, 590
- java.awt.Window API, 589, 595
- java.io.Console API, 79
- java.io.PrintWriter API, 84
- java.lang.Boolean API, 244
- java.lang.Byte API, 244
- java.lang.Character API, 244
- java.lang.Class API, 251, 282, 286, 292–293, 299, 303, 484, 493
- java.lang.ClassLoader API, 420
- java.lang.Comparable API, 317
- java.lang.Double API, 244, 318
- java.lang.Enum API, 273
- java.lang.Exception API, 409
- java.lang.Float API, 244
- java.lang.Integer API, 244, 263, 318
- java.lang.Long API, 244
- java.lang.Object API, 128, 244, 250–251, 527, 782
- java.lang.Objects API, 244
- java.lang.Process API, 852–853
- java.lang.ProcessBuilder API, 851–852
- java.lang.ProcessHandle API, 853
- java.lang.ProcessHandle.Info API, 854
- java.lang.reflect package, 287, 300
- java.lang.reflect.AccessibleObject API, 299
- java.lang.reflect.Array API, 303
- java.lang.reflect.Constructor API, 283, 293, 484
- java.lang.reflect.Field API, 293, 300
- java.lang.reflect.GenericArrayType API, 495
- java.lang.reflect.InvocationHandler API, 385
- java.lang.reflect.Method API, 293, 307, 494
- java.lang.reflect.Modifier API, 294
- java.lang.reflect.ParameterizedType API, 494
- java.lang.reflect.Proxy API, 385
- java.lang.reflect.RecordComponent API, 294
- java.lang.reflect.TypeVariable API, 494
- java.lang.reflect.WildcardType API, 494
- java.lang.Runnable API, 753
- java.lang.RuntimeException API, 409
- java.lang.Short API, 244
- java.lang.StackTraceElement API, 411
- java.lang.StackWalker API, 410
- java.lang.StackWalker.StackFrame API, 410
- java.lang.String API, 68–70
- java.lang.StringBuilder API, 73–74
- java.lang.System API, 79, 548, 575
- java.lang.Thread API, 753, 755, 757, 760–763
- java.lang.Thread.UncaughtExceptionHandler API, 762
- java.lang.ThreadGroup API, 763
- java.lang.ThreadLocal API, 797
- java.lang.Throwable API, 283, 396, 409
- java.logging module, 421
- java.math.BigDecimal API, 109
- java.math.BigInteger API, 108
- java.nio.file.Path API, 84
- java.text.NumberFormat API, 263
- java.time.LocalDate API, 141
- java.util.ArrayDeque API, 533
- java.util.ArrayList API, 254, 258
- java.util.Arrays API, 117–118, 241, 244, 318, 557
- java.util.BitSet API, 577
- java.util.Collection API, 506–507, 566
- java.util.Collections API, 556–557, 561–562, 564–566, 571, 815
- java.util.Comparator API, 563
- java.util.concurrent package, 769
  - efficient collections in, 805–806
- java.util.concurrent.ArrayBlockingQueue API, 803
- java.util.concurrent.atomic package, 787
- java.util.concurrent.BlockingDeque API, 804
- java.util.concurrent.BlockingQueue API, 804
- java.util.concurrent.Callable API, 817
- java.util.concurrent.Delayed API, 803
- java.util.concurrent.DelayQueue API, 803
- java.util.concurrent.ExecutorCompletionService API, 826
- java.util.concurrent.Executors API, 820
- java.util.concurrent.ExecutorService API, 820, 826
- java.util.concurrent.Future API, 817
- java.util.concurrent.FutureTask API, 818
- java.util.concurrent.LinkedBlockingDeque API, 803
- java.util.concurrent.LinkedBlockingQueue API, 803
- java.util.concurrent.locks.Condition API, 777
- java.util.concurrent.locks.Lock API, 771, 777

- `java.util.concurrent.locks.ReentrantLock` API, 772
- `java.util.concurrent.PriorityBlockingQueue` API, 803
- `java.util.concurrent.ScheduledExecutorService` API, 821
- `java.util.concurrent.ThreadLocalRandom` API, 797
- `java.util.concurrent.ThreadPoolExecutor` API, 820
- `java.util.concurrent.TransferQueue` API, 804
- `java.util.Deque` API, 533
- `java.util.Enumeration` API, 571
- `java.util.EnumMap` API, 547
- `java.util.EnumSet` API, 547
- `java.util.function` API, 343
- `java.util.HashMap` API, 538
- `java.util.HashSet` API, 527
- `java.util.IdentityHashMap` API, 547
- `java.util.Iterator` API, 507
- `java.util.LinkedHashMap` API, 546
- `java.util.LinkedHashSet` API, 546
- `java.util.LinkedList` API, 522
- `java.util.List` API, 521, 555, 557, 562, 566
- `java.util.ListIterator` API, 522
- `java.util.logging.ConsoleHandler` API, 439
- `java.util.logging.FileHandler` API, 439
- `java.util.logging.Filter` API, 441
- `java.util.logging.Formatter` API, 441
- `java.util.logging.Handler` API, 439
- `java.util.logging.Logger` API, 437–438
- `java.util.logging.LogManager` API, 441
- `java.util.logging.LogRecord` API, 440
- `java.util.Map` API, 537–538, 540, 542, 555
- `java.util.Map.Entry` API, 542
- `java.util.NavigableMap` API, 558
- `java.util.NavigableSet` API, 531–532, 558
- `java.util.Objects` API, 163, 241
- `java.util.prefs.Preferences` API, 644–645
- `java.util.PriorityQueue` API, 535
- `java.util.Properties` API, 574
- `java.util.Queue` API, 532
- `java.util.random` package, 178
- `java.util.Random` API, 180
- `java.util.random.RandomGenerator` API, 180
- `java.util.Scanner` API, 78–79, 84
- `java.util.ServiceLoader` API, 378
- `java.util.ServiceLoader.Provider` API, 378
- `java.util.Set` API, 555
- `java.util.SortedMap` API, 539, 558
- `java.util.SortedSet` API, 531, 557
- `java.util.Stack` API, 576
- `java.util.Timer` API, 328
- `java.util.TreeMap` API, 538
- `java.util.TreeSet` API, 531
- `java.util.WeakHashMap` API, 546
- JavaBeans, 743
- javac program, 23
  - command-line options of, 203–204
  - current directory in, 196
- javadoc program, 204–209
  - command-line options of, 209
  - comments in, 204–208
    - extracting, 209
    - overview, 209
    - redeclaring Object methods for, 342
  - HTML markup in, 205
  - links in, 205, 208
  - online documentation of, 209
- JavaFX platform, 583, 840
- `javafx.css.CssParser` class, 201–202
- javap program, 202, 363
- JavaScript programming language, 16
- `javax.swing` package, 585
- `javax.swing.AbstractAction` API, 691
- `javax.swing.AbstractButton` API, 673, 689–692, 696
- `javax.swing.Action` API, 629
- `javax.swing.border.LineBorder` API, 676
- `javax.swing.border.SoftBevelBorder` API, 676
- `javax.swing.BorderFactory` API, 675–676
- `javax.swing.ButtonGroup` API, 673
- `javax.swing.ButtonModel` API, 673
- `javax.swing.event.MenuListener` API, 698
- `javax.swing.filechooser.FileFilter` API, 745
- `javax.swing.filechooser.FileNameExtensionFilter` API, 745
- `javax.swing.filechooser.FileView` API, 746
- `javax.swing.ImageIcon` API, 590
- `javax.swing.JButton` API, 620, 737
- `javax.swing.JCheckBox` API, 669
- `javax.swing.JCheckBoxMenuItem` API, 692
- `javax.swing.JComboBox` API, 679–680
- `javax.swing.JComponent` API, 595, 612, 629, 661, 676, 694, 705, 737
- `javax.swing.JDialog` API, 730
- `javax.swing.JFileChooser` API, 744–745
- `javax.swing.JFrame` API, 595, 690
- `javax.swing.JLabel` API, 662
- `javax.swing.JMenu` API, 689
- `javax.swing.JMenuItem` API, 689–690, 696, 698

- javax.swing.JOptionPane API, 329, 725–726
- javax.swing.JPasswordField API, 663
- javax.swing.JPopupMenu API, 693
- javax.swing.JRadioButton API, 673
- javax.swing.JRadioButtonMenuItem API, 692
- javax.swing.JRootPane API, 737
- javax.swing.JScrollPane API, 667
- javax.swing.JSlider API, 686
- javax.swing.JTextArea API, 666
- javax.swing.JTextField API, 660–661
- javax.swing.JToolBar API, 704–705
- javax.swing.KeyStroke API, 629
- javax.swing.SwingUtilities API, 737
- javax.swing.SwingWorker API, 846
- javax.swing.text.JTextComponent API, 659
- javax.swing.Timer API, 328–329
- JButton class, 616, 620, 626, 652
  - isDefaultButton method, 737
- JCheckBox class, 667–669
  - is/setSelected methods, 668–669
- JCheckBoxMenuItem class, 691–692
- JComboBox class, 638, 676–680
  - addItem method, 677–679
  - getItemAt method, 677
  - getSelectedItem method, 677–680
  - insertItemAt method, 677, 679
  - isEditable method, 679
  - removeAllItems, removeItemAt methods, 678, 680
  - removeItem method, 678–679
  - setEditable method, 676, 679
  - setModel method, 678
- JComponent class, 591
  - action maps, 628
  - get/setComponentPopupMenu methods, 693–694
  - get/setInheritsPopupMenu methods, 693–694
  - getActionMap method, 629
  - getFontMetrics method, 608, 612
  - getInputMap method, 627, 629
  - getRootPane method, 733, 737
  - input maps, 627–628
  - paintComponent method, 591–593, 595, 608, 613
  - revalidate method, 660–661
  - setBorder method, 674, 676
  - setFont method, 661
  - setToolTipText method, 705
- jconsole program, 445, 790–791
  - logging control with, 425
- JDialog class, 726–730
  - setDefaultCloseOperation method, 728
  - setVisible method, 728, 730–731
- JDK. *See* Java Development Kit
- JEditorPane class, 665
- JFileChooser class, 737–746
  - addChoosableFileFilter method, 745
  - getSelectedFile/Files methods, 740, 744
  - resetChoosableFilters method, 741, 745
  - setAcceptAllFileFilterUsed method, 741, 745
  - setAccessory method, 745
  - setCurrentDirectory method, 739, 744
  - setFileFilter method, 741, 745
  - setFileSelectionMode method, 739, 744
  - setFileView method, 741–742, 745
  - setMultiSelectionEnabled method, 739, 744
  - setSelectedFile/Files methods, 739, 744
  - showDialog method, 732, 738–739, 744
  - showXXXDialog methods, 738–739, 744
- JFrame class, 583–587, 654
  - add method, 595
  - internal structure of, 591–592
  - setJMenuBar method, 687, 690
- JLabel class, 661–662, 742
  - methods of, 662
- JMenu class
  - add method, 687–689
  - addSeparator method, 687, 689
  - insert, insertSeparator methods, 689
  - remove method, 689
- JMenuBar class, 687–690
- JMenuItem class, 689–690
  - setAccelerator method, 695–696
  - setEnabled method, 697–698
  - setIcon method, 690
- Jmol applet, 9
- join method (Thread), 70, 755–757
- JOptionPane class, 721–726
  - message types, 722
  - showConfirmDialog method, 722–725
  - showInputDialog method, 722–723, 726
  - showInternalConfirmDialog method, 725
  - showInternalInputDialog method, 726
  - showInternalMessageDialog method, 725
  - showInternalOptionDialog method, 726
  - showMessageDialog method, 329, 722–725
  - showOptionDialog method, 722–724, 726
- JPanel class, 653, 657

- JPasswordField class, getPassword, setEchoChar methods, 663
  - JPopupMenu class, 692–694
    - isPopupTrigger, show methods, 693
  - JRadioButton class, 670–673
  - JRadioButtonMenuItem class, 692
  - JRootPane class, setDefaultButton method, 733, 737
  - JScrollbar class, 638
  - JScrollPane class, 667
  - JShell program, 7, 30–33
  - JSlider class, 680–686
    - setInverted method, 682
    - setLabelTable method, 461, 682, 686
    - setPaintLabels, setPaintTicks, setSnapToTicks methods, 681, 686
    - setPaintTrack method, 682, 686
    - setXxxTickSpacing methods, 686
  - JSON (JavaScript Object Notation), 274
  - JTextArea class, 663–664
    - append method, 666
    - setColumns, setRows methods, 663, 666
    - setLineWrap method, 664, 666
    - setTabSize method, 666
    - setWrapStyleWord method, 666
  - JTextComponent class
    - getText method, 660
    - is/setEditable methods, 659
    - setText method, 659–660
  - JTextField class, 638, 658–662
    - getColumns method, 661
    - setColumns method, 659, 661
  - JToolBar class, 703
    - add, addSeparator methods, 703–705
  - JUnit framework, 442
  - Just-in-time compiler, 6–7, 15, 151, 229, 578
  - JVM. *See* Java virtual machine
- K**
- K type variable, 451
  - Key/value pairs. *See* Properties
  - Keyboard
    - associating with actions, 626
    - focus of, 626
    - mnemonics for, 694–696
  - KeyEvent class, 637
    - getKeyXxx, isActionKey methods, 638
  - KeyListener interface, keyXxx methods, 638
  - keys method (Preferences), 641, 645
  - keySet method
    - of ConcurrentHashMap, 812
    - of Map, 540, 542
  - KeyStroke class, getKeyStroke method, 626, 629
  - Keywords, 855–859
    - hyphenated, 277
    - not used, 50
    - redundant, 320
    - reserved, 36, 47
    - restricted, 855
  - Knuth, Donald, 103
  - KOI-8 standard, 45
- L**
- L, l suffixes (for long integers), 41
  - Labels
    - for components, 661–662
    - for slider ticks, 682
  - Lambda expressions, 338–357
    - accessing variables in, 349–352
    - atomic updates with, 788
    - capturing values by, 350
    - for event listeners, 620
    - functional interfaces and, 342
    - method references and, 345
    - not for variables of type Object, 343
    - parameter types of, 340
    - processing, 352–356
    - result type of, 341
    - scope of, 351
    - syntax of, 339–342
    - this keyword in, 351
    - vs. inner classes, 343
    - vs. method references, 348
  - lambda/LambdaTest.java, 341
  - Langer, Angelika, 495
  - last method (SortedSet), 531
  - lastIndexOf method
    - of List, 521
    - of String, 69
  - lastIndexOfSubList method (Collections), 565
  - lastKey method (SortedMap), 539
  - Launch4J utility, 201
  - Layout management, 652–658
    - border, 655–657
    - box, 705
    - custom, 716–721
    - flow, 653

- grid, 657–658
- grid bag, 705–716
- group, 706
- sophisticated, 705–721
- spring, 705
- layoutContainer method (*LayoutManager*), 721
- LayoutManager* interface
  - designing custom, 716–721
  - methods of, 720–721
- LayoutManager2* interface, 717
- Leading (in typesetting), 607
- Legacy classes, 182
  - generics and, 461–462
- Legacy collections, 569–580
  - bit sets, 576–580
  - enumerations, 570–571
  - hash tables, 570
  - property maps, 572–575
  - stacks, 575
- length method
  - of arrays, 111
  - of *BitSet*, 577
  - of *String*, 65–66, 69
  - of *StringBuilder*, 73
- Line feed character
  - escape sequence for, 44
  - in output, 39, 74
  - in text blocks, 74
- Line2D* class, 596, 600
- Line2D.Double* class, 603
- LineBorder* class, 674, 676
- LineMetrics* class, 608
  - getXxx* methods, 611
- Lines, 596
  - constructing, 600
- @Link comment (javadoc), 208
- Linked hash maps/sets, 543–545
- Linked lists, 512–522
  - concurrent modifications of, 517
  - doubly linked, 513
  - printing, 519
  - random access in, 518, 559
  - removing elements from, 514
- LinkedBlockingDeque* class, 803
- LinkedBlockingQueue* class, 799, 803
- LinkedHashMap* class, 543–546
  - access vs. insertion order in, 544
  - as a concrete collection type, 511
  - removeEldestEntry* method, 544, 546
- LinkedHashSet* class, 543–546
  - as a concrete collection type, 511
- LinkedList* class, 500, 514, 518, 532
  - addFirst/Last* methods, 522
  - as a concrete collection type, 511
  - get* method, 518
  - getFirst/Last* methods, 522
  - listIterator* method, 515
  - next/previousIndex* methods, 519
  - removeAll* method, 519
  - removeFirst/Last* methods, 522
  - linkedList/LinkedListTest.java*, 520
- Linux operating system
  - IDEs for, 27
  - JDK in, 17, 19
  - no thread priorities in Oracle JVM for, 763
  - paths in, 19, 195–197
  - pop-up menus in, 693
  - troubleshooting Java programs in, 24
- List* interface, 509
  - add* method, 509, 521
  - addAll* method, 521
  - copyOf* method, 550, 555
  - get* method, 509, 521
  - indexOf*, *lastIndexOf* methods, 521
  - listIterator* method, 521
  - of method, 548–550, 555, 567
  - remove* method, 509, 521
  - replaceAll* method, 566
  - set* method, 509, 521
  - sort* method, 562
  - subList* method, 552, 557
- list* method (*Collections*), 571
- Listener interfaces, 614
- Listener objects, 614
- Listeners. *See* Action listeners, Event listeners, Window listeners
- ListIterator* interface, 518
  - add* method, 509, 515–517, 522
  - hasPrevious* method, 515, 522
  - next/previousIndex* methods, 522
  - previous* method, 515, 522
  - remove* method, 517
  - set* method, 517, 522
- listIterator* method
  - of *LinkedList*, 515
  - of *List*, 521

- Lists, 509
  - modifiable/resizable, 561
  - unmodifiable, 555
  - with given elements, 548–550
- load method
  - of Properties, 572, 574
  - of ServiceLoader, 378
- Local inner classes, 365
  - accessing variables from outer methods in, 366–367
- Local variables
  - annotating, 462
  - vs. instance fields, 148, 151, 171
- LocalDate class, 136–138
  - getXxx methods, 137, 141
  - hashCode method, 244
  - minusDays method, 141
  - now, of methods, 136, 141
  - plusDays method, 137, 141
  - processing arrays of, 479
- Locales, 82, 426
- Localization, 132, 284–285, 426–427
- Lock interface, 778
  - await method, 773–777
  - lock method, 771
  - newCondition method, 773, 777
  - signal method, 775–777
  - signalAll method, 774–777
  - tryLock method, 755
  - unlock method, 769, 771
  - vs. synchronization methods, 780
- Locks, 769–772
  - client-side, 783
  - condition objects for, 772–777
  - deadlocks, 774, 789, 793
  - fair, 772
  - hold count for, 770
  - in synchronized blocks, 782–784
  - inconsistent state and, 793
  - intrinsic, 778, 785–786, 793
  - not with try-with-resources statement, 769
  - not wrapper objects for, 261
  - reentrant, 770
- log, log10 methods (Math), 53
- Logger class
  - add/removeHandler methods, 438
  - entering, exiting methods, 423, 437
  - get/setFilter methods, 431, 438
  - get/setParent methods, 438
  - get/setUseParentHandlers methods, 438
  - getGlobal method, 421, 443
  - getHandlers method, 438
  - getLevel method, 438
  - getLogger method, 422, 437
  - info method, 421
  - log method, 422, 424, 437
  - logp method, 423, 438
  - logrb method, 438
  - setLevel method, 421, 438
  - severe, warning, info, config, fine, finer, finest methods, 422, 437
  - throwing method, 424, 437
- Loggers
  - configuring, 424–426
  - default, 421, 423
  - hierarchical names of, 422
  - writing your own, 421–424
- Logging, 420–441
  - advanced, 421–424
  - basic, 421
  - file pattern variables for, 429
  - file rotation for, 428
  - filters for, 431
  - formatters for, 431
  - handlers for, 427–431
  - including class names in, 370
  - levels of, 422–425
  - localizing, 426–427
  - messages for, 246
  - recipe for, 432–441
  - resource bundles and, 426–427
- Logging proxy, 443
- logging/LoggingImageViewer.java, 433
- logging.properties file, 424–426
- Logical conditions, 46
- Logical “and”, “or”, 57
- LogManager class, 426
  - getLogManager method, 441
  - read/updateConfiguration methods, 425, 441
- LogRecord class, methods of, 440
- Long class
  - converting from long, 259
  - hashCode method, 244
- Long Term Support (LTS), 18
- long type, 40, 857
  - platform-independent, 41
- LongAccumulator class, methods of, 789

LongAdder class, 788, 808  
 add, increment, sum methods, 788

Look-and-feel  
 appearance of buttons in, 648  
 pluggable, 741

## Loops

break statements in, 103–106  
 continue statements in, 105–106  
 determinate (for), 94–98  
 “for each”, 112–113  
 while, 89–94

LotteryArray/LotteryArray.java, 124

LotteryDrawing/LotteryDrawing.java, 116

LotteryOdds/LotteryOdds.java, 97

lower method (NavigableSet), 531

Low-level events, 637

## M

Mac OS X operating system

executing JARs in, 201

IDEs for, 27

JDK in, 17, 19

main method, 160–163

body of, 38

declared public, 37

declared static void, 38

not defined, 141, 177

separate for each class, 442

String[] args parameter of, 114–115

tagged with throws, 84

make program (UNIX), 145

MANIFEST.MF (manifest file), 199–200

editing, 200

newline characters in, 200

Map interface, 508

compute, computeIfXxx methods, 540

containsKey/Value methods, 538

copyOf method, 550, 555

entry method, 549, 555

entrySet method, 540, 542

forEach method, 538

get method, 508, 536–537

getOrDefault method, 537

keySet method, 540, 542

merge method, 540

of method, 548–549, 555

ofEntries method, 549, 555

put method, 508, 536–537

putAll method, 538

putIfAbsent method, 540

remove method, 536

replaceAll method, 540

values method, 540, 542

map/MapTest.java, 536

Map.Entry interface, 540

copyOf, getKey, get/setValue methods, 542

mappingCount method (ConcurrentHashMap), 805

Maps, 535–548

adding/retrieving objects to/from, 535

concurrent, 805–806

garbage collecting, 542

hash vs. tree, 535

implementations for, 535

keys for, 536

enumerating, 541

subranges of, 552

unmodifiable, 555

with given key/value pairs, 548–550

Marker interfaces, 334

Math class, 32, 51–53

E, PI static constants, 53, 157–158

floorMod method, 52

log, log10 methods, 53

pow method, 52, 158

round method, 55

sqrt method, 52, 305–306

trigonometric functions, 52

xxxExact methods, 53

Matisse, 706

max method (Collections), 565

Maximum value, computing, 452

menu/MenuFrame.java, 698

MenuListener interface, 697

menuXxx methods, 697–698

Menus, 686–705

accelerators for, 695–696

checkboxes in, 691–692

icons in, 690–691

keyboard mnemonics for, 694–696

menu bar in, 687

menu items in, 687–692

enabling/disabling, 696–701

pop-up, 692–694

radio buttons in, 691–692

submenus in, 687

merge method

of ConcurrentHashMap, 808

of Map, 540



- Merge sort algorithm, 561
- META-INF directory, 199
- META-INF/versions directory, 201
- Method class, 287
  - getDeclaringClass method, 293
  - getGenericXXX methods, 494
  - getModifiers, getName methods, 287, 293
  - getReturnType method, 293
  - getTypeParameters method, 494
  - getXXTypes methods, 293
  - invoke method, 304–307
  - toString method, 287
- Method parameters. *See* Parameters
- Method pointers, 304–305
- Method references, 344–348
  - this, super parameters in, 348
  - vs. lambda expressions, 348
- Method tables, 226
- Methods, 127
  - abstract, 266
    - in functional interfaces, 342
  - accessor, 138–141, 152–153, 476
  - adding, in subclasses, 218
  - applying to objects, 133
  - asynchronous, 816
  - body of, 38
  - bridge, 460–461, 472
  - calling by reference vs. by value, 163–170
  - casting, 229–232
  - chaining calls of, 355
  - concrete, 266
  - conflicts in, 324–326
  - consistent, 238
  - default, 323–324
  - deprecated, 137–138
  - destructor, 180–181
  - documentation comments for, 204–208
  - do-nothing, 622
  - dynamic binding for, 220, 225–228
  - error checking in, 153
  - exception specification in, 392
  - factory, 159
  - final, 226–229, 287, 319
  - generic, 453–454, 459–461, 504–507
  - helper, 155, 481
  - inlining, 7, 229
  - invoking, 39, 304–307
  - mutator, 138–141, 153, 476
  - names of, 182, 211
  - overloading, 171
  - overriding, 216–218, 241, 309
    - exceptions and, 394
    - return type and, 459
  - package scope of, 193
  - passing objects to, 133
  - private, 155, 226, 287, 323
  - protected, 204, 234, 308, 335
  - public, 204, 287, 314
  - reflexive, 238
  - return type of, 171, 226
  - signature of, 171, 226
  - static, 158–159, 189, 226, 468, 780
    - adding to interfaces, 322
  - symmetric, 238
  - synchronized, 779
  - tracing, 380
  - transitive, 238
  - utility, 323
  - varargs, 263–265, 464–465
  - visibility of, in subclasses, 228
- methods/MethodTableTest.java, 306
- Micro Edition (Java ME), 12, 18
- Microsoft
  - .NET platform, 6
  - ActiveX, 5
  - C#, 8, 12, 229
  - Internet Explorer, 10
  - J#, J++, 8
  - JDK in, 17
  - Visual Basic, 3, 132, 614
  - Visual Studio, 22
- min method (Collections), 565
- Minimum value, computing, 452
- minimumLayoutSize method (*LayoutManager*), 721
- minusDays method (*LocalDate*), 141
- mod method
  - of *BigDecimal*, 109
  - of *BigInteger*, 108
- Modality, 721, 727
- Model-view-controller, 648–652
  - classes in, 648
  - multiple views in, 650
- Modifier class
  - isXXX methods, 287, 294
  - toString method, 294
- module keyword, 857
- Module path, 198

- Modules, 12, 194
    - unnamed, 296
  - Modulus operator, 51
  - Monitor concept, 784–785
  - Mosaic browser, 11
  - Mouse events, 629–635
  - mouse/MouseComponent.java, 632
  - MouseAdapter class, 632
  - MouseEvent class, 637
    - getClickCount method, 630, 635, 638
    - getPoint method, 635, 638
    - getX/Y methods, 630, 635, 638
    - isPopupTrigger method, 693
    - translatePoint method, 638
  - MouseHandler class, 632
  - MouseListener interface, 631
    - mouseClicked method, 630, 632, 638
    - mouseDragged method, 632
    - mouseEntered/Exited methods, 632, 638
    - mousePressed/Released methods, 630, 638
  - MouseMotionHandler class, 632
  - MouseMotionListener interface, 631–632
    - mouseDragged method, 638
    - mouseMoved method, 631–632, 638
  - MouseWheelEvent class, 637
    - getScrollAmount, getWheelRotation methods, 638
  - MouseWheelListener interface, mouseWheelMoved method, 638
  - Multidimensional arrays, 118–123
    - printing, 246
    - ragged, 121–124
  - Multiple inheritance, 321
    - not supported in Java, 223
  - Multiple selections, 98–103
  - Multiplication operator, 51
  - multiply method
    - of BigDecimal, 109
    - of BigInteger, 108
  - multiplyExact method (Math), 53
  - Multi-release JARs, 201–202
  - Multitasking, 747
  - Multithreading, 8, 747–854
    - deadlocks in, 774, 789–792
    - deferred execution in, 352
    - performance and, 772, 788, 799
    - types of scheduling for, 754
    - synchronization in, 764–797
    - using pools for, 815–820
  - Mutator methods, 138, 476
    - error checking in, 153
- ## N
- n conversion character, 80
  - \n escape sequence, 44, 74
  - NaN (not a number), 43
  - native keyword, 857
  - naturalOrder method (*Comparator*), 357
  - Naughton, Patrick, 10–11
  - NavigableMap interface, 510
    - headMap, subMap, tailMap methods, 558
  - NavigableSet interface, 510, 529
    - ceiling, floor methods, 531
    - descendingIterator method, 532
    - headSet, subSet, tailSet methods, 553, 558
    - higher, lower methods, 531
    - pollFirst/Last methods, 532
  - nCopies method (Collections), 549, 556
  - negateExact method (Math), 53
  - Negation operator, 57
  - Negative infinity, 43
  - .NET platform, 6
  - NetBeans IDE, 22, 27, 441
    - Matisse, 706
  - Netscape, 11
    - IFC library, 582
    - LiveScript/JavaScript, 16
    - Navigator browser, 10
  - Networking, 4
  - new operator, 61, 67, 132, 147, 857
    - in constructor references, 348
    - not for interfaces, 319
    - return value of, 134
    - with arrays, 109
    - with generic classes, 252
    - with threads, 754
  - newCachedThreadPool method (Executors), 818–820
  - newCondition method (*Lock*), 773, 777
  - newFixedThreadPool method (Executors), 818–820
  - newInstance method
    - of Array, 300, 303
    - of Class, 282, 484
    - of Constructor, 283, 484
  - newKeySet method (ConcurrentHashMap), 812
  - Newline. *See* Line feed character
  - newProxyInstance method (Proxy), 379, 384–385

- newScheduledThreadPool method (Executors), 818–820
  - newSingleThreadXxx methods (Executors), 818–820
  - next method
    - of *Iterator*, 501–504, 507
    - of *Scanner*, 78
  - nextDouble method (*Scanner*), 77–78
  - nextElement method (*Enumeration*), 503, 570–571
  - nextInt method
    - of *LinkedList*, 519
    - of *ListIterator*, 522
  - nextInt method
    - of *RandomGenerator*, 178, 180
    - of *Scanner*, 77–78
  - nextLine method (*Scanner*), 76, 78
  - No-argument constructors, 172, 218, 377
  - NoClassDefFoundError, 24
  - node method (*Preferences*), 640, 644
  - noneOf method (*EnumSet*), 547
  - non-sealed keyword, 277, 857
  - NoSuchElementException, 502, 507, 522, 532–533
  - Notepad text editor, 24
  - notHelloWorld/NotHelloWorld.java, 594
  - notify, notifyAll methods (*Objects*), 778, 782
  - now method (*LocalDate*), 136, 141
  - null value, 134, 857
    - as a reference, 149–150
    - equality testing to, 238
  - nullFirst/Last methods (*Comparator*), 357
  - NullPointerException, 59, 149–150, 163, 261, 348, 391, 414
  - Number class, 259
  - NumberFormat class
    - factory methods, 159
    - parse method, 263
  - NumberFormatException, 413
  - Numbers
    - floating-point, 42–43, 51, 55, 80, 96, 229
    - generated random, 178, 180, 796
    - hexadecimal, octal, 41, 80
    - prime, 577
    - rounding, 43, 55, 109
    - unsigned, 42
  - Numeric types
    - casting, 54–55
    - comparing, 57, 357
    - converting:
      - to other numeric types, 53–54, 229
      - to strings, 262
    - default initialization of, 171
    - fixed sizes for, 6
    - precision of, 80, 106
    - printing, 79
- O**
- o conversion character, 80
  - Oak programming language, 10, 391
  - Object class, 128, 235–251
    - clone method, 154, 330–338, 342
    - equals method, 236–241, 251, 326, 552
    - getClass method, 250
    - hashCode method, 242, 244, 527
    - no redefining for methods of, 326
    - notify, notifyAll methods, 778, 782
    - toString method, 244–251, 326, 342
    - wait method, 755, 778, 782
  - Object references
    - as method parameters, 164
    - converting, 229
    - default initialization of, 171
    - modifying, 164
  - Object traversal algorithms, 546
  - Object variables, 267
    - in predefined classes, 132–135
    - initializing, 134
    - setting to null, 134
    - vs. C++ object pointers, 135
    - vs. objects, 133
  - objectAnalyzer/ObjectAnalyzer.java, 298
  - objectAnalyzer/ObjectAnalyzerTest.java, 297
  - Object-oriented programming (OOP), 4, 126–131, 213
    - passing objects in, 326
    - time measurement in, 136
    - vs. procedural, 126–131
  - Objects, 126–129
    - analyzing at runtime, 294–300
    - applying methods to, 133
    - behavior of, 128
    - cloning, 330–338
    - comparing, 319
    - concatenating with strings, 245–246
    - constructing, 127, 170–181

- damaged, 793
- default hash codes of, 242
- destruction of, 180–181
- equality testing for, 236–241, 281
- finalize method of, 180–181
- identity of, 128
- implementing an interface, 319
- in predefined classes, 132–135
- initializing, 133
- intrinsic locks of, 778
- passing to methods, 133
- references to, 134
- runtime type identification of, 280
- serializing, 546
- sorting, 313
- state of, 127–128, 358–362
- vs. object variables, 133

Objects class

- checkXxx methods, 414
- hash, hashCode methods, 243–244
- requireNonNull method, 149, 163, 414
- requireNonNullElse method, 149, 163

Octal numbers

- formatting output for, 80
- prefix for, 41

Octonions, 46, 67

of method

- of EnumSet, 547
- of List, Map, Set, 548–550, 555, 567
- of LocalDate, 137, 141
- of Path, 82, 84, 322
- of ProcessHandle, 850, 853
- of RandomGenerator, 180

ofEntries method (Map), 549, 555

offer method

- of BlockingQueue, 798, 804
- of Queue, 532

offerFirst/Last methods

- of BlockingDeque, 804
- of Deque, 533

offsetByCodePoints method (String), 66, 68

On-demand initialization, 794–795

onExit method (Process), 853

Online documentation, 68, 70–73, 204, 209

open access modifier, 857

OpenJ9 just-in-time compiler, 18

OpenJDK, 17–18

opens keyword, 857

Operators

- arithmetic, 51
- bitwise, 59–61
- boolean, 57
- hierarchy of, 60–61
- increment/decrement, 56–57
- no overloading for, 107
- relational, 57

Option dialogs, 722–726

Optional operations, 554

or method (BitSet), 577

Oracle, 12

Ordered collections, 509, 514

ordinal method (Enum), 273

orTimeout method (CompletableFuture), 835

OSGi platform, 376

Out-of-bounds exceptions, 414

Output

- formatting, 79–82
- statements in, 63

Overloading resolution, 170–171, 225

@Override annotation, 241

overview.html, 209

Owner frame, 727

## P

p (exponent), in hexadecimal numbers, 42

pack method (Window), 593, 595

package statement, 187, 190, 857

package.html, 208

package-info.java, 208

Packages, 186–198

- accessing, 193–194
- adding classes into, 190–193
- documentation comments for, 204, 208
- importing, 187
- names of, 187, 281
- unnamed, 190, 193, 209, 417

PackageTest/com/horstmann/corejava/Employee.java, 192

PackageTest/PackageTest.java, 191

paintComponent method (JComponent), 591–593, 595, 608, 613, 794

- overriding, 637

pair1/PairTest1.java, 452

pair2/PairTest2.java, 456

pair3/PairTest3.java, 482

Parallelism threshold, 811

parallelXxx methods (Arrays), 813–814

- Parameterized types. *See* Type parameters
- ParameterizedType* interface, 485–486
  - getXxx methods, 494
- Parameters, 39, 163–170
  - checking, with assertions, 417–419
  - documentation comments for, 206
  - explicit, 150–151
  - implicit, 150–151, 158, 442
  - modifying, 164–167
  - names of, 174
  - string, 39
  - using collection interfaces in, 569
  - variable number of, 263–265
    - passing generic types to, 464–465
- ParamTest/ParamTest.java*, 168
- Parent classes. *See* Superclasses
- parse method (*NumberFormat*), 263
- parseInt method (*Integer*), 262–263
- Pascal programming language, 10
  - compiled code in, 6
  - passing parameters in, 165
- PasswordChooser* class, 731
- Passwords
  - dialog box for, 731
  - fields for, 662–663
  - reading from console, 78–79
- PATH environment variable, 19
- Path* interface, of method, 82, 84, 322
- Paths* class, get method, 322
- Pattern matching, 232–234
- Payne, Jonathan, 11
- peek method
  - of *BlockingQueue*, 798
  - of *Queue*, 532
  - of *Stack*, 576
- peekFirst/Last methods (*Deque*), 533
- Performance, 7
  - computations and, 51, 53
  - JAR files and, 195
  - measuring, 577–580
  - multithreading and, 772, 788, 799
  - of collections, 509, 525, 805
  - of Java vs. C++, 578
  - of simple tests vs. catching exceptions, 411
- permits keyword, 274–275, 321, 857
- Physical limitations, 389
- PI constant (*Math*), 53, 157–158
- pid method (*ProcessHandle*), 853
- plusDays method (*LocalDate*), 137, 141
- Point* class, 181–182, 598
- Point size (in typesetting), 606
- Point2D* class, 598
- Point2D.Double* class, 598, 603
- Point2D.Float* class, 598
- poll method
  - of *BlockingQueue*, 798, 804
  - of *ExecutorCompletionService*, 826
  - of *Queue*, 532
- pollFirst/Last methods
  - of *Deque*, 533, 804
  - of *NavigableSet*, 532
- Polymorphism, 220, 223–225, 276, 310
- pop method (*Stack*), 576
- Pop-up menus, 692–694
- Portability, 6, 14, 51
- Positive infinity, 43
- pow method (*Math*), 52, 158
- Precision, of numbers, 80
- Preconditions, 418
- Predefined action table names, 625
- Predefined classes, 132–141
  - mutator and accessor methods in, 138–141
  - objects, object variables in, 132–135
- Predicate* interface, 344, 353
- Preemptive scheduling, 754
- Preferences
  - accessing, 640
  - enumerating keys in, 641
  - importing/exporting, 641
- Preferences* class, 639–645
  - exportXxx methods, 641, 645
  - get, get*DataType* methods, 640, 645
  - importPreferences method, 641, 645
  - keys method, 641, 645
  - node method, 640, 644
  - platform-independency of, 639
  - put, put*DataType* methods, 645
  - system/userNodeForPackage methods, 640, 645
  - system/userRoot methods, 640, 644
- preferences/ImageViewer.java*, 642
- preferredLayoutSize method (*LayoutManager*), 721
- previous method (*ListIterator*), 515, 522
- previousIndex method
  - of *LinkedList*, 519
  - of *ListIterator*, 522
- Prime numbers, 577

- Primitive types, 40–46
  - as method parameters, 164
  - comparing, 357
  - converting to objects, 259
  - final fields of, 155
  - not for type parameters, 463
  - transforming hash map values to, 812
  - values of, not object, 236
- Princeton University, 5
- print method (System.out), 39, 79
- printf method (System.out), 79–82
  - conversion characters for, 80
  - flags for, 81
  - parameters of, 263
- println method (System.out), 39, 76, 344, 420
- printStackTrace method (Throwable), 283, 407, 443
- PrintWriter class, 83–84
- Priority queues, 533
- PriorityBlockingQueue class, 799, 803
- PriorityQueue class, 535
  - as a concrete collection type, 511
  - priorityQueue/PriorityQueueTest.java, 534
- private access modifier, 146, 193–194, 360, 857
  - checking, 287
  - for fields, in superclasses, 217
  - for methods, 155
- Procedures, 126
- Process class, 847–854
  - destroy, destroyForcibly methods, 850, 853
  - exitValue method, 850, 853
  - getXxxStream methods, 847–848, 852
  - isAlive method, 850, 853
  - onExit method, 853
  - supportsNormalTermination method, 853
  - toHandle method, 850, 853
  - waitFor method, 849, 852
- process method (SwingWorker), 841–842, 846
- ProcessBuilder class, 847–854
  - directory method, 847, 851
  - environment method, 852
  - inheritIO method, 851
  - redirectXxx methods, 848, 851–852
  - start method, 849, 852
  - startPipeline method, 849, 852
- Processes, 847–854
  - building, 847–849
  - killing, 850
    - running, 849–850
    - vs. threads, 748
- ProcessHandle interface
  - allProcesses method, 850, 853
  - children, descendants methods, 850, 853
  - current method, 850, 853
  - info method, 853
  - of method, 850, 853
  - pid method, 853
- ProcessHandle.Info interface, methods of, 854
- Producer threads, 797
- Programs. *See* Applications
- Properties, 588
  - permitted to retrieve, 575
- Properties class, 569
  - getProperty method, 573–574
  - load, store methods, 572, 574
  - setProperty method, 574
  - stringPropertyNames method, 574
- Property maps, 572–575
  - reading/writing, 572
- PropertyChangeListener interface, 743
- protected access modifier, 234–235, 308, 335, 857
  - provides keyword, 857
- Proxies, 378–385
  - properties of, 383–385
  - purposes of, 380
- Proxy class, 383–385
  - get/isProxyClass methods, 384–385
  - newProxyInstance method, 379, 384–385
- proxy/ProxyTest.java, 382
- public access modifier, 36, 50, 143–146, 193–194, 314, 857
  - checking, 287
  - for fields in interfaces, 320
  - for main method, 37
  - for only one class in source file, 143
  - not specified for interfaces, 313
- publish method, 846
  - of Handler, 431, 439
  - of SwingWorker, 841
- Pure virtual functions (C++), 267
- push method (Stack), 576
- put method
  - of *BlockingQueue*, 798, 804
  - of *ConcurrentHashMap*, 807
  - of *Map*, 508, 536–537
  - of *Preferences*, 641, 645

- putAll method (*Map*), 538
  - putDataType methods (*Preferences*), 641, 645
  - putFirst/Last methods (*BlockingDeque*), 804
  - putIfAbsent method
    - of *ConcurrentHashMap*, 807
    - of *Map*, 540
  - putValue method (*Action*), 624, 629
- Q**
- Queue* interface, 532–533
    - implementing, 499–501
    - methods of, 532
  - Queues, 498–501, 532–533
    - blocking, 797–804
    - concurrent, 805–806
    - double-ended. *See* Deques
  - QuickSort algorithm, 115, 561
- R**
- \r escape sequence, 44
  - Race conditions, 764–768
    - and atomic operations, 788
  - Radio buttons, 670–673, 691–692
  - radioButton/RadioButtonFrame.java, 672
  - Ragged arrays, 121–124
  - Random class, 178, 180
    - thread-safe, 796
  - RandomAccess interface, 509, 561, 564
  - RandomGenerator interface
    - nextInt method, 178, 180
    - of method, 180
  - range method (*EnumSet*), 547
  - Raw types, 457–458
    - converting type parameters to, 473
    - type inquiring at runtime, 463
  - readConfiguration method (*LogManager*), 425, 441
  - readLine/Password methods (*Console*), 79
  - record keyword, 857
  - RecordComponent class, getXxx methods, 294
  - Records, 181–186, 216
    - adding methods to, 183
    - always final, 229
    - declared inside a class, 374
    - equals method of, 237
    - hashCode method of, 243
    - implementing interfaces, 321
    - instance fields of, 182–183
    - toString method of, 247
  - RecordTest/RecordTest.java, 185
  - Rectangle class, 529, 598
  - Rectangle2D class, 596–599
  - Rectangle2D.Double class, 597, 603
  - Rectangle2D.Float class, 597
  - Rectangles, 596
    - comparing, 529
    - drawing, 596
    - filling with color, 603
  - RectangularShape class, 598
    - getCenterX/Y methods, 598, 602
    - getHeight/Width methods, 598, 602
    - getMaxX/Y, getMinX/Y methods, 602
    - getX/Y methods, 602
  - Recursive computations, 827
  - RecursiveAction, RecursiveTask classes, 827
  - Red Hat, 17
  - Red-black trees, 528
  - redirectXxx methods (*ProcessBuilder*), 848, 851–852
  - reduce, reduceXxx methods (*ConcurrentHashMap*), 810–812
  - Reentrant locks, 770
  - ReentrantLock class, 769–772
  - Reflection, 214, 279–307
    - accessing nonpublic features with, 295
    - analyzing:
      - classes, 287–294
      - objects, at runtime, 294–300
    - generics and, 300–303, 483–495
    - overusing, 310
  - reflection/ReflectionTest.java, 289
  - Reinhold, Mark, 12
  - Relational operators, 57, 61
  - Relative resource names, 284
  - remove method
    - of *ArrayList*, 256, 258
    - of *BlockingQueue*, 798
    - of *Collection*, 505–506
    - of *Iterator*, 501, 503–504, 507
    - of *JMenu*, 689
    - of *List*, 509, 521
    - of *ListIterator*, 517
    - of *Map*, 536
    - of *Queue*, 532
    - of *ThreadLocal*, 797
  - removeAll method
    - of *Collection*, 505, 507
    - of *LinkedList*, 519

- removeAllItems method (JComboBox), 678, 680
  - removeEldestEntry method (LinkedHashMap), 544, 546
  - removeFirst/Last methods
    - of *Deque*, 533
    - of *LinkedList*, 522
  - removeHandler method (Logger), 438
  - removeIf method
    - of *ArrayList*, 344
    - of *Collection*, 507, 566
  - removeItem method (JComboBox), 678–679
  - removeItemAt method (JComboBox), 678, 680
  - removeLayoutComponent method (*LayoutManager*), 721
  - removePropertyChangeListener method (*Action*), 624
  - repaint method
    - of *Component*, 592
    - of *JComponent*, 595
  - repeat method (*String*), 63, 70
  - REPL (read-evaluate-print loop), 30
  - replace method
    - of *ConcurrentHashMap*, 807
    - of *String*, 69
  - replaceAll method
    - of *Collections*, 565
    - of *List*, 566
    - of *Map*, 540
  - requireNonNull method (*Objects*), 149, 163, 414
  - requireNonNullElse method (*Objects*), 149, 163
  - Reserved words. *See* Keywords
  - resetChoosableFilters method (*JFileChooser*), 741, 745
  - Resource bundles, 426–427
  - ResourceBundle* class, 426
  - Resources, 284–286
    - exhaustion of, 390
    - localizing, 284
    - names of, 284
  - resources/ResourceTest.java, 286
  - Restricted views, 554
  - resume method (*Thread*), 757
  - retain method (*Collection*), 505
  - retainAll method (*Collection*), 507
  - Retirement/Retirement.java, 92
  - Retirement2/Retirement2.java, 93
  - return statement, 858
    - in finally blocks, 404
    - in lambda expressions, 340
    - not allowed in switch expressions, 102
  - @return comment (javadoc), 206
  - Return types, 226
    - covariant, 461
    - documentation comments for, 206
    - for overridden methods, 459
  - Return values, 134
  - revalidate method (*JComponent*), 660–661
  - reverse method (*Collections*), 565
  - reversed, reverseOrder methods (*Comparator*), 357, 560, 563
  - rotate method (*Collections*), 566
  - round method (*Math*), 55
  - RoundingMode* class, 109
  - rt.jar file, 198
  - run method (*Thread*), 750, 753
  - runAfterXxx methods (*CompletableFuture*), 834, 836
  - runFinalizersOnExit method (*System*), 181
  - Runnable* interface, 353, 748
    - lambda expressions and, 342
    - run method, 352, 753
  - Runtime
    - adding shutdown hooks at, 181
    - analyzing objects at, 294–300
    - creating classes at, 379
    - exec method, 847
    - setting the size of an array at, 251
    - type identification at, 230, 280, 463
  - RuntimeException*, 390–391, 409, 413
- ## S
- S, s conversion characters, 80–81
  - \s escape sequence, 44, 75
  - @SafeVarargs annotation, 465
  - Scala programming language, 324
  - Scanner* class, 76–79, 82–84
    - hasNext method, 78
    - hasNextXxx methods, 79
    - next method, 78
    - nextXxx methods, 77–78
  - Scheduled execution, 820
  - ScheduledExecutorService* class, methods of, 821
  - Scroll panes, 663–667
  - sealed keyword, 274, 321, 858
  - sealed/SealedTest.java, 278



- search, *searchXxx* methods (ConcurrentHashMap), 810–812
- Security, 5, 15
- @see comment (javadoc), 207–208
- Semantic events, 637
- Serialization, 546
- Service loaders, 376–378
- ServiceLoader class, 376
  - iterator, load methods, 378
  - stream method, 377–378
- ServiceLoader.Provider* interface, methods of, 377–378
- Services, 376–378
- ServletException, 400
- Servlets, 400
- Set interface
  - add, equals, hashCode, methods of, 510
  - copyOf method, 550, 555
  - of method, 548–550, 555
- set method
  - of Array, 303
  - of ArrayList, 255, 258
  - of BitSet, 577
  - of Field, 300
  - of List, 509, 521
  - of *ListIterator*, 517, 522
  - of ThreadLocal, 797
  - of Vector, 784
- set/SetTest.java, 526
- setAccelerator method (JMenuItem), 695–696
- setAcceptAllFileFilterUsed method (JFileChooser), 741, 745
- setAccessible method (AccessibleObject), 295, 299
- setAccessory method (JFileChooser), 745
- setAction method (AbstractButton), 689
- setActionCommand method (AbstractButton), 673
- setBackground method (Component), 604–605
- setBoolean method (Array), 303
- setBorder method (JComponent), 674, 676
- setBounds method (Component), 586, 588–589
- setByte, setChar methods (Array), 303
- setCharAt method (StringBuilder), 74
- setClassAssertionStatus method (ClassLoader), 420
- setColumns method
  - of JTextArea, 663, 666
  - of JTextField, 659, 661
- setComponentPopupMenu method (JComponent), 693–694
- setCurrentDirectory method (JFileChooser), 739, 744
- setCursor method (Component), 635
- setDaemon method (Thread), 761
- setDefaultAssertionStatus method (ClassLoader), 420
- setDefaultButton method (JRootPane), 733, 737
- setDefaultCloseOperation method (JDialog), 728
- setDefaultUncaughtExceptionHandler method (Thread), 444, 761–762
- setDisplayedMnemonicIndex method (AbstractButton), 695–696
- setDouble method (Array), 303
- setEchoChar method (JPasswordField), 663
- setEditable method
  - of JComboBox, 676, 679
  - of JTextComponent, 659
- setEnabled method
  - of Action, 624, 629
  - of JMenuItem, 697–698
- setFileFilter method (JFileChooser), 741, 745
- setFileSelectionMode method (JFileChooser), 739, 744
- setFileView method (JFileChooser), 741–742, 745
- setFilter method
  - of Handler, 439
  - of Logger, 431, 438
- setFloat method (Array), 303
- setFont method (JComponent), 661
- setForeground method (Component), 604–605
- setFormatter method (Handler), 432, 439
- setFrameFromCenter method (Ellipse2D), 599
- setHorizontalTextPosition method (AbstractButton), 690–691
- setIcon method
  - of JLabel, 662
  - of JMenuItem, 690
- setIconImage method (Frame), 586, 590
- setInheritsPopupMenu method (JComponent), 693–694
- setInt method (Array), 303
- setInverted method (JSlider), 682
- setJMenuBar method (JFrame), 687, 690
- setLabelTable method (JSlider), 461, 682, 686
- setLayout method (Container), 655

- setLevel method
  - of Handler, 439
  - of Logger, 421, 438
- setLineWrap method (JTextArea), 664, 666
- setLocation method (Component), 586, 588–589
- setLocationByPlatform method (Window), 589
- setLong method (Array), 303
- setMajorTickSpacing, setMinorTickSpacing methods (JSlider), 686
- setMnemonic method (AbstractButton), 695–696
- setModel method (JComboBox), 678
- setMultiSelectionEnabled method (JFileChooser), 739, 744
- setOut method (System), 158
- setPackageAssertionStatus method (ClassLoader), 420
- setPaint method (Graphics2D), 603–604
- setPaintLabels method (JSlider), 682, 686
- setPaintTicks method (JSlider), 681–682, 686
- setPaintTrack method (JSlider), 686
- setParent method (Logger), 438
- setPriority method (Thread), 763
- setProperty method
  - of Properties, 574
  - of System, 425
- setResizable method (Frame), 586, 589
- setRows method (JTextArea), 663, 666
- Sets, 525
  - concurrent, 805–806
  - intersecting, 566
  - mutating elements of, 526
  - subranges of, 552
  - thread-safe, 812–813
  - unmodifiable, 555
  - with given elements, 548–550
- setSelected method
  - of AbstractButton, 692
  - of JCheckBox, 668–669
- setSelectedFile/Files methods (JFileChooser), 739, 744
- setShort method (Array), 303
- setSize method (Component), 589
- setSnapToTicks method (JSlider), 681, 686
- setTabSize method (JTextArea), 666
- setText method
  - of JLabel, 662
  - of JTextComponent, 659–660
- setTime method (Calendar), 228
- setTitle method (JFrame), 586, 590
- setToolTipText method (JComponent), 705
- setUncaughtExceptionHandler method (Thread), 762
- setUseParentHandlers method (Logger), 438
- setValue method (*Map.Entry*), 542
- setVisible method
  - of Component, 586, 589
  - of JDialog, 728, 730–731
- setWrapStyleWord method (JTextArea), 666
- severe method (Logger), 422, 437
- Shallow copies, 332–334
- Shape interface, 596
- Shell
  - redirection syntax of, 84
  - scripts in, 197
- Shift operators, 60
- Short class
  - converting from short, 259
  - hashCode method, 244
- short type, 40, 858
- show method (JPopupMenu), 693
- showConfirmDialog method (JOptionPane), 722–725
- showDialog method (JFileChooser), 732, 738–739, 744
- showInputDialog method (JOptionPane), 722–723, 726
- showInternalConfirmDialog method (JOptionPane), 725
- showInternalInputDialog method (JOptionPane), 726
- showInternalMessageDialog method (JOptionPane), 725
- showInternalOptionDialog method (JOptionPane), 726
- showMessageDialog method (JOptionPane), 329, 722–725
- showOpenDialog method (JFileChooser), 738–739, 744
- showOptionDialog method (JOptionPane), 722–724, 726
- showSaveDialog method (JFileChooser), 738–739, 744
- shuffle method (Collections), 561–562
- shuffle/ShuffleTest.java, 562
- Shuffling, 561
- Shutdown hooks, 181
- shutdown method (*ExecutorService*), 819–820
- shutdownNow method (*ExecutorService*), 819, 821

- Sieve of Eratosthenes benchmark, 577–580
- sieve/sieve.cpp, 579
- sieve/Sieve.java, 578
- signal method (Condition), 775–777, 791
- signalAll method (Condition), 774–777, 791
- Signatures (of methods), 171, 226
- simpleFrame/SimpleFrameTest.java, 584
- sin method (Math), 52
- singleton, singletonXxx methods (Collections), 550, 557
- size method
  - of ArrayList, 253–254
  - of BitSet, 577
  - of Collection, 505–506
  - of concurrent collections, 805
- sleep method (Thread), 749, 753, 758
- slider/SliderFrame.java, 683
- Sliders, 680–686
  - ticks on, 680, 682
  - vertical, 680
- Smart cards, 4
- SoftBevelBorder class, 674, 676
- sort method
  - of Arrays, 115–117, 313, 316, 318, 339, 343
  - of Collections, 560–563
  - of List, 562
- SortedMap interface, 510
  - comparator, first/lastKey methods, 539
  - headMap, subMap, tailMap methods, 552, 558
- SortedSet interface, 510
  - comparator, first, last methods, 531
  - headSet, subSet, tailSet methods, 552, 557
- Sorting
  - algorithms for, 115, 560–563
  - arrays, 115–118, 316
  - assertions for, 418
  - order of, 560
  - people, by name, 356–357
  - strings by length, 330, 338–341
- Source files, 197
  - editing in Eclipse, 29
  - installing, 20–22
- Space. *See* Whitespace
- Special characters, 44
- Spring layout, 705
- sqrt method
  - of BigInteger, 108
  - of Math, 52, 305–306
- src.zip file, 20
- Stack interface, 498, 569, 575
  - methods of, 576
- Stack trace, 407–411, 790
  - no displaying to users, 415
- StackFrame class
  - getXxx methods, 410
  - isNativeMethod method, 410
  - toString method, 407, 410
- Stacks, 575
- stackTrace/StackTraceTest.java, 408
- StackTraceElement class, methods of, 411
- StackWalker class, 407
  - forEach method, 410
  - getInstance method, 407, 410
  - walk method, 407, 410
- Standard Edition (Java SE), 12, 18
- Standard Java library
  - companion classes in, 322
  - online API documentation for, 68, 70–73, 204, 209
- Standard Template Library (STL), 498, 503
- start method
  - of ProcessBuilder, 849, 852
  - of Thread, 750, 753–754
  - of Timer, 329
- Starting directory, for a program, 83
- startInstant method (*ProcessHandle.Info*), 854
- startPipeline method (*ProcessBuilder*), 849, 852
- startsWith method (String), 69
- stateChanged method (*ChangeListener*), 680
- Statements, 38
  - compound. *See* Blocks
  - conditional, 86–89
  - in output, 63
- static access modifier, 156–163, 858
  - for fields in interfaces, 320
  - for main method, 38
- Static binding, 226
- Static constants, 157–158
  - documentation comments for, 207
- Static fields, 156–157
  - accessing, in static methods, 158
  - importing, 189
  - initializing, 177
  - no type variables in, 468
- static final access modifier, 49
- Static imports, 189

- Static inner classes, 358, 372–375
- Static methods, 158–159
  - accessing static fields in, 158
  - adding to interfaces, 322
  - importing, 189
  - no type variables in, 468
- Static variables, 157
- staticInnerClass/StaticInnerClassTest.java, 374
- StaticTest/StaticTest.java, 161
- stop method
  - of Thread (deprecated), 757, 793–794
  - of Timer, 329
- store method (Properties), 572, 574
- Stream interface, toArray method, 349
- stream method
  - of BitSet, 577
  - of Collection, 324
  - of ServiceLoader, 377–378
- StreamHandler class, 430
- strictfp keyword, 858
- StrictMath class, 52–53
- String class, 61–76
  - charAt method, 66, 68
  - codePointAt method, 68
  - codePointCount method, 66, 69
  - codePoints method, 67–68
  - compareTo method, 68
  - endsWith method, 69
  - equals, equalsIgnoreCase methods, 64, 69
  - format, formatted, formatTo methods, 82
  - hashCode method, 241, 523
  - immutability of, 63, 155, 229
  - implementing *CharSequence*, 322
  - indexOf method, 69, 171
  - isBlank, isEmpty methods, 69
  - join method, 70
  - lastIndexOf method, 69
  - length method, 65–66, 69
  - offsetByCodePoints method, 66, 68
  - repeat method, 63, 70
  - replace method, 69
  - startsWith method, 69
  - strip method, 70, 660
  - stripLeading/Trailing methods, 70
  - substring method, 62, 70, 552
  - toLowerCase, toUpperCase methods, 70
  - transform method, 355–356
  - trim method, 70
- StringBuffer class, 73
- StringBuilder class, 73–74
  - append method, 73–74
  - appendCodePoint method, 74
  - delete method, 74
  - implementing *CharSequence*, 322
  - insert method, 74
  - length method, 73
  - setCharAt method, 74
  - toString method, 73–74
- stringPropertyNames method (Properties), 574
- Strings, 61–76
  - building, 73–74
  - code points/code units of, 66
  - comparing, 330
  - concatenating, 62–63
    - with objects, 245–246
  - converting to numbers, 262
  - empty, 65, 69
  - equality of, 64
  - formatting output for, 79–82
  - immutability of, 63
  - length of, 62, 65
  - null, 65
  - shared, in compiler, 63, 65
  - sorting by length, 330, 338–341
  - spanning multiple lines, 74
  - substrings of, 62
  - using ". . ." for, 39
- strip method (String), 70, 660
- stripLeading/Trailing methods (String), 70
- Strongly typed languages, 40, 315
- Subclasses, 214–235
  - adding fields/methods to, 218
  - anonymous, 370, 449
  - cloning, 335
  - comparing objects from, 319
  - constructors for, 218
  - defining, 214
  - forbidding, 274
  - method visibility in, 228
  - no access to private fields of superclass, 234
  - non-sealed, 277
  - overriding superclass methods in, 218
- subList method (*List*), 552, 557
- subMap method
  - of *NavigableMap*, 558
  - of *SortedMap*, 552, 558
- Submenus, 687

- submit method
    - of `ExecutorCompletionService`, 826
    - of `ExecutorService`, 819–820
  - Subranges, 552–553
  - subSet method
    - of `NavigableSet`, 553, 558
    - of `SortedSet`, 552, 557
  - Substitution principle, 223
  - substring method (`String`), 62, 70, 552
  - subtract method
    - of `BigDecimal`, 109
    - of `BigInteger`, 108
  - subtractExact method (`Math`), 53
  - Subtraction operator, 51
  - sum method (`LongAdder`), 788
  - Sun Microsystems, 2, 5–12, 15, 582
    - HotJava browser, 11
  - super keyword, 217, 477, 858
    - in method references, 348
    - vs. `this`, 217–218
  - Superclass wins rule, 324
  - Superclasses, 214–235
    - accessing private fields of, 217
    - common fields and methods in, 267, 308
    - overriding methods of, 241
    - throws specifiers in, 394, 399
  - Supertype bounds, 476–479
  - Supplementary characters, 45
  - `Supplier` interface, 353
  - `supportsNormalTermination` method (`Process`), 853
  - `@SuppressWarnings` annotation, 101, 259, 462, 465, 469–471
  - Surrogates area (`Unicode`), 45
  - suspend method (`Thread`, deprecated), 757, 793–794
  - swap method (`Collections`), 565
  - Swing toolkit, 581–645, 840
    - building GUI with, 647–746
    - model-view-controller analysis of, 650, 652
    - starting, 585
  - `SwingConstants` interface, 661
  - `SwingUtilities` class, `getAncestorOfClass` method, 732, 737
  - `SwingWorker` class, 840
    - `doInBackground` method, 841–842, 846
    - `execute` method, 841, 846
    - `getState` method, 846
    - process, `publish` methods, 841–842, 846
  - `swingWorker/SwingWorkerTest.java`, 843
  - switch statement, 58–59, 98–103, 858
    - enumerated constants in, 59
    - throwing exceptions in, 102
    - value of, 58
    - with fallthrough, 101
    - with pattern matching, 275
  - `synch/Bank.java`, 775
  - `synch2/Bank.java`, 780
  - Synchronization, 764–797
    - condition objects for, 772–777
    - final variables and, 787
    - in `Vector`, 523
    - lock objects for, 769–772
    - monitor concept for, 784–785
    - race conditions in, 764–768, 788
    - volatile fields and, 785–787
  - Synchronization wrappers, 814–815
  - Synchronized blocks, 782–784
  - synchronized keyword, 769, 778–785, 858
  - Synchronized views, 553–554
  - `synchronizedCollection` methods (`Collections`), 553–554, 556, 815
  - System class
    - `console` method, 79
    - `exit` method, 38
    - `getProperties` method, 573, 575
    - `getProperty` method, 83, 575
    - `identityHashCode` method, 545, 548
    - `runFinalizersOnExit` method, 181
    - `setOut` method, 158
    - `setProperty` method, 425
  - `System.err` object, 444
  - `System.in` object, 76
  - `System.out` object, 39, 157, 444
    - `print` method, 79
    - `printf` method, 79–82, 263
    - `println` method, 76, 420
  - `systemNodeForPackage` method (`Preferences`), 645
  - `systemNodeForPackage`, `systemRoot` methods (`Preferences`), 640
  - `systemRoot` method (`Preferences`), 644
- ## T
- T type variable, 451
  - T, t conversion characters, 80
  - `\t` escape sequence, 44
  - Tab completion, 32
  - Tabs, in text blocks, 76

- Tagging interfaces, 334, 458, 509
- tailMap method
  - of *NavigableMap*, 558
  - of *SortedMap*, 552, 558
- tailSet method
  - of *NavigableSet*, 553, 558
  - of *SortedSet*, 552, 557
- take method
  - of *BlockingQueue*, 798, 804
  - of *ExecutorCompletionService*, 826
- takeFirst/Last methods (*BlockingDeque*), 804
- tan method (*Math*), 52
- tar command, 198
- Tasks
  - asynchronously running, 816
  - controlling groups of, 821–826
  - decoupling from mechanism of running, 750
  - long-running, 839–846
  - multiple, 747
  - scheduled, 820
  - work stealing for, 828
- Template code bloat, 457
- Terminal window, 24
- Text
  - centering, 607
  - displaying, 593
  - fonts for, 605–612
  - typesetting properties of, 607
- Text areas, 663–664
  - formatted text in, 665
  - preferred size of, 663
- Text blocks, 74–76
- Text fields, 658–662
  - columns in, 659
  - creating blank, 660
  - preferred size of, 659
- Text input, 658–667
  - labels for, 661–662
  - password fields, 662–663
  - scroll panes, 663
- text/TextComponentFrame.java, 665
- thenAccept, thenAcceptBoth, thenCombine methods (*CompletableFuture*), 834–835
- thenApply, thenApplyAsync methods (*CompletableFuture*), 833, 835
- thenComparing method (*Comparator*), 356–357
- thenCompose method (*CompletableFuture*), 833, 835
- thenRun method (*CompletableFuture*), 835
- this keyword, 151, 174, 858
  - in first statement of constructor, 175
  - in inner classes, 363
  - in lambda expressions, 351
  - in method references, 348
  - vs. super, 217–218
- Thread class
  - currentThread method, 757–760
  - extending, 750
  - get/setUncaughtExceptionHandler methods, 762
  - getDefaultUncaughtExceptionHandler method, 762
  - getState method, 757
  - interrupt, isInterrupted methods, 757–760
  - interrupted method, 759–760
  - join method, 755–757
  - methods with timeout, 755
  - resumes method, 757
  - run method, 750, 753
  - setDaemon method, 761
  - setDefaultUncaughtExceptionHandler method, 444, 761–762
  - setPriority method, 763
  - sleep method, 749, 753, 758
  - start method, 750, 753–754
  - stop method (deprecated), 757, 793–794
  - suspend method (deprecated), 757, 793–794
  - yield method, 755
- Thread dump, 791
- Thread groups, 762
- Thread pools, 815–820
- Thread.UncaughtExceptionHandler* interface, 761–763
- ThreadDeath error, 756, 763, 793
- ThreadGroup class, 762
  - uncaughtException method, 762–763
- ThreadLocal class, methods of, 797
- ThreadLocalRandom class, current method, 797
- ThreadPoolExecutor class, 818–819
  - getLargestPoolSize method, 820
- Threads, 748–753
  - accessing collections from, 553–554, 797–815
  - blocked, 755–756, 758
  - condition objects for, 772–777
  - daemon, 761
  - executing code in, 352

- idle, 827
- interrupting, 757–760
- listing all, 791
- locking, 782–784
- new, 754
- preemptive vs. cooperative scheduling
  - for, 754
- priorities of, 763
- producer/customer, 797
- runnable, 754–755
- states of, 753–757
- synchronizing, 764–797
- terminated, 749, 756–757
- thread-local variables in, 795–797
- timed waiting, 755–756
- unblocking, 775
- uncaught exceptions in, 761–763
- vs. processes, 748
- waiting, 755–756, 773
- work stealing for, 828
- worker, 839–846
- threads/Bank.java, 752
- threads/ThreadTest.java, 750
- Thread-safe collections, 797–815
  - callable and futures, 816–818
  - concurrent, 805–806
  - copy on write arrays, 813
  - synchronization wrappers, 814–815
- throw keyword, 394–395, 858
- Throwable class, 390, 413
  - add/getSuppressed methods, 406, 409
  - get/initCause methods, 409
  - getMessage method, 396
  - getStackTrace method, 407, 409
  - printStackTrace method, 283, 407, 443
  - toString method, 396
- throwing method (Logger), 424, 437
- throws keyword, 283, 391–394, 858
  - for main method, 84
- @throws comment (javadoc), 206
- Ticks, 680
  - icons for, 682
  - labeling, 682
  - snapping to, 681
- Time measurement vs. calendars, 136
- Timed waiting threads, 755–756
- TimeoutException, 816, 835
- Timer class, 326, 338, 638
  - start, stop methods, 329
- timer/TimerTest.java, 328
- to keyword, 858
- toArray method
  - of ArrayList, 468
  - of Collection, 256, 505, 507, 567
  - of Stream, 349
- toHandle method (Process), 850, 853
- toLowerCase method (String), 70
- Toolbars, 701–704
  - detaching, 702
  - dragging, 702
  - title of, 703
  - vertical, 703
- Toolkit class
  - beep method, 329
  - getDefaultToolkit method, 329, 588, 590
  - getScreenSize method, 588, 590
- Tooltips, 704–705
- toString method
  - adding to all classes, 247
  - Formattable* and, 81
  - of Arrays, 113, 117
  - of Date, 133
  - of Enum, 272–273
  - of Integer, 263
  - of Modifier, 287, 294
  - of Object, 244–251, 326
  - of proxy classes, 384
  - of records, 182, 247
  - of StackFrame, 407, 410
  - of StackTraceElement, 411
  - of StringBuilder, 73–74
  - of Throwable, 396
  - redeclaring, 342
  - working with any class, 296–297
- Total ordering, 529
- totalCpuDuration method (*ProcessHandle.Info*), 854
- toUnsignedInt method (Byte), 42
- toUpperCase method (String), 70
- TraceHandler class, 380
- Tracing execution flow, 423
- TransferQueue* interface, 800
  - transfer, tryTransfer methods, 804
- transform method (String), 355–356
- transient keyword, 858
- transitive keyword, 858
- translatePoint method (MouseEvent), 638
- Tree maps, 535

- Tree sets, 527–532
  - adding elements to, 528
  - red-black, 528
  - total ordering of, 529
  - vs. priority queues, 534
- TreeMap class, 510, 535, 538
  - as a concrete collection type, 511
  - vs. HashMap, 535
- TreeSet class, 510, 527–532
  - as a concrete collection type, 511
- treeSet/Item.java, 530
- treeSet/TreeSetTest.java, 529
- Trigonometric functions, 52
- trim method (String), 70
- trimToSize method (ArrayList), 254
- Troubleshooting. *See* Debugging
- true value, 858
- Truncated computations, 51
- try keyword, 858
- try/catch statement, 397–402
  - generics and, 469
  - wrapping entire task in try block, 412
- try/finally statement, 402–405
- tryLock method (*Lock*), 755
- trySetAccessible method (AccessibleObject), 299
- try-with-resources statement, 405–406
  - effectively final variables in, 406
  - no locks with, 769
- Two-dimensional arrays, 118–123
- Type erasure, 457–463
  - clashes after, 471–472
- Type interface, 485–486
- type method (*ServiceLoader.Provider*), 377–378
- Type parameters, 251
  - converting to raw types, 473
  - not for arrays, 463–464, 473
  - not with primitive types, 463
  - vs. inheritance, 448
- Type variables
  - bounds for, 454–456
  - in exceptions, 469
  - in static fields or methods, 468
  - matching in generic methods, 484–485
  - names of, 451
  - no instantiating for, 465–466
  - replacing with bound types, 457–458
- Typesetting terms, 607
- TypeVariable interface, 485–486
  - getBounds, getName methods, 494
- U**
  - \u escape sequence, 44–45
  - UCSD Pascal system, 6
  - UML (Unified Modeling Language)
    - notation, 130–131
  - UnaryOperator interface, 353
  - uncaughtException method (ThreadGroup), 762–763
  - UncaughtExceptionHandler interface, 761–763
    - uncaughtException method, 762
  - Unchecked exceptions, 283, 391–393, 413
  - Unequality operator, 57
  - Unicode standard, 6, 43–46, 61
  - Unit testing, 160
  - University of Illinois, 11
  - UNIX operating system, 195–197
  - unlock method (*Lock*), 769, 771
  - Unmodifiable copies, 550–552, 555
  - Unmodifiable views, 550–552
  - unmodifiableCollection methods (Collections), 551–552, 556
  - Unnamed modules, 296
  - Unnamed packages, 190, 193, 209, 417
  - UnsupportedOperationException, 541, 549, 551, 554, 556
  - unsynch/UnsynchBankTest.java, 766
  - updateAndGet method (*AtomicType*), 788
  - updateConfiguration method (*LogManager*), 425, 441
  - User input, 389, 660
  - User Interface. *See* Graphical User Interface
  - user method (*ProcessHandle.Info*), 854
  - User-defined types, 262
  - userNodeForPackage method (*Preferences*), 640, 645
  - userRoot method (*Preferences*), 640, 644
  - uses keyword, 858
  - “Uses-a” relationship, 130–131
  - UTC (Coordinated Universal Time), 136
  - UTF-8 standard, 83
  - Utility classes/methods, 322, 324
- V**
  - V type variable, 451
  - validate method (*Component*), 661
  - valueOf method
    - of BigInteger, 106–108
    - of Enum, 272–273
    - of Integer, 263



- values method (*Map*), 540, 542
  - var keyword, 148, 341, 369, 859
    - diamond syntax and, 252
  - Varargs, 263–265
    - passing generic types to, 464–465
  - VarHandle class, 296
  - Variable handles, 296
  - Variables, 47–48
    - accessing:
      - from outer methods, 366–367
      - in lambda expressions, 349–352
    - annotating, 462
    - copying, 331
    - declarations of, 47, 232–234
    - effectively final, 351, 406
    - initializing, 48, 210
    - local, 148, 234, 462
    - mutating in lambda expressions, 351
    - names of, 47–48
    - package scope of, 193
    - printing/logging values of, 442
    - static, 157
    - thread-local, 795–797
  - Vector class, 498, 569–570, 783–784, 814–815
    - for dynamic arrays, 252
    - get, set methods, 784
    - synchronization in, 523
  - @version comment (*javadoc*), 207, 209
  - Views, 548–558, 648
    - bulk operations for, 567
    - checked, 553
    - restricted, 554
    - subranges of, 552–553
    - synchronized, 553–554
    - unmodifiable, 550–552
  - Visual Basic programming language
    - built-in date type in, 132
    - event handling in, 614
    - syntax of, 3
  - Visual Studio, 22
  - void keyword, 38, 859
  - Volatile fields, 785–787
  - volatile keyword, 786–787, 859
  - von der Ahé, Peter, 454
- W**
- wait method (*Object*), 755, 778, 782
  - Wait sets, 773
  - waitFor method (*Process*), 849, 852
  - walk method (*StackWalker*), 407, 410
  - warning method (*Logger*), 422, 437
  - Warnings
    - fallthrough behavior and, 101
    - generic, 259, 462, 465, 469–471
    - suppressing, 465, 469–471
    - when using reflection, 295
  - Weak hash maps, 542–543
  - Weak references, 543
  - WeakHashMap class, 542–543, 546
    - as a concrete collection type, 511
  - Weakly consistent iterators, 805
  - WeakReference object, 543
  - Web pages
    - dynamic, 9
    - extracting links from, 832
    - reading, 833, 839
  - Welcome/Welcome.java, 23
  - whenComplete method (*CompletableFuture*), 835
  - while loop, 89–94, 859
  - Whitespace
    - escape sequence for, 44, 75
    - in text blocks, 75
    - irrelevant to compiler, 38
    - leading/trailing, 70, 75, 660
  - Wildcard types, 450, 475–483
    - arrays of, 464
    - capturing, 480–483
    - supertype bounds for, 476–479
    - unbounded, 480
  - WildcardType interface, 485–486
    - getLowerBounds, getUpperBounds methods, 494
  - Window class, 639
    - is/setLocationByPlatform methods, 589
    - pack method, 593, 595
  - Window listeners, 621–623
  - WindowClosing event, 695
  - WindowEvent class, 614, 621, 637
    - getXxx methods of, 639
  - WindowFocusListener interface, methods of, 639
  - WindowListener interface, methods of, 621–623, 639
  - Windows. *See* Dialogs
  - Windows operating system
    - Alt+F4 keyboard shortcut in, 695
    - default location in, 428
    - executing JARs in, 201

- IDEs for, 27
  - JDK in, 17, 19
  - paths in, 19–21, 195, 197
  - pop-up menus in, 693
  - registry in, 639, 641
  - thread priority levels in, 763
  - WindowStateListener* interface, *windowStateChanged* method, 623, 639
  - Wirth, Niklaus, 6, 10, 126
  - with keyword, 859
  - withInitial* method (*ThreadLocal*), 797
  - Work stealing, 828
  - Worker threads, 839–846
  - Working directory, for a process, 847
  - Wrappers, 259–263
    - class constructors for, 261
    - equality testing for, 261
    - immutability of, 259
    - locks and, 261, 783
- X**
- $\chi$ , x conversion characters, 80
  - XML (Extensible Markup Language), 12, 14
  - xor* method (*BitSet*), 577
- Y**
- yield* method (*Thread*), 754–755
  - yield* statement, 101–103, 859
- Z**
- ZIP format, 195, 198