Updated through Java 17

Core Java for the Impatient

Third Edition

Cay S. Horstmann

FREE SAMPLE CHAPTER

Core Java for the Impatient

Third Edition

This page intentionally left blank

Core Java for the Impatient

Third Edition

Cay S. Horstmann

Addison-Wesley

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 illustration by Morphart Creation / Shutterstock
Figures 1.1 and 1.3: Microsoft
Figure 1.2: Eclipse Foundation
Figures 1.4, 1.5, 11.1: Oracle Corporation

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

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

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

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

For questions about sales outside the United States, please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2022942895

Copyright © 2023 Pearson Education, Inc.

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/.

ISBN-13: 978-0-13-805210-2
ISBN-10: 0-13-805210-7

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

To Chi—the most patient person in my life.

This page intentionally left blank

Contents

Preface xxiii

Acknowledgments xxv

About the Author xxvii

I FUNDAMENTAL PROGRAMMING STRUCTURES 1

1.1 Our First Program 2

1.1.1 Dissecting the “Hello, World” Program 2

1.1.2 Compiling and Running a Java Program 4

1.1.3 Method Calls 6

1.1.4 JShell 7

1.2 Primitive Types 11

1.2.1 Signed Integer Types 11

1.2.2 Floating-Point Types 13

1.2.3 The `char` Type 14

1.2.4 The `boolean` Type 14

1.3 Variables 14

1.3.1 Variable Declarations 15

1.3.2 Identifiers 15

1.3.3 Initialization 16
1.3.4 Constants 16
1.4 Arithmetic Operations 17
1.4.1 Assignment 18
1.4.2 Basic Arithmetic 19
1.4.3 Mathematical Methods 20
1.4.4 Number Type Conversions 21
1.4.5 Relational and Logical Operators 22
1.4.6 Big Numbers 24
1.5 Strings 25
1.5.1 Concatenation 25
1.5.2 Substrings 26
1.5.3 String Comparison 26
1.5.4 Converting Between Numbers and Strings 28
1.5.5 The String API 28
1.5.6 Code Points and Code Units 31
1.5.7 Text Blocks 33
1.6 Input and Output 35
1.6.1 Reading Input 35
1.6.2 Formatted Output 36
1.7 Control Flow 38
1.7.1 Branches 38
1.7.2 Switches 39
1.7.3 Loops 41
1.7.4 Breaking and Continuing 43
1.7.5 Local Variable Scope 45
1.8 Arrays and Array Lists 46
1.8.1 Working with Arrays 46
1.8.2 Array Construction 47
1.8.3 Array Lists 48
1.8.4 Wrapper Classes for Primitive Types 49
1.8.5 The Enhanced `for` Loop 50
1.8.6 Copying Arrays and Array Lists 51
1.8.7 Array Algorithms 52

1.8.8 Command-Line Arguments 52
1.8.9 Multidimensional Arrays 53
1.9 Functional Decomposition 56
1.9.1 Declaring and Calling Static Methods 56
1.9.2 Array Parameters and Return Values 56
1.9.3 Variable Arguments 57
Exercises 58

2 OBJECT-ORIENTED PROGRAMMING 61
2.1 Working with Objects 62
2.1.1 Accessor and Mutator Methods 64
2.1.2 Object References 65
2.2 Implementing Classes 67
2.2.1 Instance Variables 67
2.2.2 Method Headers 67
2.2.3 Method Bodies 68
2.2.4 Instance Method Invocations 68
2.2.5 The `this` Reference 69
2.2.6 Call by Value 70
2.3 Object Construction 71
2.3.1 Implementing Constructors 71
2.3.2 Overloading 72
2.3.3 Calling One Constructor from Another 73
2.3.4 Default Initialization 73
2.3.5 Instance Variable Initialization 74
2.3.6 Final Instance Variables 75
2.3.7 The Constructor with No Arguments 75
2.4 Records 76
2.4.1 The Record Concept 77
2.4.2 Constructors: Canonical, Custom, and Compact 78
2.5 Static Variables and Methods 79
2.5.1 Static Variables 79
2.5.2 Static Constants 80
2.5.3 Static Initialization Blocks 81

2.5.4 Static Methods 81
2.5.5 Factory Methods 83
2.6 Packages 83
2.6.1 Package Declarations 83
2.6.2 The `jar` Command 85
2.6.3 The Class Path 86
2.6.4 Package Access 87
2.6.5 Importing Classes 88
2.6.6 Static Imports 89
2.7 Nested Classes 90
2.7.1 Static Nested Classes 90
2.7.2 Inner Classes 91
2.7.3 Special Syntax Rules for Inner Classes 94
2.8 Documentation Comments 95
2.8.1 Comment Insertion 95
2.8.2 Class Comments 96
2.8.3 Method Comments 97
2.8.4 Variable Comments 97
2.8.5 General Comments 97
2.8.6 Links 98
2.8.7 Package, Module, and Overview Comments 99
2.8.8 Comment Extraction 99
Exercises 100

3 INTERFACES AND LAMBDA EXPRESSIONS 105
3.1 Interfaces 106
3.1.1 Using Interfaces 106
3.1.2 Declaring an Interface 107
3.1.3 Implementing an Interface 108
3.1.4 Converting to an Interface Type 110
3.1.5 Casts and the `instanceof` Operator 110
3.1.6 The "Pattern-Matching" Form of `instanceof` 111
3.1.7 Extending Interfaces 112

3.1.8 Implementing Multiple Interfaces 113
3.1.9 Constants 113
3.2 Static, Default, and Private Methods 113
3.2.1 Static Methods 113
3.2.2 Default Methods 114
3.2.3 Resolving Default Method Conflicts 115
3.2.4 Private Methods 117
3.3 Examples of Interfaces 117
3.3.1 The `Comparable` Interface 117
3.3.2 The `Comparator` Interface 119
3.3.3 The `Runnable` Interface 120
3.3.4 User Interface Callbacks 120
3.4 Lambda Expressions 121
3.4.1 The Syntax of Lambda Expressions 122
3.4.2 Functional Interfaces 123
3.5 Method and Constructor References 124
3.5.1 Method References 124
3.5.2 Constructor References 126
3.6 Processing Lambda Expressions 127
3.6.1 Implementing Deferred Execution 127
3.6.2 Choosing a Functional Interface 128
3.6.3 Implementing Your Own Functional Interfaces 130
3.7 Lambda Expressions and Variable Scope 131
3.7.1 Scope of a Lambda Expression 131
3.7.2 Accessing Variables from the Enclosing Scope 132
3.8 Higher-Order Functions 135
3.8.1 Methods that Return Functions 135
3.8.2 Methods That Modify Functions 135
3.8.3 `Comparator` Methods 136
3.9 Local and Anonymous Classes 137
3.9.1 Local Classes 137
3.9.2 Anonymous Classes 138
Exercises 139

4 INHERITANCE AND REFLECTION 143
4.1 Extending a Class 144
4.1.1 Super- and Subclasses 145
4.1.2 Defining and Inheriting Subclass Methods 145
4.1.3 Method Overriding 145
4.1.4 Subclass Construction 147
4.1.5 Superclass Assignments 147
4.1.6 Casts 148
4.1.7 Anonymous Subclasses 149
4.1.8 Method Expressions with `super` 150
4.2 Inheritance Hierarchies 150
4.2.1 Final Methods and Classes 150
4.2.2 Abstract Methods and Classes 151
4.2.3 Protected Access 152
4.2.4 Sealed Types 153
4.2.5 Inheritance and Default Methods 157
4.3 `Object`: The Cosmic Superclass 157
4.3.1 The `toString` Method 158
4.3.2 The `equals` Method 159
4.3.3 The `hashCode` Method 162
4.3.4 Cloning Objects 163
4.4 Enumerations 166
4.4.1 Methods of Enumerations 166
4.4.2 Constructors, Methods, and Fields 168
4.4.3 Bodies of Instances 168
4.4.4 Static Members 169
4.4.5 Switching on an Enumeration 170
4.5 Runtime Type Information and Resources 170
4.5.1 The Class `Class` 170
4.5.2 Loading Resources 174
4.5.3 Class Loaders 174
4.5.4 The Context Class Loader 176
4.5.5 Service Loaders 177

4.6 Reflection 179
4.6.1 Enumerating Class Members 179
4.6.2 Inspecting Objects 180
4.6.3 Invoking Methods 182
4.6.4 Constructing Objects 182
4.6.5 JavaBeans 183
4.6.6 Working with Arrays 185
4.6.7 Proxies 186
Exercises 188

5 EXCEPTIONS, ASSERTIONS, AND LOGGING 191
5.1 Exception Handling 192
5.1.1 Throwing Exceptions 192
5.1.2 The Exception Hierarchy 193
5.1.3 Declaring Checked Exceptions 195
5.1.4 Catching Exceptions 196
5.1.5 The Try-with-Resources Statement 197
5.1.6 The `finally` Clause 199
5.1.7 Rethrowing and Chaining Exceptions 201
5.1.8 Uncaught Exceptions and the Stack Trace 202
5.1.9 API Methods for Throwing Exceptions 203
5.2 Assertions 204
5.2.1 Using Assertions 204
5.2.2 Enabling and Disabling Assertions 205
5.3 Logging 206
5.3.1 Should You Use the Java Logging Framework? 206
5.3.2 Logging 101 207
5.3.3 The Platform Logging API 208
5.3.4 Logging Configuration 209
5.3.5 Log Handlers 211
5.3.6 Filters and Formatters 213
Exercises 214

6 GENERIC PROGRAMMING 219
6.1 Generic Classes 220
6.2 Generic Methods 221
6.3 Type Bounds 222
6.4 Type Variance and Wildcards 223
6.4.1 Subtype Wildcards 224
6.4.2 Supertype Wildcards 225
6.4.3 Wildcards with Type Variables 226
6.4.4 Unbounded Wildcards 227
6.4.5 Wildcard Capture 228
6.5 Generics in the Java Virtual Machine 228
6.5.1 Type Erasure 229
6.5.2 Cast Insertion 229
6.5.3 Bridge Methods 230
6.6 Restrictions on Generics 231
6.6.1 No Primitive Type Arguments 231
6.6.2 At Runtime, All Types Are Raw 232
6.6.3 You Cannot Instantiate Type Variables 233
6.6.4 You Cannot Construct Arrays of Parameterized Types 235
6.6.5 Class Type Variables Are Not Valid in Static Contexts 236
6.6.6 Methods May Not Clash after Erasure 236
6.6.7 Exceptions and Generics 237
6.7 Reflection and Generics 238
6.7.1 The `Class<T>` Class 239
6.7.2 Generic Type Information in the Virtual Machine 239
Exercises 241
7 COLLECTIONS 247
7.1 An Overview of the Collections Framework 248
7.2 Iterators 252
7.3 Sets 254

7.4 Maps 255
7.5 Other Collections 259
7.5.1 Properties 259
7.5.2 Bit Sets 260
7.5.3 Enumeration Sets and Maps 262
7.5.4 Stacks, Queues, Deques, and Priority Queues 262
7.5.5 Weak Hash Maps 263
7.6 Views 264
7.6.1 Small Collections 264
7.6.2 Ranges 265
7.6.3 Unmodifiable Views 266
Exercises 267

8 STREAMS 271
8.1 From Iterating to Stream Operations 272
8.2 Stream Creation 273
8.3 The `filter`, `map`, and `flatMap` Methods 276
8.4 Extracting Substreams and Combining Streams 278
8.5 Other Stream Transformations 279
8.6 Simple Reductions 280
8.7 The Optional Type 281
8.7.1 Producing an Alternative 281
8.7.2 Consuming the Value If Present 281
8.7.3 Pipelining Optional Values 282
8.7.4 How Not to Work with Optional Values 282
8.7.5 Creating Optional Values 284
8.7.6 Composing Optional Value Functions with `flatMap` 284
8.7.7 Turning an Optional into a Stream 285
8.8 Collecting Results 286
8.9 Collecting into Maps 287
8.10 Grouping and Partitioning 289
8.11 Downstream Collectors 289
8.12 Reduction Operations 292
8.13 Primitive Type Streams 294

8.14 Parallel Streams 295
Exercises 298

9 PROCESSING INPUT AND OUTPUT 301
9.1 Input/Output Streams, Readers, and Writers 302
9.1.1 Obtaining Streams 302
9.1.2 Reading Bytes 303
9.1.3 Writing Bytes 304
9.1.4 Character Encodings 305
9.1.5 Text Input 307
9.1.6 Text Output 308
9.1.7 Reading and Writing Binary Data 310
9.1.8 Random-Access Files 310
9.1.9 Memory-Mapped Files 311
9.1.10 File Locking 312
9.2 Paths, Files, and Directories 312
9.2.1 Paths 312
9.2.2 Creating Files and Directories 314
9.2.3 Copying, Moving, and Deleting Files 315
9.2.4 Visiting Directory Entries 316
9.2.5 ZIP File Systems 319
9.3 HTTP Connections 320
9.3.1 The `URLConnection` and `HttpURLConnection` Classes 320
9.3.2 The HTTP Client API 321
9.4 Regular Expressions 323
9.4.1 The Regular Expression Syntax 324
9.4.2 Testing a Match 329
9.4.3 Finding All Matches 329
9.4.4 Groups 330
9.4.5 Splitting along Delimiters 331
9.4.6 Replacing Matches 332
9.4.7 Flags 333
9.5 Serialization 333
9.5.1 The `Serializable` Interface 334

9.5.2 Transient Instance Variables 336
9.5.3 The `readObject` and `writeObject` Methods 336
9.5.4 The `readExternal` and `writeExternal` Methods 338
9.5.5 The `readResolve` and `writeReplace` Methods 339
9.5.6 Versioning 340
9.5.7 Deserialization and Security 342
Exercises 344

10 CONCURRENT PROGRAMMING 347
10.1 Concurrent Tasks 348
10.1.1 Running Tasks 348
10.1.2 Futures 351
10.2 Asynchronous Computations 353
10.2.1 Completable Futures 353
10.2.2 Composing Completable Futures 355
10.2.3 Long-Running Tasks in User-Interface Callbacks 358
10.3 Thread Safety 360
10.3.1 Visibility 360
10.3.2 Race Conditions 362
10.3.3 Strategies for Safe Concurrency 364
10.3.4 Immutable Classes 365
10.4 Parallel Algorithms 366
10.4.1 Parallel Streams 366
10.4.2 Parallel Array Operations 367
10.5 Threadsafe Data Structures 368
10.5.1 Concurrent Hash Maps 369
10.5.2 Blocking Queues 370
10.5.3 Other Threadsafe Data Structures 372
10.6 Atomic Counters and Accumulators 373
10.7 Locks and Conditions 375
10.7.1 Locks 375
10.7.2 The `synchronized` Keyword 377
10.7.3 Waiting on Conditions 379
10.8 Threads 381

10.8.1 Starting a Thread 381
10.8.2 Thread Interruption 382
10.8.3 Thread-Local Variables 384
10.8.4 Miscellaneous Thread Properties 385
10.9 Processes 386
10.9.1 Building a Process 386
10.9.2 Running a Process 388
10.9.3 Process Handles 389
Exercises 390

11 ANNOTATIONS 397
11.1 Using Annotations 398
11.1.1 Annotation Elements 398
11.1.2 Multiple and Repeated Annotations 400
11.1.3 Annotating Declarations 400
11.1.4 Annotating Type Uses 401
11.1.5 Making Receivers Explicit 402
11.2 Defining Annotations 403
11.3 Standard Annotations 406
11.3.1 Annotations for Compilation 407
11.3.2 Meta-Annotations 408
11.4 Processing Annotations at Runtime 410
11.5 Source-Level Annotation Processing 413
11.5.1 Annotation Processors 413
11.5.2 The Language Model API 414
11.5.3 Using Annotations to Generate Source Code 415
Exercises 417

12 THE DATE AND TIME API 421
12.1 The Time Line 422
12.2 Local Dates 424
12.3 Date Adjusters 428
12.4 Local Time 429
12.5 Zoned Time 430

12.6 Formatting and Parsing 433
12.7 Interoperating with Legacy Code 436
Exercises 437

13 INTERNATIONALIZATION 441
13.1 Locales 442
13.1.1 Specifying a Locale 443
13.1.2 The Default Locale 445
13.1.3 Display Names 446
13.2 Number Formats 447
13.3 Currencies 448
13.4 Date and Time Formatting 449
13.5 Collation and Normalization 451
13.6 Message Formatting 453
13.7 Resource Bundles 455
13.7.1 Organizing Resource Bundles 455
13.7.2 Bundle Classes 457
13.8 Character Encodings 458
13.9 Preferences 459
Exercises 461

14 COMPILING AND SCRIPTING 463
14.1 The Compiler API 463
14.1.1 Invoking the Compiler 464
14.1.2 Launching a Compilation Task 464
14.1.3 Capturing Diagnostics 465
14.1.4 Reading Source Files from Memory 465
14.1.5 Writing Byte Codes to Memory 466
14.2 The Scripting API 467
14.2.1 Getting a Scripting Engine 468
14.2.2 Evaluating Scripts 468
14.2.3 Bindings 469
14.2.4 Redirecting Input and Output 469
14.2.5 Calling Scripting Functions and Methods 470

14.2.6 Compiling a Script 471
Exercises 472

15 THE JAVA PLATFORM MODULE SYSTEM 475
15.1 The Module Concept 476
15.2 Naming Modules 478
15.3 The Modular “Hello, World!” Program 478
15.4 Requiring Modules 480
15.5 Exporting Packages 482
15.6 Modules and Reflective Access 485
15.7 Modular JARs 488
15.8 Automatic Modules 489
15.9 The Unnamed Module 491
15.10 Command-Line Flags for Migration 491
15.11 Transitive and Static Requirements 493
15.12 Qualified Exporting and Opening 495
15.13 Service Loading 496
15.14 Tools for Working with Modules 497
Exercises 499

Index 501

Preface

Java has seen many changes since its initial release in 1996. The classic book, *Core Java*, covers, in meticulous detail, not just the language but all core libraries and a multitude of changes between versions, spanning two volumes and over 2,000 pages. However, if you just want to be productive with modern Java, there is a much faster, easier pathway for learning the language and core libraries. In this book, I don't retrace history and don't dwell on features of past versions. I show you the good parts of Java as it exists today, so you can put your knowledge to work quickly.

As with my previous "Impatient" books, I quickly cut to the chase, showing you what you need to know to solve a programming problem without lecturing about the superiority of one paradigm over another. I also present the information in small chunks, organized so that you can quickly retrieve it when needed.

Assuming you are proficient in some other programming language, such as C++, JavaScript, Swift, PHP, or Ruby, with this book you will learn how to become a competent Java programmer. I cover all aspects of Java that a developer needs to know today, including the powerful concepts of lambda expressions and streams, as well as modern constructs such as records and sealed classes.

A key reason to use Java is to tackle concurrent programming. With parallel algorithms and threadsafe data structures readily available in the Java library, the way application programmers should handle concurrent programming has completely changed. I provide fresh coverage, showing you how to use the powerful library features instead of error-prone low-level constructs.

Traditionally, books on Java have focused on user interface programming, but nowadays, few developers produce user interfaces on desktop computers. If you intend to use Java for server-side programming or Android programming, you will be able to use this book effectively without being distracted by desktop GUI code.

Finally, this book is written for application programmers, not for a college course and not for systems wizards. The book covers issues that application programmers need to wrestle with, such as logging and working with files, but you won't learn how to implement a linked list by hand or how to write a web server.

I hope you enjoy this rapid-fire introduction into modern Java, and I hope it will make your work with Java productive and enjoyable.

If you find errors or have suggestions for improvement, please visit `http://horstmann.com/javaimpatient`, head for the errata page, and leave a comment. Be sure to visit that site to download the runnable code examples that complement this book.

Register your copy of *Core Java for the Impatient, Third 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` and log in or create an account. Enter the product ISBN (9780138052102) 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

My thanks go, as always, to my editor Greg Doench, who enthusiastically supported the vision of a short book that gives a fresh introduction to Java. Dmitry Kirsanov and Alina Kirsanova once again turned an XHTML manuscript into an attractive book with amazing speed and attention to detail. My special gratitude goes to the excellent team of reviewers for all editions who spotted many errors and gave thoughtful suggestions for improvement. They are: Andres Almiray, Gail Anderson, Paul Anderson, Marcus Biel, Brian Goetz, Mark Lawrence, Doug Lea, Simon Ritter, Yoshiki Shibata, and Christian Ullenboom.

Cay Horstmann
Berlin
August 2022

This page intentionally left blank

About the Author

Cay S. Horstmann is the author of *JavaScript for the Impatient* and *Scala for the Impatient* (both from Addison-Wesley), is principal author of *Core Java, Volumes I and II, Twelfth Edition* (Pearson, 2022), and has written a dozen other books for professional programmers and computer science students. He is professor emeritus of computer science at San Jose State University and is a Java Champion.

Fundamental Programming Structures

Topics in This Chapter

- 1.1 Our First Program — page 2
- 1.2 Primitive Types — page 11
- 1.3 Variables — page 14
- 1.4 Arithmetic Operations — page 17
- 1.5 Strings — page 25
- 1.6 Input and Output — page 35
- 1.7 Control Flow — page 38
- 1.8 Arrays and Array Lists — page 46
- 1.9 Functional Decomposition — page 56
- Exercises — page 58

Processing Input and Output

Topics in This Chapter

- 9.1 Input/Output Streams, Readers, and Writers — page 302
- 9.2 Paths, Files, and Directories — page 312
- 9.3 HTTP Connections — page 320
- 9.4 Regular Expressions — page 323
- 9.5 Serialization — page 333
- Exercises — page 344

Chapter 9

In this chapter, you will learn how to work with files, directories, and web pages, and how to read and write data in binary and text format. You will also find a discussion of regular expressions, which can be useful for processing input. (I couldn't think of a better place to handle that topic, and apparently neither could the Java developers—when the regular expression API specification was proposed, it was attached to the specification request for "new I/O" features.) Finally, this chapter shows you the object serialization mechanism that lets you store objects as easily as you can store text or numeric data.

The key points of this chapter are:

1. An `InputStream` is a source of bytes, and an `OutputStream` is a destination for bytes.
2. A `Reader` reads characters, and a `Writer` writes them. Be sure to specify a character encoding.
3. The `Files` class has convenience methods for reading all bytes or lines of a file.
4. The `DataInput` and `DataOutput` interfaces have methods for writing numbers in binary format.
5. Use a `RandomAccessFile` or a memory-mapped file for random access.

6. A `Path` is an absolute or relative sequence of path components in a file system. Paths can be combined (or "resolved").
7. Use the methods of the `Files` class to copy, move, or delete files and to recursively walk through a directory tree.
8. To read or update a ZIP file, use a ZIP file system.
9. You can read the contents of a web page with the `URL` class. To read metadata or write data, use the `URLConnection` class.
10. With the `Pattern` and `Matcher` classes, you can find all matches of a regular expression in a string, as well as the captured groups for each match.
11. The serialization mechanism can save and restore any object implementing the `Serializable` interface, provided its instance variables are also serializable.

9.1 Input/Output Streams, Readers, and Writers

In the Java API, a source from which one can read bytes is called an *input stream*. The bytes can come from a file, a network connection, or an array in memory. (These streams are unrelated to the streams of Chapter 8.) Similarly, a destination for bytes is an *output stream*. In contrast, *readers* and *writers* consume and produce sequences of *characters*. In the following sections, you will learn how to read and write bytes and characters.

9.1.1 Obtaining Streams

The easiest way to obtain a stream from a file is with the static methods

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
```

Here, `path` is an instance of the `Path` class that is covered in Section 9.2.1, "Paths" (page 312). It describes a path in a file system.

If you have a URL, you can read its contents from the input stream returned by the `openStream` method of the `URL` class:

```
var url = new URL("https://horstmann.com/index.html");
InputStream in = url.openStream();
```

Section 9.3, "HTTP Connections" (page 320) shows how to send data to a web server.

The `ByteArrayInputStream` class lets you read from an array of bytes.

```
byte[] bytes = ...;
var in = new ByteArrayInputStream(bytes);
Read from in
```

Conversely, to send output to a byte array, use a `ByteArrayOutputStream`:

```
var out = new ByteArrayOutputStream();
Write to out
byte[] bytes = out.toByteArray();
```

9.1.2 Reading Bytes

The `InputStream` class has a method to read a single byte:

```
InputStream in = ...;
int b = in.read();
```

This method either returns the byte as an integer between `0` and `255`, or returns `-1` if the end of input has been reached.

CAUTION: The Java `byte` type has values between `-128` and `127`. You can cast the returned value into a `byte` *after* you have checked that it is not `-1`.

More commonly, you will want to read the bytes in bulk. The most convenient method is the `readAllBytes` method that simply reads all bytes from the stream into a byte array:

```
byte[] bytes = in.readAllBytes();
```

TIP: If you want to read all bytes from a file, call the convenience method

```
byte[] bytes = Files.readAllBytes(path);
```

If you want to read some, but not all bytes, provide a byte array and call the `readNBytes` method:

```
var bytes = new byte[len];
int bytesRead = in.readNBytes(bytes, offset, n);
```

The method reads until either `n` bytes are read or no further input is available, and returns the actual number of bytes read. If no input was available at all, the methods return `-1`.

NOTE: There is also a `read(byte[], int, int)` method whose description seems exactly like `readNBytes`. The difference is that the `read` method only attempts to read the bytes and returns immediately with a lower count if it fails. The `readNBytes` method keeps calling `read` until all requested bytes have been obtained or `read` returns `-1`.

Finally, you can skip bytes:

```
long bytesToSkip = ...;
in.skipNBytes(bytesToSkip);
```

9.1.3 Writing Bytes

The `write` methods of an `OutputStream` can write individual bytes and byte arrays.

```
OutputStream out = ...;
int b = ...;
out.write(b);
byte[] bytes = ...;
out.write(bytes);
out.write(bytes, start, length);
```

When you are done writing a stream, you must *close* it in order to commit any buffered output. This is best done with a try-with-resources statement:

```
try (OutputStream out = ...) {
    out.write(bytes);
}
```

If you need to copy an input stream to an output stream, use the `InputStream.transferTo` method:

```
try (InputStream in = ...; OutputStream out = ...) {
    in.transferTo(out);
}
```

Both streams need to be closed after the call to `transferTo`. It is best to use a try-with-resources statement, as in the code example.

To write a file to an `OutputStream`, call

```
Files.copy(path, out);
```

Conversely, to save an `InputStream` to a file, call

```
Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
```

9.1.4 Character Encodings

Input and output streams are for sequences of bytes, but in many cases you will work with text—that, is, sequences of characters. It then matters how characters are encoded into bytes.

Java uses the Unicode standard for characters. Each character or "code point" has a 21-bit integer number. There are different *character encodings*—methods for packaging those 21-bit numbers into bytes.

The most common encoding is UTF-8, which encodes each Unicode code point into a sequence of one to four bytes (see Table 9-1). UTF-8 has the advantage that the characters of the traditional ASCII character set, which contains all characters used in English, only take up one byte each.

Table 9–1 UTF-8 Encoding

Character range	Encoding
`0...7F`	$0a_6a_5a_4a_3a_2a_1a_0$
`80...7FF`	$110a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$
`800...FFFF`	$1110a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$
`10000...10FFFF`	$11110a_{20}a_{19}a_{18}$ $10a_{17}a_{16}a_{15}a_{14}a_{13}a_{12}$ $10a_{11}a_{10}a_9a_8a_7a_6$ $10a_5a_4a_3a_2a_1a_0$

Another common encoding is UTF-16, which encodes each Unicode code point into one or two 16-bit values (see Table 9-2). This is the encoding used in Java strings. Actually, there are two forms of UTF-16, called "big-endian" and "little-endian." Consider the 16-bit value `0x2122`. In big-endian format, the more significant byte comes first: `0x21` followed by `0x22`. In little-endian format, it is the other way around: `0x22 0x21`. To indicate which of the two is used, a file can start with the "byte order mark," the 16-bit quantity `0xFEFF`. A reader can use this value to determine the byte order and discard it.

Table 9–2 UTF-16 Encoding

Character range	Encoding
`0...FFFF`	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$
`10000...10FFFF`	$110110b_{19}b_{18}b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}$ $110111a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$ where $b_{19}b_{18}b_{17}b_{16} = a_{20}a_{19}a_{18}a_{17}a_{16} - 1$

CAUTION: Some programs, including Microsoft Notepad, add a byte order mark at the beginning of UTF-8 encoded files. Clearly, this is unnecessary since there are no byte ordering issues in UTF-8. But the Unicode standard allows it, and even suggests that it's a pretty good idea since it leaves little doubt about the encoding. It is supposed to be removed when reading a UTF-8 encoded file. Sadly, Java does not do that, and bug reports against this issue are closed as "will not fix." Your best bet is to strip out any leading `\uFEFF` that you find in your input.

In addition to the UTF encodings, there are partial encodings that cover a character range suitable for a given user population. For example, ISO 8859-1 is a one-byte code that includes accented characters used in Western European languages. Shift_JIS is a variable-length code for Japanese characters. A large number of these encodings are still in widespread use.

There is no reliable way to automatically detect the character encoding from a stream of bytes. Some API methods let you use the "default charset"—the character encoding that is preferred by the operating system of the computer. Is that the same encoding that is used by your source of bytes? These bytes may well originate from a different part of the world. Therefore, you should always explicitly specify the encoding. For example, when reading a web page, check the `Content-Type` header.

NOTE: The platform encoding is returned by the static method `Charset.defaultCharset`. The static method `Charset.availableCharsets` returns all available `Charset` instances, as a map from canonical names to `Charset` objects.

CAUTION: The Oracle implementation has a system property `file.encoding` for overriding the platform default. This is not an officially supported property, and it is not consistently followed by all parts of Oracle's implementation of the Java library. You should not set it.

The `StandardCharsets` class has static variables of type `Charset` for the character encodings that every Java virtual machine must support:

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

To obtain the `Charset` for another encoding, use the static `forName` method:

```
Charset shiftJIS = Charset.forName("Shift_JIS");
```

Use the `Charset` object when reading or writing text. For example, you can turn an array of bytes into a string as

```
var contents = new String(bytes, StandardCharsets.UTF_8);
```

TIP: Some methods allow you to specify a character encoding with a `Charset` object or a string. Choose the `StandardCharsets` constants, so you don't have to worry about the correct spelling. For example, `new String(bytes, "UTF 8")` is not acceptable and will cause a runtime error.

CAUTION: Some methods (such as the `String(byte[])` constructor) use the default platform encoding if you don't specify any; others (such as `Files.readAllLines`) use UTF-8.

9.1.5 Text Input

To read text input, use a `Reader`. You can obtain a `Reader` from any input stream with the `InputStreamReader` adapter:

```
InputStream inStream = ...;
var in = new InputStreamReader(inStream, charset);
```

If you want to process the input one UTF-16 code unit at a time, you can call the `read` method:

```
int ch = in.read();
```

The method returns a code unit between `0` and `65536`, or `-1` at the end of input.

That is not very convenient. Here are several alternatives.

With a short text file, you can read it into a string like this:

```
String content = Files.readString(path, charset);
```

But if you want the file as a sequence of lines, call

```
List<String> lines = Files.readAllLines(path, charset);
```

If the file is large, process them lazily as a `Stream<String>`:

```
try (Stream<String> lines = Files.lines(path, charset)) {
    ...
}
```

NOTE: If an `IOException` occurs as the stream fetches the lines, that exception is wrapped into an `UncheckedIOException` which is thrown out of the stream operation. (This subterfuge is necessary because stream operations are not declared to throw any checked exceptions.)

To read numbers or words from a file, use a `Scanner`, as you have seen in Chapter 1. For example,

```
var in = new Scanner(path, StandardCharsets.UTF_8);
while (in.hasNextDouble()) {
    double value = in.nextDouble();
    ...
}
```

TIP: To read alphabetic words, set the scanner's delimiter to a regular expression that is the complement of what you want to accept as a token. For example, after calling

```
in.useDelimiter("\\PL+");
```

the scanner reads in letters since any sequence of nonletters is a delimiter. See Section 9.4.1, "The Regular Expression Syntax" (page 324) for the regular expression syntax.

You can then obtain a stream of all words as

```
Stream<String> words = in.tokens();
```

If your input does not come from a file, wrap the `InputStream` into a `BufferedReader`:

```
try (var reader = new BufferedReader(new InputStreamReader(url.openStream()))) {
    Stream<String> lines = reader.lines();
    ...
}
```

A `BufferedReader` reads input in chunks for efficiency. (Oddly, this is not an option for basic readers.) It has methods `readLine` to read a single line and `lines` to yield a stream of lines.

If a method asks for a `Reader` and you want it to read from a file, call `Files.newBufferedReader(path, charset)`.

9.1.6 Text Output

To write text, use a `Writer`. With the `write` method, you can write strings. You can turn any output stream into a `Writer`:

```
OutputStream outStream = ...;
var out = new OutputStreamWriter(outStream, charset);
out.write(str);
```

To get a writer for a file, use

```
Writer out = Files.newBufferedWriter(path, charset);
```

It is more convenient to use a `PrintWriter`, which has the `print`, `println`, and `printf` that you have always used with `System.out`. Using those methods, you can print numbers and use formatted output.

If you write to a file, construct a `PrintWriter` like this:

```
var out = new PrintWriter(Files.newBufferedWriter(path, charset));
```

If you write to another stream, use

```
var out = new PrintWriter(new OutputStreamWriter(outStream, charset));
```

NOTE: `System.out` is an instance of `PrintStream`, not `PrintWriter`. This is a relic from the earliest days of Java. However, the `print`, `println`, and `printf` methods work the same way for the `PrintStream` and `PrintWriter` classes, using a character encoding for turning characters into bytes.

If you already have the text to write in a string, call

```
String content = ...;
Files.write(path, content.getBytes(charset));
```

or

```
Files.write(path, lines, charset);
```

Here, `lines` can be a `Collection<String>`, or even more generally, an `Iterable<? extends CharSequence>`.

To append to a file, use

```
Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);
Files.write(path, lines, charset, StandardOpenOption.APPEND);
```

CAUTION: When writing text with a partial character set such as ISO 8859-1, any unmappable characters are silently changed to a “replacement”—in most cases, either the `?` character or the Unicode replacement character `U+FFFD`.

Sometimes, a library method wants a `Writer` to write output. If you want to capture that output in a string, hand it a `StringWriter`. Or, if it wants a `PrintWriter`, wrap the `StringWriter` like this:

```
var writer = new StringWriter();
throwable.printStackTrace(new PrintWriter(writer));
String stackTrace = writer.toString();
```

9.1.7 Reading and Writing Binary Data

The `DataInput` interface declares the following methods for reading a number, a character, a `boolean` value, or a string in binary format:

```
byte readByte()
int readUnsignedByte()
char readChar()
short readShort()
int readUnsignedShort()
int readInt()
long readLong()
float readFloat()
double readDouble()
void readFully(byte[] b)
```

The `DataOutput` interface declares corresponding `write` methods.

NOTE: These methods read and write numbers in big-endian format.

CAUTION: There are also `readUTF`/`writeUTF` methods that use a "modified UTF-8" format. These methods are *not* compatible with regular UTF-8, and are only useful for JVM internals.

The advantage of binary I/O is that it is fixed width and efficient. For example, `writeInt` always writes an integer as a big-endian 4-byte binary quantity regardless of the number of digits. The space needed is the same for each value of a given type, which speeds up random access. Also, reading binary data is faster than parsing text. The main drawback is that the resulting files cannot be easily inspected in a text editor.

You can use the `DataInputStream` and `DataOutputStream` adapters with any stream. For example,

```
DataInput in = new DataInputStream(Files.newInputStream(path));
DataOutput out = new DataOutputStream(Files.newOutputStream(path));
```

9.1.8 Random-Access Files

The `RandomAccessFile` class lets you read or write data anywhere in a file. You can open a random-access file either for reading only or for both reading and writing; specify the option by using the string `"r"` (for read access) or `"rw"` (for read/write access) as the second argument in the constructor. For example,

```
var file = new RandomAccessFile(path.toString(), "rw");
```

A random-access file has a *file pointer* that indicates the position of the next byte to be read or written. The `seek` method sets the file pointer to an arbitrary byte position within the file. The argument to `seek` is a long integer between zero and the length of the file (which you can obtain with the `length` method). The `getFilePointer` method returns the current position of the file pointer.

The `RandomAccessFile` class implements both the `DataInput` and `DataOutput` interfaces. To read and write numbers from a random-access file, use methods such as `readInt`/`writeInt` that you saw in the preceding section. For example,

```
int value = file.readInt();
file.seek(file.getFilePointer() - 4);
file.writeInt(value + 1);
```

9.1.9 Memory-Mapped Files

Memory-mapped files provide another, very efficient approach for random access that works well for very large files. However, the API for data access is completely different from that of input/output streams. First, get a *channel* to the file:

```
FileChannel channel = FileChannel.open(path,
    StandardOpenOption.READ, StandardOpenOption.WRITE)
```

Then, map an area of the file (or, if it is not too large, the entire file) into memory:

```
ByteBuffer buffer = channel.map(FileChannel.MapMode.READ_WRITE,
    0, channel.size());
```

Use methods `get`, `getInt`, `getDouble`, and so on to read values, and the equivalent `put` methods to write values.

```
int offset = ...;
int value = buffer.getInt(offset);
buffer.put(offset, value + 1);
```

At some point, and certainly when the channel is closed, these changes are written back to the file.

NOTE: By default, the methods for reading and writing numbers use big-endian byte order. You can change the byte order with the command

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

9.1.10 File Locking

When multiple simultaneously executing programs modify the same file, they need to communicate in some way, or the file can easily become damaged. File locks can solve this problem.

Suppose your application saves a configuration file with user preferences. If a user invokes two instances of the application, it could happen that both of them want to write the configuration file at the same time. In that situation, the first instance should lock the file. When the second instance finds the file locked, it can decide to wait until the file is unlocked or simply skip the writing process. To lock a file, call either the `lock` or `tryLock` methods of the `FileChannel` class.

```
FileChannel channel = FileChannel.open(path, StandardOpenOption.WRITE);
FileLock lock = channel.lock();
```

or

```
FileLock lock = channel.tryLock();
```

The first call blocks until the lock becomes available. The second call returns immediately, either with the lock or with `null` if the lock is not available. The file remains locked until the lock or the channel is closed. It is best to use a try-with-resources statement:

```
try (FileLock lock = channel.lock()) {
    ...
}
```

9.2 Paths, Files, and Directories

You have already seen `Path` objects for specifying file paths. In the following sections, you will see how to manipulate these objects and how to work with files and directories.

9.2.1 Paths

A `Path` is a sequence of directory names, optionally followed by a file name. The first component of a path may be a root component, such as / or `C:\`. The permissible root components depend on the file system. A path that starts with a root component is *absolute*. Otherwise, it is *relative*. For example, here we construct an absolute and a relative path. For the absolute path, we assume we are running on a Unix-like file system.

```
Path absolute = Path.of("/", "home", "cay");
Path relative = Path.of("myapp", "conf", "user.properties");
```

The static `Path.of` method receives one or more strings, which it joins with the path separator of the default file system (/ for a Unix-like file system, \ for Windows). It then parses the result, throwing an `InvalidPathException` if the result is not a valid path in the given file system. The result is a `Path` object.

You can also provide a string with separators to the `Path.of` method:

```
Path homeDirectory = Path.of("/home/cay");
```

NOTE: A `Path` object does not have to correspond to a file that actually exists. It is merely an abstract sequence of names. To create a file, first make a path, then call a method to create the corresponding file—see Section 9.2.2, "Creating Files and Directories" (page 314).

It is very common to combine or "resolve" paths. The call `p.resolve(q)` returns a path according to these rules:

- If `q` is absolute, then the result is `q`.
- Otherwise, the result is "`p` then `q`," according to the rules of the file system.

For example, suppose your application needs to find its configuration file relative to the home directory. Here is how you can combine the paths:

```
Path workPath = homeDirectory.resolve("myapp/work");
    // Same as homeDirectory.resolve(Path.of("myapp/work"));
```

There is a convenience method `resolveSibling` that resolves against a path's parent, yielding a sibling path. For example, if `workPath` is `/home/cay/myapp/work`, the call

```
Path tempPath = workPath.resolveSibling("temp");
```

yields `/home/cay/myapp/temp`.

The opposite of `resolve` is `relativize`. The call `p.relativize(r)` yields the path `q` which, when resolved with `p`, yields `r`. For example,

```
Path.of("/home/cay").relativize(Path.of("/home/fred/myapp"))
```

yields `../fred/myapp`, assuming we have a file system that uses .. to denote the parent directory.

The `normalize` method removes any redundant . and .. components (or whatever the file system may deem redundant). For example, normalizing the path `/home/cay/../fred/./myapp` yields `/home/fred/myapp`.

The `toAbsolutePath` method yields the absolute path of a given path. If the path is not already absolute, it is resolved against the "user directory"—that is, the directory from which the JVM was invoked. For example, if you launched a program from `/home/cay/myapp`, then `Path.of("config").toAbsolutePath()` returns `/home/cay/myapp/config`.

The `Path` interface has methods for taking paths apart and combining them with other paths. This code sample shows some of the most useful ones:

```
Path p = Path.of("/home", "cay", "myapp.properties");
Path parent = p.getParent(); // The path /home/cay
Path file = p.getFileName(); // The last element, myapp.properties
Path root = p.getRoot(); // The initial segment / (null for a relative path)
Path first = p.getName(0); // The first element
Path dir = p.subpath(1, p.getNameCount());
    // All but the first element, cay/myapp.properties
```

The `Path` interface extends the `Iterable<Path>` element, so you can iterate over the name components of a `Path` with an enhanced `for` loop:

```
for (Path component : path) {
    ...
}
```

NOTE: Occasionally, you may need to interoperate with legacy APIs that use the `File` class instead of the `Path` interface. The `Path` interface has a `toFile` method, and the `File` class has a `toPath` method.

9.2.2 Creating Files and Directories

To create a new directory, call

```
Files.createDirectory(path);
```

All but the last component in the path must already exist. To create intermediate directories as well, use

```
Files.createDirectories(path);
```

You can create an empty file with

```
Files.createFile(path);
```

The call throws an exception if the file already exists. The checks for existence and the creation are atomic. If the file doesn't exist, it is created before anyone else has a chance to do the same.

The call `Files.exists(path)` checks whether the given file or directory exists. To test whether it is a directory or a "regular" file (that is, with data in it, not something like a directory or symbolic link), call the static methods `isDirectory` and `isRegularFile` of the `Files` class.

There are convenience methods for creating a temporary file or directory in a given or system-specific location.

```
Path tempFile = Files.createTempFile(dir, prefix, suffix);
Path tempFile = Files.createTempFile(prefix, suffix);
Path tempDir = Files.createTempDirectory(dir, prefix);
Path tempDir = Files.createTempDirectory(prefix);
```

Here, `dir` is a `Path`, and `prefix`/`suffix` are strings which may be null. For example, the call `Files.createTempFile(null, ".txt")` might return a path such as `/tmp/1234405522364837194.txt`.

9.2.3 Copying, Moving, and Deleting Files

To copy a file from one location to another, simply call

```
Files.copy(fromPath, toPath);
```

To move a file (that is, copy and delete the original), call

```
Files.move(fromPath, toPath);
```

You can also use this command to move an empty directory.

The copy or move will fail if the target exists. If you want to overwrite an existing target, use the `REPLACE_EXISTING` option. If you want to copy all file attributes, use the `COPY_ATTRIBUTES` option. You can supply both like this:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.COPY_ATTRIBUTES);
```

You can specify that a move should be atomic. Then you are assured that either the move completed successfully, or the source continues to be present. Use the `ATOMIC_MOVE` option:

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

See Table 9-3 for a summary of the options that are available for file operations.

Finally, to delete a file, simply call

```
Files.delete(path);
```

This method throws an exception if the file doesn't exist, so instead you may want to use

```
boolean deleted = Files.deleteIfExists(path);
```

The deletion methods can also be used to remove an empty directory.

Table 9-3 Standard Options for File Operations

Option	Description
StandardOpenOption; use with newBufferedWriter, newInputStream, newOutputStream, write	
READ	Open for reading.
WRITE	Open for writing.
APPEND	If opened for writing, append to the end of the file.
TRUNCATE_EXISTING	If opened for writing, remove existing contents.
CREATE_NEW	Create a new file and fail if it exists.
CREATE	Atomically create a new file if it doesn't exist.
DELETE_ON_CLOSE	Make a "best effort" to delete the file when it is closed.
SPARSE	A hint to the file system that this file will be sparse.
DSYNC|SYNC	Requires that each update to the file data|data and metadata be written synchronously to the storage device.
StandardCopyOption; use with copy, move	
ATOMIC_MOVE	Move the file atomically.
COPY_ATTRIBUTES	Copy the file attributes.
REPLACE_EXISTING	Replace the target if it exists.
LinkOption; use with all of the above methods and exists, isDirectory, isRegularFile	
NOFOLLOW_LINKS	Do not follow symbolic links.
FileVisitOption; use with find, walk, walkFileTree	
FOLLOW_LINKS	Follow symbolic links.

9.2.4 Visiting Directory Entries

The static `Files.list` method returns a `Stream<Path>` that reads the entries of a directory. The directory is read lazily, making it possible to efficiently process directories with huge numbers of entries.

Since reading a directory involves a system resource that needs to be closed, you should use a try-with-resources block:

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {
    ...
}
```

The `list` method does not enter subdirectories. To process all descendants of a directory, use the `Files.walk` method instead.

```
try (Stream<Path> entries = Files.walk(pathToRoot)) {
    // Contains all descendants, visited in depth-first order
}
```

Here is a sample traversal of the unzipped `src.zip` tree:

```
java
java/nio
java/nio/DirectCharBufferU.java
java/nio/ByteBufferAsShortBufferRL.java
java/nio/MappedByteBuffer.java
...
java/nio/ByteBufferAsDoubleBufferB.java
java/nio/charset
java/nio/charset/CoderMalfunctionError.java
java/nio/charset/CharsetDecoder.java
java/nio/charset/UnsupportedCharsetException.java
java/nio/charset/spi
java/nio/charset/spi/CharsetProvider.java
java/nio/charset/StandardCharsets.java
java/nio/charset/Charset.java
...
java/nio/charset/CoderResult.java
java/nio/HeapFloatBufferR.java
...
```

As you can see, whenever the traversal yields a directory, it is entered before continuing with its siblings.

You can limit the depth of the tree that you want to visit by calling `Files.walk(pathToRoot, depth)`. Both `walk` methods have a varargs parameter of type `FileVisitOption...`, but there is only one option you can supply: `FOLLOW_LINKS` to follow symbolic links.

NOTE: If you filter the paths returned by `walk` and your filter criterion involves the file attributes stored with a directory, such as size, creation time, or type (file, directory, symbolic link), then use the `find` method instead of `walk`. Call that method with a predicate function that accepts a path and a `BasicFileAttributes` object. The only advantage is efficiency. Since the directory is being read anyway, the attributes are readily available.

This code fragment uses the `Files.walk` method to copy one directory to another:

```
Files.walk(source).forEach(p -> {
    try {
        Path q = target.resolve(source.relativize(p));
        if (Files.isDirectory(p))
            Files.createDirectory(q);
        else
            Files.copy(p, q);
    } catch (IOException ex) {
        throw new UncheckedIOException(ex);
    }
});
```

Unfortunately, you cannot easily use the `Files.walk` method to delete a tree of directories since you need to first visit the children before deleting the parent. In that case, use the `walkFileTree` method. It requires an instance of the `FileVisitor` interface. Here is when the file visitor gets notified:

1. Before a directory is processed:

   ```
   FileVisitResult preVisitDirectory(T dir, IOException ex)
   ```

2. When a file is encountered:

   ```
   FileVisitResult visitFile(T path, BasicFileAttributes attrs)
   ```

3. When an exception occurs in the `visitFile` method:

   ```
   FileVisitResult visitFileFailed(T path, IOException ex)
   ```

4. After a directory is processed:

   ```
   FileVisitResult postVisitDirectory(T dir, IOException ex)
   ```

In each case, the notification method returns one of the following results:

- Continue visiting the next file: `FileVisitResult.CONTINUE`
- Continue the walk, but without visiting the entries in this directory: `FileVisitResult.SKIP_SUBTREE`
- Continue the walk, but without visiting the siblings of this file: `FileVisitResult.SKIP_SIBLINGS`
- Terminate the walk: `FileVisitResult.TERMINATE`

If any of the methods throws an exception, the walk is also terminated, and that exception is thrown from the `walkFileTree` method.

The `SimpleFileVisitor` class implements this interface, continuing the iteration at each point and rethrowing any exceptions.

Here is how you can delete a directory tree:

```
Files.walkFileTree(root, new SimpleFileVisitor<Path>() {
    public FileVisitResult visitFile(Path file,
            BasicFileAttributes attrs) throws IOException {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }
    public FileVisitResult postVisitDirectory(Path dir,
            IOException ex) throws IOException {
        if (ex != null) throw ex;
        Files.delete(dir);
        return FileVisitResult.CONTINUE;
    }
});
```

9.2.5 ZIP File Systems

The `Paths` class looks up paths in the default file system—the files on the user's local disk. You can have other file systems. One of the more useful ones is a ZIP file system. If `zipname` is the name of a ZIP file, then the call

```
FileSystem zipfs = FileSystems.newFileSystem(Path.of(zipname));
```

establishes a file system that contains all files in the ZIP archive. It's an easy matter to copy a file out of that archive if you know its name:

```
Files.copy(zipfs.getPath(sourceName), targetPath);
```

Here, `zipfs.getPath` is the analog of `Path.of` for an arbitrary file system.

To list all files in a ZIP archive, walk the file tree:

```
Files.walk(zipfs.getPath("/")).forEach(p -> {
    Process p
});
```

You have to work a bit harder to create a new ZIP file. Here is the magic incantation:

```
Path zipPath = Path.of("myfile.zip");
var uri = new URI("jar", zipPath.toUri().toString(), null);
    // Constructs the URI jar:file://myfile.zip
try (FileSystem zipfs = FileSystems.newFileSystem(uri,
        Collections.singletonMap("create", "true"))) {
    // To add files, copy them into the ZIP file system
    Files.copy(sourcePath, zipfs.getPath("/").resolve(targetPath));
}
```

NOTE: There is an older API for working with ZIP archives, with classes `ZipInputStream` and `ZipOutputStream`, but it's not as easy to use as the one described in this section.

9.3 HTTP Connections

You can read from a URL by using the input stream returned from `URL.getInputStream` method. However, if you want additional information about a web resource, or if you want to write data, you need more control over the process than the `URL` class provides. The `URLConnection` class was designed before HTTP was the universal protocol of the Web. It provides support for a number of protocols, but its HTTP support is somewhat cumbersome. When the decision was made to support HTTP/2, it became clear that it would be best to provide a modern client interface instead of reworking the existing API. The `HttpClient` provides a more convenient API and HTTP/2 support.

In the following sections, I provide a cookbook for using the `HttpURLConnection` class, and then give an overview of the API.

9.3.1 The `URLConnection` and `HttpURLConnection` Classes

To use the `URLConnection` class, follow these steps:

1. Get an `URLConnection` object:

   ```
   URLConnection connection = url.openConnection();
   ```

   For an HTTP URL, the returned object is actually an instance of `HttpURLConnection`.

2. If desired, set request properties:

   ```
   connection.setRequestProperty("Accept-Charset", "UTF-8, ISO-8859-1");
   ```

   If a key has multiple values, separate them by commas.

3. To send data to the server, call

   ```
   connection.setDoOutput(true);
   try (OutputStream out = connection.getOutputStream()) {
       // Write to out
   }
   ```

4. If you want to read the response headers and you haven't called `getOutputStream`, call

   ```
   connection.connect();
   ```

   Then query the header information:

   ```
   Map<String, List<String>> headers = connection.getHeaderFields();
   ```

   For each key, you get a list of values since there may be multiple header fields with the same key.

5. Read the response:

```
try (InputStream in = connection.getInputStream()) {
    // Read from in
}
```

A common use case is to post form data. The `URLConnection` class automatically sets the content type to `application/x-www-form-urlencoded` when writing data to a HTTP URL, but you need to encode the name/value pairs:

```
URL url = ...;
URLConnection connection = url.openConnection();
connection.setDoOutput(true);
try (var out = new OutputStreamWriter(
        connection.getOutputStream(), StandardCharsets.UTF_8)) {
    Map<String, String> postData = ...;
    boolean first = true;
    for (Map.Entry<String, String> entry : postData.entrySet()) {
        if (first) first = false;
        else out.write("&");
        out.write(URLEncoder.encode(entry.getKey(), "UTF-8"));
        out.write("=");
        out.write(URLEncoder.encode(entry.getValue(), "UTF-8"));
    }
}
try (InputStream in = connection.getInputStream()) {
    ...
}
```

9.3.2 The HTTP Client API

The HTTP client API provides another mechanism for connecting to a web server which is simpler than the `URLConnection` class with its rather fussy set of stages. More importantly, the implementation supports HTTP/2.

An `HttpClient` can issue requests and receive responses. You get a client by calling

```
HttpClient client = HttpClient.newHttpClient();
```

Alternatively, if you need to configure the client, use a builder API like this:

```
HttpClient client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();
```

That is, you get a builder, call methods to customize the item that is going to be built, and then call the `build` method to finalize the building process. This is a common pattern for constructing immutable objects.

Follow the same pattern for formulating requests. Here is a `GET` request:

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://horstmann.com"))
    .GET()
    .build();
```

The URI is the "uniform resource identifier" which is, when using HTTP, the same as a URL. However, in Java, the `URL` class has methods for actually opening a connection to a URL, whereas the `URI` class is only concerned with the syntax (scheme, host, port, path, query, fragment, and so on).

When sending the request, you have to tell the client how to handle the response. If you just want the body as a string, send the request with a `HttpResponse.BodyHandlers.ofString()`, like this:

```
HttpResponse<String> response
    = client.send(request, HttpResponse.BodyHandlers.ofString());
```

The `HttpResponse` class is a template whose type denotes the type of the body. You get the response body string simply as

```
String bodyString = response.body();
```

There are other response body handlers that get the response as a byte array or a file. One can hope that eventually the JDK will support JSON and provide a JSON handler.

With a `POST` request, you similarly need a "body publisher" that turns the request data into the data that is being posted. There are body publishers for strings, byte arrays, and files. Again, one can hope that the library designers will wake up to the reality that most `POST` requests involve form data or JSON objects, and provide appropriate publishers.

In the meantime, to send a form post, you need to URL-encode the request data, just like in the preceding section.

```
Map<String, String> postData = ...;
boolean first = true;
var body = new StringBuilder();
for (Map.Entry<String, String> entry : postData.entrySet()) {
    if (first) first = false;
    else body.append("&");
    body.append(URLEncoder.encode(entry.getKey(), "UTF-8"));
    body.append("=");
    body.append(URLEncoder.encode(entry.getValue(), "UTF-8"));
}
HttpRequest request = HttpRequest.newBuilder()
    .uri(httpUrlString)
    .header("Content-Type", "application/x-www-form-urlencoded")
    .POST(HttpRequest.BodyPublishers.ofString(body.toString()))
    .build();
```

Note that, unlike with the `URLConnection` class, you need to specify the content type for forms.

Similarly, for posting JSON data, you specify the content type and provide a JSON string.

The `HttpResponse` object also yields the status code and the response headers.

```
int status = response.statusCode();
HttpHeaders responseHeaders = response.headers();
```

You can turn the `HttpHeaders` object into a map:

```
Map<String, List<String>> headerMap = responseHeaders.map();
```

The map values are lists since in HTTP, each key can have multiple values.

If you just want the value of a particular key, and you know that there won't be multiple values, call the `firstValue` method:

```
Optional<String> lastModified = headerMap.firstValue("Last-Modified");
```

You get the response value or an empty optional if none was supplied.

TIP: To enable logging for the `HttpClient`, add this line to `net.properties` in your JDK:

```
jdk.httpclient.HttpClient.log=all
```

Instead of `all`, you can specify a comma-separated list of `headers`, `requests`, `content`, `errors`, `ssl`, `trace`, and `frames`, optionally followed by `:control`, `:data`, `:window`, or `:all`. Don't use any spaces.

Then set the logging level for the logger named `jdk.httpclient.HttpClient` to `INFO`, for example by adding this line to the `logging.properties` file in your JDK:

```
jdk.httpclient.HttpClient.level=INFO
```

9.4 Regular Expressions

Regular expressions specify string patterns. Use them whenever you need to locate strings that match a particular pattern. For example, suppose you want to find hyperlinks in an HTML file. You need to look for strings of the pattern `<a href="...">`. But wait—there may be extra spaces, or the URL may be enclosed in single quotes. Regular expressions give you a precise syntax for specifying what sequences of characters are legal matches.

In the following sections, you will see the regular expression syntax used by the Java API, and how to put regular expressions to work.

9.4.1 The Regular Expression Syntax

In a regular expression, a character denotes itself unless it is one of the reserved characters

```
. * + ? { | ( ) [ \ ^ $
```

For example, the regular expression `Java` only matches the string `Java`.

The symbol . matches any single character. For example, `.a.a` matches `Java` and `data`.

The `*` symbol indicates that the preceding constructs may be repeated 0 or more times; for a `+`, it is 1 or more times. A suffix of `?` indicates that a construct is optional (0 or 1 times). For example, `be+s?` matches `be`, `bee`, and `bees`. You can specify other multiplicities with `{ }` (see Table 9-4).

A `|` denotes an alternative: `.(oo|ee)f` matches `beef` or `woof`. Note the parentheses—without them, `.oo|eef` would be the alternative between `.oo` and `eef`. Parentheses are also used for grouping—see Section 9.4.4, "Groups" (page 330).

A *character class* is a set of character alternatives enclosed in brackets, such as `[Jj]`, `[0-9]`, `[A-Za-z]`, or `[^0-9]`. Inside a character class, the - denotes a range (all characters whose Unicode values fall between the two bounds). However, a - that is the first or last character in a character class denotes itself. A ^ as the first character in a character class denotes the complement (all characters except those specified).

There are many *predefined character classes* such as `\d` (digits) or `\p{Sc}` (Unicode currency symbols). See Tables 9-4 and 9-5.

The characters ^ and $ match the beginning and end of input.

If you need to have a literal `. * + ? { | ( ) [ \ ^ $`, precede it by a backslash. Inside a character class, you only need to escape `[` and `\`, provided you are careful about the positions of `] - ^`. For example, `[]^-]` is a class containing all three of them.

Alternatively, surround a string with `\Q` and `\E`. For example, `\(\$0\.99\)` and `\Q($0.99)\E` both match the string `($0.99)`.

TIP: If you have a string that may contain some of the many special characters in the regular expression syntax, you can escape them all by calling `Parse.quote(str)`. This simply surrounds the string with `\Q` and `\E`, but it takes care of the special case where `str` may contain `\E`.

Table 9-4 Regular Expression Syntax

Expression	Description	Example
Characters		
c, not one of `. * + ? { | ( ) [ \ ^ $`	The character *c*.	`J`
`.`	Any character except line terminators, or any character if the `DOTALL` flag is set.	
`\x{`*p*`}`	The Unicode code point with hex code *p*.	`\x{1D546}`
`\u`*hhhh*, `\x`*hh*, `\0`*o*, `\0`*oo*, `\0`*ooo*	The UTF-16 code unit with the given hex or octal value.	`\uFEFF`
`\a`, `\e`, `\f`, `\n`, `\r`, `\t`	Alert (`\x{7}`), escape (`\x{1B}`), form feed (`\x{B}`), newline (`\x{A}`), carriage return (`\x{D}`), tab (`\x{9}`).	`\n`
`\c`*c*, where *c* is in `[A-Z]` or one of `@ [ \ ] ^ _ ?`	The control character corresponding to the character *c*.	`\cH` is a backspace (`\x{8}`).
`\`*c*, where *c* is not in `[A-Za-z0-9]`	The character *c*.	`\\`
`\Q` ... `\E`	Everything between the start and the end of the quotation.	`\Q(...)\E` matches the string (...).
Character Classes		
`[`C_1C_2`...]`, where C_i are characters, ranges *c*-*d*, or character classes	Any of the characters represented by C_1, C_2, . . .	`[0-9+-]`
`[^...]`	Complement of a character class.	`[^\d\s]`
`[...&&...]`	Intersection of character classes.	`[\p{L}&&[^A-Za-z]]`

(Continues)

Table 9-4 Regular Expression Syntax *(Continued)*

Expression	Description	Example
`\p{...}`, `\P{...}`	A predefined character class (see Table 9-5); its complement.	`\p{L}` matches a Unicode letter, and so does `\pL`—you can omit braces around a single letter.
`\d`, `\D`	Digits (`[0-9]`, or `\p{Digit}` when the `UNICODE_CHARACTER_CLASS` flag is set); the complement.	`\d+` is a sequence of digits.
`\w`, `\W`	Word characters (`[a-zA-Z0-9_]`, or Unicode word characters when the `UNICODE_CHARACTER_CLASS` flag is set); the complement.	
`\s`, `\S`	Spaces (`[\n\r\t\f\x{B}]`, or `\p{IsWhite_Space}` when the `UNICODE_CHARACTER_CLASS` flag is set); the complement.	`\s*,\s*` is a comma surrounded by optional white space.
`\h`, `\v`, `\H`, `\V`	Horizontal whitespace, vertical whitespace, their complements.	
Sequences and Alternatives		
XY	Any string from *X*, followed by any string from *Y*.	`[1-9][0-9]*` is a positive number without leading zero.
X|*Y*	Any string from *X* or *Y*.	`http|ftp`
Grouping		
(*X*)	Captures the match of *X*.	`'([^']*)'` captures the quoted text.
\n	The *n*th group.	`(['"]).*\1` matches `'Fred'` or `"Fred"` but not `"Fred'`.

(Continues)

Table 9-4 Regular Expression Syntax *(Continued)*

Expression	Description	Example
`(?<`*name*`>`*X*`)`	Captures the match of *X* with the given name.	`'(?<id>[A-Za-z0-9]+)'` captures the match with name `id`.
`\k<`*name*`>`	The group with the given name.	`\k<id>` matches the group with name `id`.
`(?:`*X*`)`	Use parentheses without capturing *X*.	In `(?:http|ftp)://(.*)`, the match after `://` is `\1`.
`(?`$f_1 f_2$`...:`*X*`)`, `(?`f_1`...-`f_k`...:`*X*`)`, with f_i in `[dimsuUx]`	Matches, but does not capture, *X* with the given flags on or off (after -).	`(?i:jpe?g)` is a case-insensitive match.
Other `(?...)`	See the `Pattern` API documentation.	
Quantifiers		
X`?`	Optional *X*.	`\+?` is an optional + sign.
X`*`, *X*`+`	0 or more *X*, 1 or more *X*.	`[1-9][0-9]+` is an integer ≥ 10.
X`{`*n*`}`, *X*`{`*n*`,}`, *X*`{`*m*`,`*n*`}`	*n* times *X*, at least *n* times *X*, between *m* and *n* times *X*.	`[0-7]{1,3}` are one to three octal digits.
Q`?`, where *Q* is a quantified expression	Reluctant quantifier, attempting the shortest match before trying longer matches.	`.*(<.+?>).*` captures the shortest sequence enclosed in angle brackets.
Q`+`, where *Q* is a quantified expression	Possessive quantifier, taking the longest match without backtracking.	`'[^']*+'` matches strings enclosed in single quotes and fails quickly on strings without a closing quote.
Boundary Matches		
`^ $`	Beginning, end of input (or beginning, end of line in multiline mode).	`^Java$` matches the input or line `Java`.

(Continues)

Table 9-4 Regular Expression Syntax *(Continued)*

Expression	Description	Example
`\A \Z \z`	Beginning of input, end of input, absolute end of input (unchanged in multiline mode).	
`\b \B`	Word boundary, nonword boundary.	`\bJava\b` matches the word `Java`.
`\R`	A Unicode line break.	
`\G`	The end of the previous match.	

Table 9-5 Predefined Character Classes `\p{...}`

Name	Description
posixClass	*posixClass* is one of `Lower`, `Upper`, `Alpha`, `Digit`, `Alnum`, `Punct`, `Graph`, `Print`, `Cntrl`, `XDigit`, `Space`, `Blank`, `ASCII`, interpreted as POSIX or Unicode class, depending on the `UNICODE_CHARACTER_CLASS` flag.
`Is`*Script*, `sc=`*Script*, `script=`*Script*	A script accepted by `Character.UnicodeScript.forName`.
`In`*Block*, `blk=`*Block*, `block=`*Block*	A block accepted by `Character.UnicodeBlock.forName`.
Category, `In`*Category*, `gc=`*Category*, `general_category=`*Category*	A one- or two-letter name for a Unicode general category.
`Is`*Property*	*Property* is one of `Alphabetic`, `Ideographic`, `Letter`, `Lowercase`, `Uppercase`, `Titlecase`, `Punctuation`, `Control`, `White_Space`, `Digit`, `Hex_Digit`, `Join_Control`, `Noncharacter_Code_Point`, `Assigned`.
`java`*Method*	Invokes the method `Character.is`*Method* (must not be deprecated).

9.4.2 Testing a Match

Generally, there are two ways to use a regular expression: Either you want to test whether a string conforms to the expression, or you want to find all matches of the expressions in a string.

In the first case, simply use the static `matches` method:

```
String regex = "[+-]?\\d+";
CharSequence input = ...;
if (Pattern.matches(regex, input)) {
    ...
}
```

If you need to use the same regular expression many times, it is more efficient to compile it. Then, create a `Matcher` for each input:

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) ...
```

If the match succeeds, you can retrieve the location of matched groups—see Section 9.4.4, "Groups" (page 330).

If you want to test whether the input *contains* a match, use the `find` method instead:

```
if (matcher.find()) ...
```

You can turn the pattern into a predicate:

```
Pattern digits = Pattern.compile("[0-9]+");
List<String> strings = List.of("December", "31st", "1999");
List<String> matchingStrings = strings.stream()
    .filter(digits.asMatchPredicate())
    .toList(); // ["1999"]
```

The result contains all strings that match the regular expression.

Use the `asPredicate` method to test whether a string contains a match:

```
List<String> sringsContainingMatch = strings.stream()
    .filter(digits.asPredicate())
    .toList(); // ["31st", "1999"]
```

9.4.3 Finding All Matches

In this section, we consider the other common use case for regular expressions—finding all matches in an input. Use this loop:

```
String input = ...;
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
    String match = matcher.group();
    int matchStart = matcher.start();
    int matchEnd = matcher.end();
    ...
}
```

In this way, you can process each match in turn. As shown in the code fragment, you can get the matched string as well as its position in the input string.

More elegantly, you can call the `results` method to get a `Stream<MatchResult>`. The `MatchResult` interface has methods `group`, `start`, and `end`, just like `Matcher`. (In fact, the `Matcher` class implements this interface.) Here is how you get a list of all matches:

```
List<String> matches = pattern.matcher(input)
    .results()
    .map(Matcher::group)
    .toList();
```

If you have the data in a file, then you can use the `Scanner.findAll` method to get a `Stream<MatchResult>`, without first having to read the contents into a string. You can pass a `Pattern` or a pattern string:

```
var in = new Scanner(path, StandardCharsets.UTF_8);
Stream<String> words = in.findAll("\\pL+")
    .map(MatchResult::group);
```

9.4.4 Groups

It is common to use groups for extracting components of a match. For example, suppose you have a line item in the invoice with item name, quantity, and unit price such as

```
Blackwell Toaster    USD29.95
```

Here is a regular expression with groups for each component:

```
(\p{Alnum}+(\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

After matching, you can extract the `n`th group from the matcher as

```
String contents = matcher.group(n);
```

Groups are ordered by their opening parenthesis, starting at 1. (Group 0 is the entire input.) In this example, here is how to take the input apart:

```
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) {
    item = matcher.group(1);
    currency = matcher.group(3);
    price = matcher.group(4);
}
```

We aren't interested in group 2; it only arose from the parentheses that were required for the repetition. For greater clarity, you can use a noncapturing group:

```
(\p{Alnum}+(?:\s+\p{Alnum}+)*)\s+([A-Z]{3})([0-9.]*)
```

Or, even better, capture by name:

```
(?<item>\p{Alnum}+(\s+\p{Alnum}+)*)\s+(?<currency>[A-Z]{3})(?<price>[0-9.]*)
```

Then, you can retrieve the items by name:

```
item = matcher.group("item");
```

With the `start` and `end` methods, you can get the group positions in the input:

```
int itemStart = matcher.start("item");
int itemEnd = matcher.end("item");
```

NOTE: Retrieving groups by name only works with a `Matcher`, not with a `MatchResult`.

NOTE: When you have a group inside a repetition, such as `(\s+\p{Alnum}+)*` in the example above, it is not possible to get all of its matches. The `group` method only yields the last match, which is rarely useful. You need to capture the entire expression with another group.

9.4.5 Splitting along Delimiters

Sometimes, you want to break an input along matched delimiters and keep everything else. The `Pattern.split` method automates this task. You obtain an array of strings, with the delimiters removed:

```
String input = ...;
Pattern commas = Pattern.compile("\\s*,\\s*");
String[] tokens = commas.split(input);
    // "1, 2, 3" turns into ["1", "2", "3"]
```

If there are many tokens, you can fetch them lazily:

```
Stream<String> tokens = commas.splitAsStream(input);
```

If you don't care about precompiling the pattern or lazy fetching, you can just use the `String.split` method:

```
String[] tokens = input.split("\\s*,\\s*");
```

If the input is in a file, use a scanner:

```
var in = new Scanner(path, StandardCharsets.UTF_8);
in.useDelimiter("\\s*,\\s*");
Stream<String> tokens = in.tokens();
```

9.4.6 Replacing Matches

If you want to replace all matches of a regular expression with a string, call `replaceAll` on the matcher:

```
Matcher matcher = commas.matcher(input);
String result = matcher.replaceAll(",");
    // Normalizes the commas
```

Or, if you don't care about precompiling, use the `replaceAll` method of the `String` class.

```
String result = input.replaceAll("\\s*,\\s*", ",");
```

The replacement string can contain group numbers `$`*n* or names `${`*name*`}`. They are replaced with the contents of the corresponding captured group.

```
String result = "3:45".replaceAll(
    "(\\d{1,2}):(?<minutes>\\d{2})",
    "$1 hours and ${minutes} minutes");
    // Sets result to "3 hours and 45 minutes"
```

You can use `\` to escape `$` and `\` in the replacement string, or you can call the `Matcher.quoteReplacement` convenience method:

```
matcher.replaceAll(Matcher.quoteReplacement(str))
```

If you want to carry out a more complex operation than splicing in group matches, then you can provide a replacement function instead of a replacement string. The function accepts a `MatchResult` and yields a string. For example, here we replace all words with at least four letters with their uppercase version:

```
String result = Pattern.compile("\\pL{4,}")
    .matcher("Mary had a little lamb")
    .replaceAll(m -> m.group().toUpperCase());
    // Yields "MARY had a LITTLE LAMB"
```

The `replaceFirst` method replaces only the first occurrence of the pattern.

9.4.7 Flags

Several *flags* change the behavior of regular expressions. You can specify them when you compile the pattern:

```
Pattern pattern = Pattern.compile(regex,
    Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CHARACTER_CLASS);
```

Or you can specify them inside the pattern:

```
String regex = "(?iU:expression)";
```

Here are the flags:

- `Pattern.CASE_INSENSITIVE` or `i`: Match characters independently of the letter case. By default, this flag takes only US ASCII characters into account.
- `Pattern.UNICODE_CASE` or `u`: When used in combination with `CASE_INSENSITIVE`, use Unicode letter case for matching.
- `Pattern.UNICODE_CHARACTER_CLASS` or `U`: Select Unicode character classes instead of POSIX. Implies `UNICODE_CASE`.
- `Pattern.MULTILINE` or `m`: Make `^` and `$` match the beginning and end of a line, not the entire input.
- `Pattern.UNIX_LINES` or `d`: Only `'\n'` is a line terminator when matching `^` and `$` in multiline mode.
- `Pattern.DOTALL` or `s`: Make the `.` symbol match all characters, including line terminators.
- `Pattern.COMMENTS` or `x`: Whitespace and comments (from `#` to the end of a line) are ignored.
- `Pattern.LITERAL`: The pattern is taking literally and must be matched exactly, except possibly for letter case.
- `Pattern.CANON_EQ`: Take canonical equivalence of Unicode characters into account. For example, u followed by ¨ (diaeresis) matches ü.

The last two flags cannot be specified inside a regular expression.

9.5 Serialization

In the following sections, you will learn about object serialization—a mechanism for turning an object into a bunch of bytes that can be shipped somewhere else or stored on disk, and for reconstituting the object from those bytes.

Serialization is an essential tool for distributed processing, where objects are shipped from one virtual machine to another. It is also used for fail-over and load balancing, when serialized objects can be moved to another server. If you work with server-side software, you will often need to enable serialization for classes. The following sections tell you how to do that.

9.5.1 The `Serializable` Interface

In order for an object to be serialized—that is, turned into a bunch of bytes—it must be an instance of a class that implements the `Serializable` interface. This is a marker interface with no methods, similar to the `Cloneable` interface that you saw in Chapter 4.

For example, to make `Employee` objects serializable, the class needs to be declared as

```
public class Employee implements Serializable {
    private String name;
    private double salary;
    ...
}
```

It is appropriate for a class to implement the `Serializable` interface if all instance variables have primitive or `enum` type, or contain references to serializable objects. Many classes in the standard library are serializable. Arrays and the collection classes that you saw in Chapter 7 are serializable provided their elements are.

In the case of the `Employee` class, and indeed with most classes, there is no problem. In the following sections, you will see what to do when a little extra help is needed.

To serialize objects, you need an `ObjectOutputStream`, which is constructed with another `OutputStream` that receives the actual bytes.

```
var out = new ObjectOutputStream(Files.newOutputStream(path));
```

Now call the `writeObject` method:

```
var peter = new Employee("Peter", 90000);
var paul = new Manager("Paul", 180000);
out.writeObject(peter);
out.writeObject(paul);
```

To read the objects back in, construct an `ObjectInputStream`:

```
var in = new ObjectInputStream(Files.newInputStream(path));
```

Retrieve the objects in the same order in which they were written, using the `readObject` method.

```
var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

When an object is written, the name of the class and the names and values of all instance variables are saved. If the value of an instance variable belongs to a primitive type, it is saved as binary data. If it is an object, it is again written with the `writeObject` method.

When an object is read in, the process is reversed. The class name and the names and values of the instance variables are read, and the object is reconstituted.

There is just one catch. Suppose there were two references to the same object. Let's say each employee has a reference to their boss:

```
var peter = new Employee("Peter", 90000);
var paul = new Manager("Barney", 105000);
var mary = new Manager("Mary", 180000);
peter.setBoss(mary);
paul.setBoss(mary);
out.writeObject(peter);
out.writeObject(paul);
```

When reading these two objects back in, both of them need to have the *same* boss, not two references to identical but distinct objects.

In order to achieve this, each object gets a *serial number* when it is saved. When you pass an object reference to `writeObject`, the `ObjectOutputStream` checks if the object reference was previously written. In that case, it just writes out the serial number and does not duplicate the contents of the object.

In the same way, an `ObjectInputStream` remembers all objects it has encountered. When reading in a reference to a repeated object, it simply yields a reference to the previously read object.

NOTE: If the superclass of a serializable class is not serializable, it must have an accessible no-argument constructor. Consider this example:

```
class Person // Not serializable
class Employee extends Person implements Serializable
```

When an `Employee` object is deserialized, its instance variables are read from the object input stream, but the `Person` instance variables are set by the `Person` constructor.

9.5.2 Transient Instance Variables

Certain instance variables should not be serialized—for example, database connections that are meaningless when an object is reconstituted. Also, when an object keeps a cache of values, it might be better to drop the cache and recompute it instead of storing it.

To prevent an instance variable from being serialized, simply tag it with the `transient` modifier. Always mark instance variables as transient if they hold instances of nonserializable classes. Transient instance variables are skipped when objects are serialized.

9.5.3 The `readObject` and `writeObject` Methods

In rare cases, you need to tweak the serialization mechanism. A serializable class can add any desired action to the default read and write behavior, by defining methods with the signature

```
@Serial private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
@Serial private void writeObject(ObjectOutputStream out)
    throws IOException
```

Then, the object headers continue to be written as usual, but the instance variables fields are no longer automatically serialized. Instead, these methods are called.

Note the `@Serial` annotation. The methods for tweaking serialization don't belong to interfaces. Therefore, you can't use the `@Override` annotation to have the compiler check the method declarations. The `@Serial` annotation is meant to enable the same checking for serialization methods. Up to Java 17, the `javac` compiler doesn't do that checking, but it might happen in the future. Some IDEs check the annotation.

A number of classes in the `java.awt.geom` package, such as `Point2D.Double`, are not serializable. Now, suppose you want to serialize a class `LabeledPoint` that stores a `String` and a `Point2D.Double`. First, you need to mark the `Point2D.Double` field as `transient` to avoid a `NotSerializableException`.

```
public class LabeledPoint implements Serializable {
    private String label;
    private transient Point2D.Double point;
    ...
}
```

In the `writeObject` method, first write the object descriptor and the `String` field, `label`, by calling the `defaultWriteObject` method. This is a special method of the `ObjectOutputStream` class that can only be called from within a `writeObject` method of a serializable class. Then we write the point coordinates, using the standard `DataOutput` calls.

```
@Serial before private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject();
    out.writeDouble(point.getX());
    out.writeDouble(point.getY());
}
```

In the `readObject` method, we reverse the process:

```
@Serial before private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    double x = in.readDouble();
    double y = in.readDouble();
    point = new Point2D.Double(x, y);
}
```

Another example is the `HashSet` class that supplies its own `readObject` and `writeObject` methods. Instead of saving the internal structure of the hash table, the `writeObject` method simply saves the capacity, load factor, size, and elements. The `readObject` method reads back the capacity and load factor, constructs a new table, and inserts the elements.

The `readObject` and `writeObject` methods only need to save and load their data. They do not concern themselves with superclass data or any other class information.

The `Date` class uses this approach. Its `writeObject` method saves the milliseconds since the "epoch" (January 1, 1970). The data structure that caches calendar data is not saved.

CAUTION: Just like a constructor, the `readObject` method operates on partially initialized objects. If you call a non-final method inside `readObject` that is overridden in a subclass, it may access uninitialized data.

NOTE: If a serializable class defines a field

```
@Serial private static final ObjectStreamField[] serialPersistentFields
```

then serialization uses those field descriptors instead of the non-transient non-static fields. There is also an API for setting the field values before serialization or reading them after deserialization. This is useful for preserving a legacy layout after a class has evolved. For example, the `BigDecimal` class uses this mechanism to serialize its instances in a format that no longer reflects the instance fields.

9.5.4 The `readExternal` and `writeExternal` Methods

Instead of letting the serialization mechanism save and restore object data, a class can define its own mechanism. For example, you can encrypt the data or use a format that is more efficient than the serialization format.

To do this, a class must implement the `Externalizable` interface. This, in turn, requires it to define two methods:

```
public void readExternal(ObjectInputStream in)
    throws IOException
public void writeExternal(ObjectOutputStream out)
    throws IOException
```

Unlike the `readObject` and `writeObject` methods, these methods are fully responsible for saving and restoring the entire object, *including the superclass data*. When writing an object, the serialization mechanism merely records the class of the object in the output stream. When reading an externalizable object, the object input stream creates an object with the no-argument constructor and then calls the `readExternal` method.

In this example, the `LabeledPixel` class extends the serializable `Point` class, but it takes over the serialization of the class and superclass. The fields of the object are not stored in the standard serialization format. Instead, the data are placed in an opaque block.

```
public class LabeledPixel extends Point implements Externalizable {
    private String label;

    public LabeledPixel() {} // required for externalizable class

    @Override public void writeExternal(ObjectOutput out)
            throws IOException {
        out.writeInt((int) getX());
        out.writeInt((int) getY());
        out.writeUTF(label);
    }

    @Override public void readExternal(ObjectInput in)
            throws IOException, ClassNotFoundException {
        int x = in.readInt();
        int y = in.readInt();
        setLocation(x, y);
        label = in.readUTF();
    }
    ...
}
```

NOTE: The `readExternal` and `writeExternal` methods should not be annotated with `@Serial`. Since they are defined in the `Externalizable` interface, you can simply annotate them with `@Override`.

CAUTION: Unlike the `readObject` and `writeObject` methods, which are private and can only be called by the serialization mechanism, the `readExternal` and `writeExternal` methods are public. In particular, `readExternal` potentially permits modification of the state of an existing object.

9.5.5 The `readResolve` and `writeReplace` Methods

We take it for granted that objects can only be constructed with the constructor. However, a deserialized object is *not constructed*. Its instance variables are simply restored from an object stream.

This is a problem if the constructor enforces some condition. For example, a singleton object may be implemented so that the constructor can only be called once. As another example, database entities can be constructed so that they always come from a pool of managed instances.

You shouldn't implement your own mechanism for singletons. If you need a singleton, make an enumerated type with one instance that is, by convention, called `INSTANCE`.

```
public enum PersonDatabase {
    INSTANCE;

    public Person findById(int id) { ... }
    ...
}
```

This works because `enum` are guaranteed to be deserialized properly.

Now let's suppose that you are in the rare situation where you want to control the identity of each deserialized instance. As an example, suppose a `Person` class wants to restore its instances from a database when deserializing. Then don't serialize the object itself but some proxy that can locate or construct the object. Provide a `writeReplace` method that returns the proxy object:

```
public class Person implements Serializable {
    private int id;
    // Other instance variables
    ...
```

```
        @Serial private Object writeReplace() {
            return new PersonProxy(id);
        }
    }
```

When a `Person` object is serialized, none of its instance variables are saved. Instead, the `writeReplace` method is called and *its return value* is serialized and written to the stream.

The proxy class needs to implement a `readResolve` method that yields a `Person` instance:

```
    class PersonProxy implements Serializable {
        private int id;

        public PersonProxy(int id) {
            this.id = id;
        }

        @Serial private Object readResolve() {
            return PersonDatabase.INSTANCE.findById(id);
        }
    }
```

When the `readObject` method finds a `PersonProxy` in an `ObjectInputStream`, it deserializes the proxy, calls its `readResolve` method, and returns the result.

NOTE: Unlike the `readObject` and `writeObject` methods, the `readResolve` and `writeReplace` methods need not be private.

NOTE: With enumerations and records, `readObject`/`writeObject` or `readExternal`/`writeExternal` methods are not used for serialization. With records, but not with enumerations, the `writeReplace` method will be used.

9.5.6 Versioning

Serialization was intended for sending objects from one virtual machine to another, or for short-term persistence of state. If you use serialization for long-term persistence, or in any situation where classes can change between serialization and deserialization, you will need to consider what happens when your classes evolve. Can version 2 read the old data? Can the users who still use version 1 read the files produced by the new version?

The serialization mechanism supports a simple versioning scheme. When an object is serialized, both the name of the class and its `serialVersionUID` are written to the object stream. That unique identifier is assigned by the implementor, by defining an instance variable

```
@Serial private static final long serialVersionUID = 1L; // Version 1
```

When the class evolves in an incompatible way, the implementor should change the UID. Whenever a deserialized object has a nonmatching UID, the `readObject` method throws an `InvalidClassException`.

If the `serialVersionUID` matches, deserialization proceeds even if the implementation has changed. Each non-transient instance variable of the object to be read is set to the value in the serialized state, provided that the name and type match. All other instance variables are set to the default: `null` for object references, zero for numbers, and `false` for `boolean` values. Anything in the serialized state that doesn't exist in the object to be read is ignored.

Is that process safe? Only the implementor of the class can tell. If it is, then the implementor should give the new version of the class the same `serialVersionUID` as the old version.

If you don't assign a `serialVersionUID`, one is automatically generated by hashing a canonical description of the instance variables, methods, and supertypes. You can see the hash code with the `serialver` utility. The command

```
serialver ch09.sec05.Employee
```

displays

```
private static final long serialVersionUID = -4932578720821218323L;
```

When the class implementation changes, there is a very high probability that the hash code changes as well.

If you need to be able to read old version instances, and you are certain that is safe to do so, run `serialver` on the old version of your class and add the result to the new version.

NOTE: If you want to implement a more sophisticated versioning scheme, override the `readObject` method and call the `readFields` method instead of the `defaultReadObject` method. You get a description of all fields found in the stream, and you can do with them what you want.

NOTE: Enumerations and records ignore the `serialVersionUID` field. An enumeration always has a `serialVersionUID` of `0L`. You can declare the `serialVersionUID` of a record, but the IDs don't have to match for deserialization.

NOTE: In this section, you saw what happens when the reader's version of a class has instance variables that aren't present in the object stream. It is also possible during class evolution for a superclass to be added. Then a reader using the new version may read an object stream in which the instance variables of the superclass are not set. By default, those instance fields are set to their `0`/`false`/`null` default. That may leave the superclass in an unsafe state. The superclass can defend against that problem by defining an initialization method

```
@Serial private void readObjectNoData() throws ObjectStreamException
```

The method should either set the same state as the no-argument constructor or throw an `InvalidObjectException`. It is only called in the unusual circumstance where an object stream is read that contains an instance of a subclass with missing superclass data.

9.5.7 Deserialization and Security

During deserialization of a serializable class, objects are created without invoking any constructor of the class. Even if the class has a no-argument constructor, it is not used. The field values are set directly from the values of the object input stream.

NOTE: For serializable *records*, deserialization calls the canonical constructor, passing it the values of the components from the object input stream. (As a consequence, cyclic references in records are not restored.)

Bypassing construction is a security risk. An attacker can craft bytes describing an invalid object that could have never been constructed. Suppose, for example, that the `Employee` constructor throws an exception when called with a negative salary. We would like to think that no `Employee` object can have a negative salary as a result. But it is not difficult to inspect the bytes for a serialized object and modify some of them. This way, one can craft bytes for an employee with a negative salary and then deserialize them.

A serializable class can optionally implement the `ObjectInputValidation` interface and define a `validateObject` method to check whether its objects are properly deserialized. For example, the `Employee` class can check that salaries are not negative:

```
public void validateObject() throws InvalidObjectException {
    System.out.println("validateObject");
    if (salary < 0)
        throw new InvalidObjectException("salary < 0");
}
```

Unfortunately, the method is not invoked automatically. To invoke it, you also must provide the following method:

```
@Serial private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
    in.registerValidation(this, 0);
    in.defaultReadObject();
}
```

The object is then scheduled for validation, and the `validateObject` method is called when this object and all dependent objects have been loaded. The second parameter lets you specify a priority. Validation requests with higher priorities are done first.

There are other security risks. Adversaries can create data structures that consume enough resources to crash a virtual machine. More insidiously, any class on the class path can be deserialized. Hackers have been devious about piecing together "gadget chains"—sequences of operations in various utility classes that use reflection and culminate in calling methods such as `Runtime.exec` with a string of their choice.

Any application that receives serialized data from untrusted sources over a network connection is vulnerable to such attacks. For example, some servers serialize session data and deserialize whatever data are returned in the HTTP session cookie.

You should avoid situations in which arbitrary data from untrusted sources are deserialized. In the example of session data, the server should sign the data, and only deserialize data with a valid signature.

A *serialization filter* mechanism can harden applications from such attacks. The filters see the names of deserialized classes and several metrics (stream size, array sizes, total number of references, longest chain of references). Based on those data, the deserialization can be aborted.

In its simplest form, you provide a pattern describing the valid and invalid classes. For example, if you start our sample serialization demo as

```
java -Djdk.serialFilter='serial.*;java.**;!*' serial.ObjectStreamTest
```

then the objects will be loaded. The filter allows all classes in the `serial` package and all classes whose package name starts with `java`, but no others. If you don't allow `java.**`, or at least `java.util.Date`, deserialization fails.

You can place the filter pattern into a configuration file and specify multiple filters for different purposes. You can also implement your own filters. See `https://docs.oracle.com/en/java/javase/17/core/serialization-filtering1.html` for details.

Exercises

1. Write a utility method for copying all of an `InputStream` to an `OutputStream`, without using any temporary files. Provide another solution, without a loop, using operations from the `Files` class, using a temporary file.
2. Write a program that reads a text file and produces a file with the same name but extension `.toc`, containing an alphabetized list of all words in the input file together with a list of line numbers in which each word occurs. Assume that the file's encoding is UTF-8.
3. Write a program that reads a file containing text and, assuming that most words are English, guesses whether the encoding is ASCII, ISO 8859-1, UTF-8, or UTF-16, and if the latter, which byte ordering is used.
4. Using a `Scanner` is convenient, but it is a bit slower than using a `BufferedReader`. Read in a long file a line at a time, counting the number of input lines, with (a) a `Scanner` and `hasNextLine`/`nextLine`, (b) a `BufferedReader` and `readLine`, (c) a `BufferedReader` and `lines`. Which is the fastest? The most convenient?
5. When an encoder of a `Charset` with partial Unicode coverage can't encode a character, it replaces it with a default—usually, but not always, the encoding of `"?"`. Find all replacements of all available character sets that support encoding. Use the `newEncoder` method to get an encoder, and call its `replacement` method to get the replacement. For each unique result, report the canonical names of the charsets that use it.
6. The BMP file format for uncompressed image files is well documented and simple. Using random access, write a program that reflects each row of pixels in place, without writing a new file.
7. Look up the API documentation for the `MessageDigest` class and write a program that computes the SHA-512 digest of a file. Feed blocks of bytes to the `MessageDigest` object with the `update` method, then display the result of calling `digest`. Verify that your program produces the same result as the `sha512sum` utility.
8. Write a utility method for producing a ZIP file containing all files from a directory and its descendants.

9. Using the `URLConnection` class, read data from a password-protected web page with "basic" authentication. Concatenate the user name, a colon, and the password, and compute the Base64 encoding:

   ```
   String input = username + ":" + password;
   String encoding = Base64.getEncoder().encodeToString(
       input.getBytes(StandardCharsets.UTF_8));
   ```

   Set the HTTP header `Authorization` to the value `"Basic " + encoding`. Then read and print the page contents.

10. Using a regular expression, extract all decimal integers (including negative ones) from a string into an `ArrayList<Integer>` (a) using `find`, and (b) using `split`. Note that a `+` or `-` that is not followed by a digit is a delimiter.

11. Using regular expressions, extract the directory path names (as an array of strings), the file name, and the file extension from an absolute or relative path such as `/home/cay/myfile.txt`.

12. Come up with a realistic use case for using group references in `Matcher.replaceAll` and implement it.

13. Implement a method that can produce a clone of any serializable object by serializing it into a byte array and deserializing it.

14. Implement a serializable class `Point` with instance variables for `x` and `y`. Write a program that serializes an array of `Point` objects to a file, and another that reads the file.

15. Continue the preceding exercise, but change the data representation of `Point` so that it stores the coordinates in an array. What happens when the new version tries to read a file generated by the old version? What happens when you fix up the `serialVersionUID`? Suppose your life depended upon making the new version compatible with the old. What could you do?

16. Which classes in the standard Java library implement `Externalizable`? Which of them use `writeReplace`/`readResolve`?

17. Unzip the API source and investigate how the `LocalDate` class is serialized. Why does the class define `writeExternal` and `readExternal` methods even though it doesn't implement `Externalizable`? (Hint: Look at the `Ser` class. Why does the class define a `readObject` method? How could it be invoked?

Index

Symbols and Numbers

- (minus sign)
  flag (for output), 38
  in dates, 434
  in regular expressions, 324
  operator, 18–19
--
  in command-line options, 85
  operator, 18, 20
-= operator, 18–19
->, in lambda expressions, 122, 125
-∞, in string templates, 454
_ (underscore)
  in number literals, 12
  in variable names, 15, 69
, (comma)
  flag (for output), 38
  in numbers, 442, 448, 453
  normalizing, 332
  trailing, in arrays, 48
; (semicolon)
  in Java vs. JavaScript, 470
  path separator (Windows), 86, 260
: (colon)
  in assertions, 204–205
  in dates, 434
  in switch statement, 40
  path separator (Unix), 86, 260
:: operator, 125, 150
! (exclamation sign)
  in property files, 259
  operator, 18, 23
!= operator, 18, 22–23
  for wrapper classes, 50
? (quotation mark)
  in regular expressions, 324–325, 327
  replacement character, 309, 458
  wildcard, for types, 224–228, 239
? : operator, 18, 23
/ (slash)
  file separator (Unix), 260, 313
  in javac path segments, 5
  operator, 18–19
  root component, 312
//, /*...*/ comments, 3
/**...*/ comments, 95
/= operator, 18
. (period)
  in method calls, 6
  in numbers, 442, 448, 453

in package names, 5, 83
in regular expressions, 324–325, 333
operator, 18
.., parent directory, 313
... (ellipsis), for varargs, 57
^ (caret)
for function parameters, 122
in regular expressions, 324–327, 333
operator, 18, 23
^= operator, 18
~ (tilde), operator, 18, 23
'...' (single quotes)
for character literals, 14
in JavaScript, 470
in string templates, 454
"..." (double quotes)
for strings, 6
in javadoc hyperlinks, 98
in text blocks, 34
"" (empty string), 27–28, 159
""", for text boxes, 33–34
((left parenthesis), in formatted output, 38
(...) (parentheses)
empty, for anonymous classes, 138
for casts, 22, 110
in regular expressions, 324–326, 330–331
operator, 18
[...] (square brackets)
for arrays, 46–47, 53
in regular expressions, 324–325
operator, 18
{...} (curly braces)
in annotation elements, 399
in lambda expressions, 122
in regular expressions, 324–327, 332
in string templates, 453
with arrays, 47
{{...}}, double brace initialization, 149
@ (at)
in java command, 492–493
in javadoc comments, 95
$ (dollar sign)
currency symbol, 453
flag (for output), 38
in regular expressions, 324–325, 327, 332–333
in variable names, 15
€ currency symbol, 448, 453
* (asterisk)
for annotation processors, 413
in documentation comments, 96
in regular expressions, 324–327, 331
operator, 18–19
wildcard:
in class path, 86
in imported classes, 88–89
*= operator, 18
\ (backslash)
character literal, 14
file separator (Windows), 260, 313
in option files, 493
in regular expressions, 324–325, 332
in text blocks, 34
& (ampersand), operator, 18, 23–24
&& (double ampersand)
in regular expressions, 325
operator, 18, 23
&= operator, 18
# (number sign)
flag (for output), 38
in javadoc hyperlinks, 98
in option files, 493
in property files, 259
in string templates, 454
% (percent sign)
conversion character, 37
operator, 18–19
%% pattern variable, 213
%= operator, 18
+ (plus sign)
flag (for output), 38
in regular expressions, 324–327
operator, 18–19
for strings, 25, 28, 159
++ operator, 18, 20
+= operator, 18
< (left angle bracket)
flag (for output), 38
in shell syntax, 36
in string templates, 454

operator, 22
<< operator, 18, 23–24
<<= operator, 18
<= operator, 18, 22
<%...%>, <%=...%> delimiters (JSP), 472
≤, in string templates, 454
<> (diamond syntax)
for constructors of generic classes, 221
<...> (angle brackets)
for element types, in array lists, 48
for type parameters, 117, 220
in javadoc hyperlinks, 98
in regular expressions, 327
= operator, 18–19
== operator, 18, 22–23, 161
for class objects, 172
for enumerations, 166
for strings, 26
for wrapper classes, 50
> (right angle bracket)
in shell syntax, 36
operator, 22
>=, >>, >>> operators, 18, 22–23
>>=, >>>= operators, 18
| (vertical bar)
in regular expressions, 324–326
in string templates, 454
operator, 18, 23–24
|= operator, 18
|| operator, 18, 23
0 (zero)
as default value, 73, 76
flag (for output), 38
formatting symbol (date/time), 436
prefix (for octal literals), 12
0b prefix, 12
0x prefix, 12, 38
0xFEFF byte order mark, 305
\0, in regular expressions, 325

A
a formatting symbol (date/time), 436
a, A conversion characters, 37
\a, \A, in regular expressions, 325, 328
abstract classes, 151–152
abstract methods, 123
abstract modifier, 109, 151–152
AbstractCollection class, 114
AbstractMethodError, 115
AbstractProcessor class, 413
accept methods (Consumer, *Xxx*Consumer), 129–130, 277
acceptEither method (CompletableFuture), 357–358
AccessibleObject class
setAccessible method, 181, 183
trySetAccessible method, 181
accessors, 64
accumulate method (LongAccumulator), 374
accumulateAndGet method (Atomic*Xxx*), 373
accumulator functions, 293
ActionListener interface, 120
add method
of ArrayDeque, 262
of ArrayList, 49, 64
of BlockingQueue, 371
of Collection, 249
of List, 250
of ListIterator, 253
of LongAdder, 374
addAll method
of Collection, 226, 249
of Collections, 251
of List, 250
addExact method (Math), 20
addition, 19
identity for, 292
addSuppressed method (IOException), 199
aggregators, 494
allMatch method (Stream), 280
allOf method
of CompletableFuture, 357–358
of EnumSet, 262
allProcesses method (ProcessHandle), 389
and, andNot methods (BitSet), 261
and, andThen methods (functional interfaces), 129
Android, 121, 359
AnnotatedConstruct interface, 414
AnnotatedElement interface, 411–413
annotation interfaces, 403–406
annotation processors, 413

annotations
  accessing, 404, 494
  applicability of, 406–407
  container, 409, 412
  declaration, 400–401
  documented, 407–408
  generating source code with, 415–417
  inherited, 407–408, 411
  key/value pairs in, 398–399, 405
  meta, 404–410
  modifiers and, 402
  multiple, 400
  processing:
    at runtime, 410–413
    source-level, 413–417
  repeatable, 400, 407, 409–410, 412
  standard, 406–410
  type use, 401–402
anonymous classes, 138
anyMatch method (Stream), 280
anyOf method (CompletableFuture), 357–358
Apache Commons CSV, 490
API documentation, 29–31
  generating, 95
Applet class, 174
applications. *See* programs
apply, applyAs*Xxx* methods (functional interfaces), 129–130
applyToEither method (CompletableFuture), 357–358
arithmetic operations, 17–24
Array class, 185–186
array lists, 48–49
  anonymous, 149
  checking for nulls, 227
  constructing, 49
  converting between, 224
  copying, 51
  elements of, 49–50
  filling, 52
  instantiating with type variables, 234
  size of, 49
  sorting, 52
  variables of, 49
array variables
  assigning values to, 48
  copying, 51
  declaring, 46–47
  initializing, 46
ArrayBlockingQueue class, 371
ArrayDeque class, 262
ArrayIndexOutOfBoundsException, 47
ArrayList class, 48–49, 248
  add method, 49, 64
  clone method, 165–166
  forEach method, 125
  get, remove methods, 49
  removeIf method, 124
  set, size methods, 49
arrays, 46–48
  accessing nonexisting elements in, 47
  allocating, 234
  annotating, 401
  casting, 185
  checking, 185
  comparing, 161
  computing values of, 367
  constructing, 46–47
  constructor references with, 126
  converting:
    to a reference of type Object, 157
    to/from streams, 286, 296, 368
  copying, 51
  covariant, 223
  filling, 47, 52
  generating Class objects for, 171
  growing, 185–186
  hash codes of, 163
  length of, 47–48, 134
  multidimensional, 53–55, 159
  of bytes, 302–303
  of generic types, 126, 235
  of objects, 47, 367
  of primitive types, 367
  of strings, 331
  passing into methods, 56
  printing, 52, 55, 159
  serializable, 334
  sorting, 52, 117–119, 367–368
  superclass assignment in, 148
  using class literals with, 171

Arrays class
  asList method, 265
  copyOf method, 51, 186
  deepToString method, 159
  equals method, 161
  fill method, 52
  hashCode method, 163
  parallel*Xxx* methods, 52, 367
  setAll method, 127
  sort method, 52, 119, 123–124
  stream method, 274, 294
  toString method, 52, 159
ArrayStoreException, 148, 223, 235
ASCII (American Standard Code for Information Interchange), 31–32, 305
  for property files, 457
  for source files, 458
ASM tool, 417
asMatchPredicate, asPredicate methods (Pattern), 329
assert statement, 204–205
AssertionError, 204
assertions, 204–206
  checking, 401
  enabling/disabling, 205–206
assignment operators, 18–19
associative operations, 292
asSubclass method (Class), 239
asynchronous computations, 353–359
AsyncTask class (Android), 359
atomic operations, 364, 369, 373–375, 379
  performance and, 374
Atomic*Xxx* classes, 373
atZone method (LocalDateTime), 430
@author tag (javadoc), 96, 99
autoboxing, 50, 131
AutoCloseable interface, 197, 222
  close method, 198
automatic modules, 489–491
availableCharsets method (Charset), 306
availableProcessors method (Runtime), 349
average method (*Xxx*Stream), 295

B

b, B conversion characters, 37
\b (backspace), 14
\b, \B, in regular expressions, 328
BasicFileAttributes class, 317
BeanInfo class, 184
between method (Duration), 423
BiConsumer interface, 129
BiFunction interface, 129, 131
BigDecimal class, 14, 24, 337
big-endian format, 305, 310–311
BigInteger class, 12, 24
binary data, reading/writing, 310
binary numbers, 12, 14
binary trees, 254
BinaryOperator interface, 129
binarySearch method (Collections), 252
bindings, 469
Bindings interface, 469
BiPredicate interface, 129
BitSet class, 260–261
  collecting streams into, 293
  methods of, 261
bitwise operators, 23–24
block statement, labeled, 44
blocking queues, 370–372
BlockingQueue interface, 371
Boolean class, 49
boolean type, 14
  default value of, 73, 76
  formatting for output, 37
  reading/writing, 310
  streams of, 294
BooleanSupplier interface, 130
bootstrap class loader, 174
boxed method (*Xxx*Stream), 294
branches, 38–39
break statement, 40–41, 43–44
bridge methods, 230–231
  clashes of, 237
BufferedReader class, 308
build method (HttpClient), 321
bulk operations, 370
Byte class, 49
  MAX_VALUE, MIN_VALUE constants, 11
  toUnsignedInt method, 12

byte codes, 4
  writing to memory, 466–467
byte order mark, 305
byte type, 11–12, 303
  streams of, 294
  type conversions of, 21
ByteArrayClass class, 466
ByteArrayClassLoader class, 467
ByteArray*Xxx*Stream classes, 302–303
ByteBuffer class, 311
bytes
  arrays of, 302–303
  converting to strings, 307
  reading, 303
  skipping, writing, 304

C

c, C conversion characters, 37
C:\ root component, 312
C/C++ programming languages
  #include directive in, 89
  allocating memory in, 364
  integer types in, 12
  pointers in, 65
C# programming language, 227
\c, in regular expressions, 325
CachedRowSetImpl class, 492
calculators, 168–169
Calendar class, 421
  getFirstDayOfWeek method, 451
  weekends in, 427
calendars, 62
call by reference, 71
call method (CompilationTask), 465
Callable interface, 120
  call method, 351
  extending, 465
callbacks, 120–121, 355
  registering, 353
camel case, 16
cancel method
  of CompletableFuture, 355
  of Future, 351
cancellation requests, 382
CancellationException, 355
cardinality method (BitSet), 261
carriage return, 14
case label, 39–41
cast method (Class), 239
cast operator, 22
casts, 22, 110–111, 148
  annotating, 402
  generic types and, 232
  inserting, 229–230
catch statement, 196–197
  annotating parameters of, 400
  in try-with-resources, 199
  no type variables in, 237
ceiling method (NavigableSet), 255
Channel interface, 112
channels, 311
char type, 14
  streams of, 294
  type conversions of, 21
Character class, 49
character classes, 324
character encodings, 305–307
  detecting, 306
  localizing, 458
  partial, 306, 309
  platform, 306, 458
character literals, 14
characters, 302
  combined, 452
  formatting for output, 37
  normalized, 452–453
  reading/writing, 310
charAt method (String), 32
CharSequence interface, 29, 275
  chars, codePoints methods, 294
Charset class
  availableCharsets method, 306
  defaultCharset method, 306, 458
  displayName method, 458
  forName method, 307
checked exceptions, 194–196
  combining in a superclass, 195
  declaring, 195–196
  documenting, 196
  generic types and, 238
  in lambda expressions, 196
  no-argument constructors and, 182

not allowed in a method, 202
rethrowing, 201
checked views, 233, 266
checkedXxx methods (Collections), 252, 266
Checker Framework, 401
checkIndex method (Objects), 204
childrenNames method (Preferences), 460
choice indicator, in string templates, 454
Church, Alonzo, 122, 424
Class class, 170–173, 240
asSubclass, cast methods, 239
comparing objects of, 172
forName method, 171–172, 175–176, 194, 203, 467
generic, 239
getCanonicalName method, 171–172
getClassLoader method, 173
getComponentType method, 172, 185
getConstructor(s) methods, 173, 179, 182, 239
getDeclaredConstructor(s) methods, 173, 179, 239
getDeclaredField(s) methods, 173
getDeclaredMethod(s) methods, 173, 182
getDeclaringClass method, 172
getEnclosingXxx methods, 172
getEnumConstants method, 239
getField(s) methods, 173, 179
getInterfaces method, 172
getMethod(s) methods, 173, 179, 182
getModifiers method, 172
getName method, 171–172
getPackage method, 172
getPackageName method, 173
getPermittedSubclasses method, 172
getRecordComponents method, 173
getResource method, 174, 455
getResourceAsStream method, 173–174
getSimpleName method, 172
getSuperclass method, 172, 239
getTypeName method, 172
getTypeParameters method, 240
isXxx methods, 172–173, 185
newInstance method, 182, 239
toGenericString method, 172
toString method, 172
class declarations
annotations in, 400, 408
initialization blocks in, 74–75
class files, 4, 174
paths of, 84
processing annotations in, 417
class literals, 171
no annotations for, 402
no type variables in, 233
class loaders, 174–176, 467
class objects, 171
class path, 85, 477
.class suffix, 171–172
ClassCastException, 110, 232
classes, 2, 62
abstract, 109, 116, 151–152
accessing from a different module, 494
adding to packages, 88
anonymous, 138
companion, 114
compiling on the fly, 466
constructing objects of, 15
deprecated, 97, 406–407
deserialization of, 342–344
documentation comments for, 95–96
encapsulation of, 475–476
evolving, 340
extending, 144–150
fields of, 143
final, 150–151
generic, 48
immutable, 29, 365
implementing, 67–71, 164
importing, 88–89
inner, 91–93
instances of, 6, 67, 83
loading, 180
local, 137–138
members of, 143
enumerating, 169, 179–180
naming, 15–16, 83, 171
nested, 90–95, 402
not known at compile time, 171, 186
protected, 152–153
public, 88, 482
sealed, 154

serializable, 336–337
static initialization of, 175
static methods of, 82
system, 205
testing, 88
utility, 87, 176
wrapper, 49–50
classes win rule, 163
classifier functions, 289
ClassLoader class
defineClass method, 492
extending, 467
findClass, loadClass methods, 175
set*Xxx*AssertionStatus methods, 206
classloader inversion, 176
ClassNotFoundException, 194
CLASSPATH environment variable, 87
clear method
of BitSet, 261
of Collection, 249
of Map, 257
clone method
of ArrayList, 165–166
of Enum, 167
of Message, 165–166
of Object, 153, 158, 163–166, 182
protected, 163
Cloneable interface, 165
CloneNotSupportedException, 165–167
cloning, 163–166
close method
of AutoCloseable, Closeable, 198
of PrintWriter, 197–198
throwing exceptions, 198
Closeable interface, 112
close method, 198
closures, 133
COBOL, scripting engine for, 468
code element (HTML), 96
code generator tools, 408
code points, 32, 276, 305
code units, 14, 32, 294
in regular expressions, 325
codePoints method
of CharSequence, 294
of String, 32–33, 276–278
codePoint*Xxx* methods (String), 32
Collator class, 28
methods of, 452
collect method (Stream), 286–287, 293
Collection interface, 114, 248
add method, 249
addAll method, 226, 249
clear, contains, containsAll methods, 249
isEmpty method, 249
iterator method, 249
parallelStream method, 249, 272–273, 295, 366
remove, remove*Xxx*, retainAll methods, 249
size method, 249
spliterator method, 249
stream method, 249, 272–273
toArray method, 249
collections, 247–266
branching, 291
generic, 266
given elements of, 264
iterating over elements of, 272–273
mutable, 265
processing, 251
serializable, 334
threadsafe, 372
unmodifiable views of, 265–266
vs. streams, 273
Collections class, 114, 251
addAll method, 251
binarySearch method, 252
copy method, 251
disjoint method, 251
fill method, 52, 251
frequency method, 251
indexOfSubList, lastIndexOfSubList methods, 251
nCopies method, 249, 251
replaceAll method, 251
reverse method, 52, 252
rotate method, 252
shuffle method, 52, 252
sort method, 52, 226–227, 241, 252
swap method, 252
synchronized*Xxx* methods, 252
unmodifiable*Xxx* methods, 252

Collector interface, 286
Collectors class, 90
  counting method, 290
  filtering method, 291
  flatMapping method, 291
  groupingBy method, 289–292
  groupingByConcurrent method, 289, 296
  joining method, 286–287
  mapping method, 290
  maxBy, minBy methods, 290
  partitioningBy method, 289, 292
  reducing method, 291
  summarizing*Xxx* methods, 287, 291
  summing*Xxx* methods, 290
  teeing method, 291
  toCollection method, 286
  toConcurrentMap method, 288
  toMap method, 287–288
  toSet method, 286, 290
command-line arguments, 52–53
comments, 3
  documentation, 95–100
commonPool method (ForkJoinPool), 297, 353
companion classes, 114
Comparable interface, 117–119, 167, 226, 254
  compareTo method, 117
  priority queues with, 263
  streams of, 279
Comparator interface, 90, 119–120, 135–137, 254
  comparing, comparing*Xxx* methods, 136–137
  naturalOrder method, 136
  nullsFirst, nullsLast methods, 136
  priority queues with, 263
  reversed method, 136
  reverseOrder method, 137
  streams of, 279
  thenComparing method, 136–137
compare method (Integer, Double), 118
compareTo method
  of Enum, 167
  of Instant, 423
  of String, 27–28, 117, 451
compareToIgnoreCase method (String), 124
compareUnsigned method (Integer, Long), 21
compatibility, drawbacks of, 228
Compilable interface, 471
compilation, 4
CompilationTask interface, 464
  call method, 465
compile method (Pattern), 329, 333
compiler
  instruction reordering in, 361
  invoking, 464
compile-time errors, 16, 112
completable futures, 353–358
  combining, 358
  composing, 355–358
  interrupting, 355
CompletableFuture class, 353–358
  acceptEither method, 357–358
  allOf, anyOf methods, 357–358
  applyToEither method, 357–358
  cancel method, 355
  complete, completeExceptionally methods, 354
  completeOnTimeout method, 357
  exceptionally method, 356–357
  exceptionallyCompose method, 357
  handle method, 357
  isDone method, 354
  orTimeout method, 357
  runAfter*Xxx* methods, 357–358
  supplyAsync method, 353–355
  thenAccept method, 353, 357
  thenAcceptBoth method, 357–358
  thenApply, thenApplyAsync methods, 355–357
  thenCombine method, 357–358
  thenCompose method, 356–357
  thenRun method, 357
  whenComplete method, 354, 356–357
CompletionStage interface, 358
compose method (functional interfaces), 129
computations
  asynchronous, 353–359
  mutator, 64
  precision of, 14

compute method
  of ConcurrentHashMap, 369–370
  of Map, 256
computeIf*Xxx* methods
  of ConcurrentHashMap, 369
  of Map, 256–257
concat method (Stream), 278
concatenation, 25
  objects with strings, 159
concurrent programming, 347–389
  access errors in, 134
  for scripts, 469
  strategies for, 364
ConcurrentHashMap class, 369–370, 381
  compute method, 369–370
  computeIf*Xxx* methods, 369
  forEach*Xxx* methods, 370
  keySet method, 372
  merge method, 369–370
  newKeySet method, 372
  no null values in, 258
  putIfAbsent method, 369
  reduce*Xxx* methods, 370
  search*Xxx* methods, 370
  threadsafe, 377
ConcurrentModificationException, 253, 368
ConcurrentSkipList*Xxx* classes, 372
conditional operator, 23
configuration files, 459–461
  editing, 209–211
  locating, 174
  resolving paths for, 313
confinement, 364
connect method (URLConnection), 320
Console class, 36
console, displaying fonts on, 458
ConsoleHandler class, 211, 213
constants, 16–17, 113
  naming, 16
  static, 80–81
  using in another class, 17
Constructor class, 179–180
  getModifiers method, 179
  getName method, 179
  newInstance method, 182–183
constructor references, 126
  annotating, 402
constructors, 71–76
  abstract classes and, 152
  annotating, 236, 400–401
  canonical, compact, custom, 78–79
  documentation comments for, 95
  executing, 72
  for subclasses, 147
  implementing, 71–72
  invoking another constructor from, 73
  no-argument, 75, 147, 182
  overloading, 72–73
  public, 72, 179
  references in, 366
Consumer interface, 129, 277
contains method
  of String, 29
  of Collection, 249
containsAll method (Collection), 249
contains*Xxx* methods (Map), 257
Content-Type header, 306
context class loaders, 176–177
continue statement, 43–44
control flow, 38–46
conversion characters, 37
cooperative cancellation, 382
copy method
  of Collections, 251
  of Files, 304, 315–316, 319
copyOf method (Arrays), 51, 186
CopyOnWriteArray*Xxx* classes, 372
CORBA (Common Object Request Broker Architecture), 476
count method (Stream), 273, 280
counters
  atomic, 373–375
  de/incrementing, 199
counting method (Collectors), 290
country codes, 289, 443–444
covariance, 223
createBindings method (ScriptEngine), 469
createDirectory, createDirectories, createFile methods (Files), 314
createInstance method (Util), 176–177
createTemp*Xxx* methods (Files), 315

critical sections, 364, 375, 382
Crockford, Douglas, 471
currencies, 448–449
  formatting, 453
`Currency` class, 448
`current` method
  of `ProcessHandlex`, 389
  of `ThreadLocalRandomx`, 384

D

`d`
  conversion character, 37
  formatting symbol (date/time), 436
`D` suffix, 13
`\d`, `\D`, in regular expressions, 326
daemon threads, 385
databases, 397
  persisting objects in, 485
`DataInput/Output` interfaces, 310
  `read/write`*`Xxx`* methods, 310–311
`Data`*`Xxx`*`Stream` classes, 310
`Date` class, 421, 436–437
`DateFormat` class, 449
dates
  computing, 428–429
  formatting, 433–436, 442, 449–451, 453
  local, 424–427
  nonexistent, 427, 431, 450
  parsing, 435
`datesUntil` method (`LocalDate`), 426–427
`DateTimeFormat` class, 449–451
`DateTimeFormatter` class, 433–436
  `format` method, 433, 450
  legacy classes and, 437
  `ofLocalized`*`Xxx`* methods, 433, 449
  `ofPattern` method, 435
  `parse` method, 435
  `toFormat` method, 435
  `withLocale` method, 434, 450
`DateTimeParseException`, 450
daylight savings time, 430–433
`DayOfWeek` enumeration, 63, 426–427, 432
  `getDisplayName` method, 435, 450
`dayOfWeekInMonth` method (`TemporalAdjusters`), 428
deadlocks, 364, 376, 380, 382
debugging
  messages for, 193
  overriding methods for, 151
  primary arrays for, 52
  streams, 279
  threads, 385
  with anonymous subclasses, 149–150
  with assertions, 204
`DecimalFormat` class, 83
  number format patterns of, 453
declaration-site variance, 227
decomposition
  of characters, 452
  of classes, 56–57
decrement operator, 20
`decrementExact` method (`Math`), 20
deep copies, 164
`deepToString` method (`Arrays`), 159
`default` label (in `switch`), 39–41
default methods, 114–116
  conflicts of, 115–116, 157
  in interfaces, 163
`default` modifier, 114, 405
`defaultCharset` method (`Charset`), 306, 458
`defaultReadObject` method (`ObjectInputStream`), 337, 341
`defaultWriteObject` method (`ObjectOutputStream`), 336–337
defensive programming, 204
deferred execution, 127–128
`defineClass` method (`ClassLoader`), 492
`delete` method (`Files`), 315
`deleteIfExists` method (`Files`), 315
delimiters, for scanners, 308
`@Deprecated` annotation, 97, 406–407
`@deprecated` tag (`javadoc`), 97, 407
`Deque` interface, 250, 262
`destroy`, `destroyForcibly` methods
  of `Process`, 389
  of `ProcessHandle`, 390
`DiagnosticCollector` class, 465
`DiagnosticListener` interface, 465
diamond syntax (`<>`)
  for array lists, 49
  for constructors of generic classes, 221

directories, 312
  checking for existence, 314, 316
  creating, 314–316
  deleting, 315, 318–319
  moving, 315
  temporary, 315
  user, 314
  visiting, 316–319
  working, 386
directory method (ProcessBuilder), 386
disjoint method (Collections), 251
displayName method (Charset), 458
distinct method (Stream), 279, 296
dividedBy method (Duration), 424
divideUnsigned method (Integer, Long), 21
division, 19
do statement, 42
doc-files directory, 96
documentation comments, 95–100
@Documented annotation, 407–408
domain names
  for modules, 478
  for packages, 83
dot notation, 6, 17
double brace initialization, 149
Double class, 49
  compare method, 118
  equals method, 161
  isFinite, isInfinite methods, 13
  NaN, NEGATIVE_INFINITY, POSITIVE_INFINITY values, 13
  parseDouble method, 28
  toString method, 28
double type, 13–14
  atomic operations on, 375
  functional interfaces for, 130
  streams of, 294
  type conversions of, 21–22
DoubleAccumulator, DoubleAdder classes, 375
DoubleConsumer, Double*Xxx*Operator, DoublePredicate, DoubleSupplier, DoubleTo*Xxx*Function interfaces, 130
DoubleFunction interface, 130, 232
doubles method (RandomGenerator), 294
DoubleStream class, 294–295
DoubleSummaryStatistics class, 287, 295
doubleValue method (Number), 448
downstream collectors, 289–292, 296
Driver.parentLogger method, 494
dropWhile method (Stream), 278
Duration class
  between method, 423
  dividedBy method, 424
  immutability of, 365, 424
  isNegative, isZero methods, 424
  minus, minus*Xxx*, multipliedBy, negated methods, 424
  of*Xxx* methods, 423–424, 426, 431
  plus, plus*Xxx* methods, 424
  to*Xxx* methods, 423
dynamic method lookup, 148, 230–231

E

E constant (Math), 20
e, E
  conversion characters, 37
  formatting symbols (date/time), 436
\e, \E, in regular expressions, 324–325
Eclipse IDE, 5
effectively final variables, 133–134
efficiency, and final modifier, 151
Element interface, 414
element method (BlockingQueue), 371
elements (in annotations), 398–399, 405
else statement, 39
em element (HTML), 96
empty method
  of Optional, 284
  of Stream, 274
empty string, 27, 159
  concatenating, 28
encapsulation, 62, 475–477, 485
encodings. *See* character encodings
end method (Matcher, MatchResult), 330–331
endsWith method (String), 29
engine scope, 469
enhanced for loop, 50, 55, 134
  for collections, 253
  for enumerations, 167
  for iterators, 178
  for paths, 314
Entry class, 229

entrySet method (Map), 257–258
Enum class, 166–167
enum instances
  adding methods to, 168–169
  construction, 168
  referred by name, 170
enum keyword, 17, 166
enumeration sets, 262
enumerations, 166–170
  annotating, 400
  comparing, 166–167
  constructing, 168
  defining, 17
  nested inside classes, 169
  serialization of, 339
  static members of, 169
  traversing instances of, 167
  using in switch, 170
EnumMap, EnumSet classes, 262
environment variables, 387
epoch, 422
equality, testing for, 22–23
equals method
  final, 162
  null-safe, 161
  of Arrays, 161
  of Double, 161
  of Instant, 423
  of Object, 158–162
  of Objects, 161
  of records, 77
  of String, 26–27
  of subclasses vs. superclass, 161
  of wrapper classes, 50
  overriding, 160–162
  symmetric, 161
  values from different classes and, 161
equalsIgnoreCase method (String), 27
Error class, 193
error messages, for generic methods, 222
errorReader method (Process), 387
errors
  AbstractMethodError, 115
  AssertionError, 204
eval method (ScriptEngine), 468–471
even numbers, 19
Exception class, 194
exceptionally method (CompletableFuture), 356–357
exceptionallyCompose method (CompletableFuture), 357
exceptions, 192–204
  annotating, 402
  ArrayIndexOutOfBoundsException, 47
  ArrayStoreException, 148, 223, 235
  CancellationException, 355
  catching, 196–200
  chaining, 201–202
  checked, 182, 194–196
  ClassCastException, 110, 232
  ClassNotFoundException, 194
  CloneNotSupportedException, 165–167
  combining in a superclass, 195
  ConcurrentModificationException, 253, 368
  creating, 194–195
  DateTimeParseException, 450
  documenting, 196
  ExecutionException, 351
  FileNotFoundException, 194
  generic types and, 237–238
  hierarchy of, 193–195
  IllegalArgumentException, 204
  IllegalStateException, 287, 371
  InaccessibleObjectException, 181, 486
  IndexOutOfBoundsException, 204
  InterruptedException, 381, 383
  InvalidClassException, 341
  InvalidPathException, 313
  IOException, 194, 199, 308
  NoSuchElementException, 283, 371
  NullPointerException, 27, 48, 66, 74, 194, 203, 256, 280
  NumberFormatException, 194
  ParseException, 448
  ReflectiveOperationException, 171
  rethrowing, 199–202
  RuntimeException, 194
  SecurityException, 181
  ServletException, 201–202
  suppressed, 199
  throwing, 192–193
  TimeoutException, 351

uncaught, 202
unchecked, 194
UncheckedIOException, 308
exec method (Runtime), 386
Executable class
getModifiers method, 183
getName method, 183
getParameters method, 180, 183
ExecutableElement interface, 414
ExecutionException, 351
Executor interface, 356
executor services, 349, 353
ExecutorCompletionService class, 352
Executors class, 349
ExecutorService interface, 465
execute method, 349
invokeAll, invokeAny methods, 352
exists method (Files), 314, 316
exitValue method (Process), 389
exports keyword, 479, 482–485
qualified, 495
exportSubtree method (Preferences), 460
extends keyword, 112, 145, 222–226
Externalizable interface, read/writeExternal methods, 338–339

F

f conversion character, 37
F suffix, 13
\f, in regular expressions, 325
factory methods, 72, 83
failures, logging, 201
false value (boolean), 14
as default value, 73, 76
Field class, 179–180
get method, 181, 183
getBoolean, getByte, getChar, getDouble, getFloat, getInt, getLong methods, 181, 183
getModifiers, getName methods, 179, 183
getShort method, 181, 183
getType method, 179
set, set*Xxx* methods, 183
fields (instance and static variables), 143
enumerating, 179–180
final, 362
provided, 153
public, 179
retrieving values of, 180–181
setting, 181
transient, 336
file attributes
copying, 315
filtering paths by, 317
File class, 314
file handlers, 211–212
file managers, 466
file pointers, 311
file.encoding system property, 306
file.separator system property, 260
FileChannel class
get, get*Xxx* methods, 311
lock method, 312
open method, 311
put, put*Xxx* methods, 311
tryLock method, 312
FileFilter class, 128
FileHandler class, 211–213
FileNotFoundException, 194
files
archiving, 319
channels to, 311
checking for existence, 194, 314–316
closing, 197
copying, 315–316
creating, 313–316
deleting, 315
empty, 314
encoding of, 305
locking, 312
memory-mapped, 297, 311
missing, 465
moving, 315–316
random-access, 310–311
reading from/writing to, 36, 194, 303
temporary, 315
Files class
copy method, 304, 315–316, 319
createTemp*Xxx* methods, 315
create*Xxx* methods, 314
delete, deleteIfExists methods, 315
exists method, 314, 316

find method, 316–317
isDirectory, isRegularFile methods, 314, 316
lines method, 275, 297, 307
list method, 316–317
move method, 315–316
newBufferedReader method, 308, 468
newBufferedWriter method, 308, 316
new*Xxx*Stream methods, 302, 316, 334
read method, 304
readAllBytes method, 303, 307
readAllLines method, 307
readNBytes method, 303
skipNBytes method, 304
walk method, 316–319
walkFileTree method, 316, 318
write method, 309, 316
FileSystem, FileSystems classes, 319
FileTime class, 437
FileVisitor interface, 318
fill method
of Arrays, 52
of Collections, 52, 251
Filter interface, 213
filter method
of Optional, 282
of Stream, 273–276, 280
filtering method (Collectors), 291
final fields, 362
final methods, 366
final modifier, 16, 75, 150
final variables, 361, 365
finalize method (Object), 158
finally statement, 199–200
for locks, 376
return statements in, 199
financial calculations, 14
find method (Files), 316–317
findAll method (Scanner), 330
findAny method (Stream), 280
findClass method (ClassLoader), 175
findFirst method (Stream), 179, 280
first day of week, 451
first method (SortedSet), 255
firstDayOf*Xxx* methods (TemporalAdjusters), 428
flag bits, sequences of, 260
flatMap method
of Optional, 284–285
of Stream, 277
flatMapping method (Collectors), 291
flip method (BitSet), 261
Float class, 49
float type, 13–14
streams of, 294
type conversions of, 21–22
floating-point types, 13–14
binary number system and, 14
comparing, 118
division of, 19
formatting for output, 37
in hexadecimal notation, 13
type conversions of, 21–22
floor method (NavigableSet), 255
floorMod method (Math), 20
fonts, displaying, 458
for statement, 42–43
declaring variables for, 45
enhanced, 50, 55, 134, 167, 253, 314
multiple variables in, 43
forEach method
of ArrayList, 125
of Map, 257
of Stream, 286
forEachOrdered method (Stream), 286
forEach*Xxx* methods (ConcurrentHashMap), 370
ForkJoinPool class, 356
commonPool method, 297, 353
forLanguageTag method (Locale), 446
Format class, 437
format method
of DateTimeFormatter, 433, 450
of MessageFormat, 453–454
of String, 447
format specifiers, 37
formatted method (String), 38
formatted output, 36–38
Formatter class, 213
formatters, for date/time values, 434–435
forms, posting data from, 321–323

forName method
  of Charset, 307
  of Class, 171–172, 175–176, 194, 203, 467
frequency method (Collections), 251
from method (Instant, ZonedDateTime), 436
full indicator, in string templates, 453
Function interface, 129, 287
function types, 121, 128
functional interfaces, 123–124, 406, 408
  as method parameters, 225–226
  common, 129
  contravariant in parameter types, 225
  for primitive types, 130
  implementing, 130–131
@FunctionalInterface annotation, 131, 406, 408
functions, 62
  higher-order, 135–137
Future interface, 352
  cancel, isCancelled, isDone methods, 351
  get method, 351, 353
futures, 351–353
  completable, 353–358

G

g, G
  conversion characters, 37
  formatting symbols (date/time), 436
\G, in regular expressions, 328
%g pattern variable, 213
gadget chains, 343
garbage collector, 263
generate method (Stream), 274, 294
@Generated annotation, 406, 408
generators, converting to streams, 296
generic classes, 48, 220–221
  constructing objects of, 221
  information available at runtime, 239
  instantiating, 221
generic collections, 266
generic constructors, 240
generic methods, 221–222
  calling, 221
  declaring, 221
  information available at runtime, 239
generic type declarations, 240–241
generic types, 118
  annotating, 401
  arrays of, 126
  casting, 232
  exceptions and, 237–238
  in JVM, 228–231
  invariant, 223, 225
  lambda expressions and, 225
  reflection and, 238–241
  restrictions on, 231–238
GenericArrayType interface, 240
get method
  of Array, 186
  of ArrayList, 49
  of BitSet, 261
  of Field, 181, 183, 311
  of Future, 351, 353
  of List, 250
  of LongAccumulator, 374
  of Map, 255–256
  of Optional, 283–285
  of Path, 314
  of Preferences, 460
  of ServiceLoader.Provider, 178
  of Supplier, 129
GET requests, 321
getAnd*Xxx* methods (Atomic*Xxx*), 373
getAnnotation, getAnnotationsByType methods
  of AnnotatedConstruct, 414
  of AnnotatedElement, 411–413
getAs*Xxx* methods
  of Optional*Xxx*, 295
  of *Xxx*Supplier, 130
getAudioClip method (Applet), 174
getAvailableCurrencies method (Currency), 448
getAvailableIds method (ZoneId), 430
getAvailableLocales method (Locale), 445
getAverage method (*Xxx*SummaryStatistics), 287
getBoolean method
  of Array, 186
  of Field, 181, 183
  of FileChannel, 311
  of Preferences, 460

getBundle method (ResourceBundle), 456–458
getByte method
  of Array, 186
  of Field, 181, 183
  of FileChannel, 311
getByteArray method (Preferences), 460
getCanonicalName method (Class), 171–172
getChar method
  of Array, 186
  of Field, 181, 183
  of FileChannel, 311
getClass method (Object), 151, 158, 160, 170, 233, 239
getClassLoader method (Class), 173
getComponentType method (Class), 172, 185
getConstructor(s) methods (Class), 173, 179, 182, 239
getContents method (ListResourceBundle), 457
getContextClassLoader method (Thread), 176–177
getCountry method (Locale), 289
getCurrencyInstance method (NumberFormat), 83, 447
getDayOf*Xxx* methods
  of LocalDate, 63, 425–427
  of ZonedDateTime, 432
getDeclaredAnnotation*Xxx* methods (AnnotatedElement), 411–413
getDeclaredConstructor(s) methods (Class), 173, 179, 239
getDeclaredField(s) methods (Class), 173
getDeclaredMethod(s) methods (Class), 173, 182
getDeclaringClass method
  of Class, 172
  of Enum, 167
getDefault method (Locale), 445–446
getDisplayDefault method (Locale), 456
getDisplayName method
  of Currency, 449
  of DayOfWeek, 435, 450
  of Locale, 446
  of Month, 435, 450
getDouble method
  of Array, 186
  of Field, 181, 183
  of FileChannel, 311
  of Preferences, 460
getElementsAnnotatedWith method (RoundEnvironment), 414
getEnclosedElements method (TypeElement), 414
getEnclosing*Xxx* methods (Class), 172
getEngine*Xxx* methods (ScriptEngineManager), 468
getEnumConstants method (Class), 239
getErrorStream method (Process), 386–387
getFactory method (ScriptEngine), 469
getField(s) methods (Class), 173, 179
getFileName method (Path), 314
getFilePointer method (RandomAccessFile), 311
getFirstDayOfWeek method (Calendar), 451
getFloat method
  of Array, 186
  of Field, 181, 183
  of FileChannel, 311
  of Preferences, 460
getHead method (Formatter), 214
getHeaderFields method (URLConnection), 320
getInputStream method
  of Process, 386
  of URL, 320
  of URLConnection, 321
getInstance method
  of Collator, 452
  of Currency, 448
getInstant method (LogRecord), 214
getInt method
  of Array, 186
  of Field, 181, 183
  of FileChannel, 311
  of Preferences, 460
getInterfaces method (Class), 172
getISO*Xxx* methods (Locale), 445
getLength method (Array), 186
getLevel method (LogRecord), 214
getLogger method (System), 207–208
getLoggerName method (LogRecord), 214

getLong method
  of Array, 186
  of Field, 181, 183
  of FileChannel, 311
  of Preferences, 460
getLongThreadID method (LogRecord), 214
getMax method (*Xxx*SummaryStatistics), 287
getMessage method (LogRecord), 214
getMethod(s) methods (Class), 173, 179, 182
getMethodCallSyntax method (ScriptEngineFactory), 471
getMinute method
  of LocalTime, 429
  of ZonedDateTime, 432
getModifiers method
  of Class, 172
  of Constructor, 179
  of Executable, 183
  of Field, 179, 183
  of Method, 179
getMonth method
  of LocalDate, 426
  of ZonedDateTime, 432
getMonthValue method
  of LocalDate, 63, 426
  of ZonedDateTime, 432
getName method
  of Class, 171–172
  of Constructor, 179
  of Executable, 183
  of Field, 179, 183
  of Method, 179
  of Parameter, 183
  of Path, 314
  of PropertyDescriptor, 184
  of System.Logger, 209
getNano method
  of LocalTime, 429
  of ZonedDateTime, 432
getNumberInstance method (NumberFormat), 447
getObject method (ResourceBundle), 457
getOffset method (ZonedDateTime), 433
getOrDefault method (Map), 256
getOutputStream method
  of Process, 386
  of URLConnection, 320
getPackage method (Class), 172
getPackageName method (Class), 173
getParameters method
  of Executable, 180, 183
  of LogRecord, 214
getParent method (Path), 314
getPath method (FileSystem), 319
getPercentInstance method (NumberFormat), 83, 447
getPermittedSubclasses method (Class), 172
getProperties method (System), 260
getProperty method (System), 175, 204, 259
getPropertyDescriptors method (BeanInfo), 184
getPropertyType, getReadMethod methods (PropertyDescriptor), 184
getQualifiedName method (TypeElement), 414
getRecordComponents method (Class), 173
getResource method (Class), 174, 455
getResourceAsStream method
  of Class, 173–174
  of Module, 487
getResourceBundle, getResourceBundleName methods (LogRecord), 214
getRoot method (Path), 314
getSecond method
  of LocalTime, 429
  of ZonedDateTime, 432
getSequenceNumber method (LogRecord), 214
getShort method
  of Array, 186
  of Field, 181, 183
  of FileChannel, 311
getSimpleName method
  of Class, 172
  of Element, 414
getSource*Xxx*Name methods (LogRecord), 214
getString method (ResourceBundle), 456
getSuperclass method (Class), 172, 239
getSuppressed method (IOException), 199
getSymbol method (Currency), 449
getSystemJavaCompiler method (ToolProvider), 464

getTail method (Formatter), 214
getTask method (JavaCompiler), 464–465
getThrown method (LogRecord), 214
getType method
  of Field, 179
  of Parameter, 183
getTypeName method (Class), 172
getTypeParameters method (Class), 240
getURLs method (URLClassLoader), 175
getValue method (LocalDate), 63
getWriteMethod method (PropertyDescriptor), 184
getYear method
  of LocalDate, 426
  of LocalTime, 429
  of ZonedDateTime, 432
Goetz, Brian, 347
Gregorian calendar reform, 426
GregorianCalendar class, 436–437
  toZonedDateTime method, 436–437
group method (Matcher, MatchResult), 330–331
grouping, 289
  classifier functions of, 289
  reducing to numbers, 290
groupingBy method (Collectors), 289–292
groupingByConcurrent method (Collectors), 289, 296
GUI (graphical user interface)
  callbacks in, 120–121
  long-running tasks in, 358–359
  missing fonts in, 458

H

H formatting symbol (date/time), 436
h, H conversion characters, 37
\h, \H, in regular expressions, 326
%h pattern variable, 213
handle method (CompletableFuture), 357
Hansen, Per Brinch, 379
hash codes, 162–163
  computing in String class, 162
  formatting for output, 37
hash functions, 162–163, 254
hash maps
  concurrent, 369–370
  weak, 263
hash method (Object), 163
hash tables, 254
hashCode method
  of Arrays, 163
  of Enum, 167
  of Object, 158, 160, 162–163
  of records, 77
HashMap class, 255
  null values in, 258
HashSet class, 254
  readObject, writeObject methods, 337
Hashtable class, 378
hasNext method
  declaring, 107
  of Iterator, 252
  of Scanner, 35, 308
hasNext*Xxx* methods (Scanner), 35, 308
headMap method (SortedMap), 265
headSet method
  of NavigableSet, 255
  of SortedSet, 255, 265
heap pollution, 232–233, 266
Hello, World! program, 2
  modular, 478–480
helper methods, 228
hexadecimal numbers, 12–13
  formatting for output, 37
higher method (NavigableSet), 255
higher-order functions, 135–137
h*n*, hr elements (HTML), 96
Hoare, Tony, 379
HTML (HyperText Markup Language)
  generating documentation in, 417
  including code in, 34
HTTP connections, 320–323
HTTP/2 support, 320
HttpClient class, 320–323
  enabling logging for, 323
  newBuilder, newHttpClient methods, 321, 353
HttpHeaders class, 323
HttpResponse class, 322–323
HttpURLConnection class, 320–321

hyperlinks
  in documentation comments, 98
  regular expressions for, 323

I

[I prefix, 159, 171
IANA (Internet Assigned Numbers Authority), 430
IDE (integrated development environment), 4–5
identity method
  of Function, 129, 287
  of UnaryOperator, 129
identity values, 292
if statement, 38–39
ifPresent, ifPresentOrElse methods (Optional), 281
IllegalArgumentException, 204
IllegalStateException, 287, 371
ImageIcon class, 174
images, locating, 174
img element (HTML), 96
immutability, 364
immutable classes, 365
implements keyword, 108
import statement, 7, 88–89
  no annotations for, 402
  static, 89–90
import static statement, 170
importPreferences method (Preferences), 461
InaccessibleObjectException, 181, 486
increment method (LongAdder), 374
increment operator, 20
incrementAndGet method (Atomic*Xxx*), 373
incrementExact method (Math), 20
indexOf method
  of List, 250
  of String, 29
indexOfSubList method (Collections), 251
IndexOutOfBoundsException, 204
info method (ProcessHandle), 390
inheritance, 144–166
  classes win rule, 157, 163
  default methods and, 157
@Inherited annotation, 407–408
initCause method (Throwable), 202
initialization blocks, 74–75
  static, 81
inlining, 151
inner classes, 91–93
  anonymous, 138
  capturing this references in, 126
  invoking methods of outer classes, 93
  local, 133, 137–138
  syntax for, 94
input
  reading, 35–36, 307–308
  redirecting, 469
  setting locales for, 447
  splitting along delimiters, 331
input prompts, 36
input streams, 302
  copying, 304
  obtaining, 302
  reading from, 303
inputReader method (Process), 387
InputStream class, 303
  transferTo method, 304
InputStreamReader class, 307
INSTANCE instance (enum types), 339
instance methods, 6, 68–69
instance variables, 67, 69
  abstract classes and, 152
  annotating, 400
  comparing, 161
  default values of, 73–74
  final, 75
  in records, 77–78
  initializing, 74–75, 147
  not accessible from static methods, 82
  of deserialized objects, 339–341
  protected, 152
  setting, 72
  transient, 336
  vs. local, 74
instanceof operator, 110, 149, 160–161
  annotating, 402
  with pattern matching, 111–112
instances, 2, 6
Instant class, 422
  compareTo method, 423
  equals method, 423

from method, 436
immutability of, 365, 424
legacy classes and, 437
minus, minus*Xxx* methods, 424
now method, 423
plus, plus*Xxx* methods, 424
instruction reordering, 361
int type, 11–12
functional interfaces for, 130
processing values of, 128
random number generator for, 7, 41
streams of, 294
type conversions of, 21–22
using class literals with, 171
IntBinaryOperator interface, 130
IntConsumer interface, 128, 130
Integer class, 49
compare method, 118
MAX_VALUE, MIN_VALUE constants, 11
parseInt method, 28, 194
toString method, 28
unsigned division in, 12
*xxx*Unsigned methods, 21
integer indicator, in string templates, 453
integer types, 11–12
comparing, 118
computing, 19–20
formatting for output, 37
in hexadecimal notation, 12
reading/writing, 310–311
type conversions of, 21–22
values of:
even/odd, 19
signed, 12
@interface declaration, 404–405
interface keyword, 107
sealed, 154
interface methods, 114–116
interfaces, 106–113
annotating, 400–401
compatibility of, 115
declarations of, 107–108
defining variables in, 113
documentation comments for, 95
evolution of, 114
extending, 112
functional, 123–124, 406, 408
implementing, 108–109
in scripting engines, 471
multiple, 113
methods of, 108–109
nested, enumerating, 179–180
no instance variables in, 113
no redefining methods of the Object class in, 163
views of, 264
Internet Engineering Task Force, 444
interrupted method (Thread), 382
interrupted status, 382
InterruptedException, 381, 383
intersects method (BitSet), 261
IntFunction interface, 130, 232
IntPredicate interface, 130
intrinsic locks, 377–379
ints method (RandomGenerator), 294
IntSequence interface, 109, 137
IntStream class, 294–295
mapToObj method, 277
parallel method, 295
IntSummaryStatistics class, 287, 295
IntSupplier, IntTo*Xxx*Function, IntUnaryOperator interfaces, 130
InvalidClassException, 341
InvalidPathException, 313
Invocable interface, 470
InvocationHandler interface, 186
invoke method (Method), 182–183
invokeAll, invokeAny methods (ExecutorService), 352
IOException, 194, 308
addSuppressed, getSuppressed methods, 199
isAbstract method (Modifier), 173, 179
isAfter method
of LocalDate, 426
of LocalTime, 429
of ZonedDateTime, 433
isAlive method
of Process, 389
of ProcessHandle, 390
isAnnotation method (Class), 172
isAnonymousClass method (Class), 172
isArray method (Class), 172, 185

isAssignableFrom method (Class), 173
isBefore method
  of LocalDate, 426
  of LocalTime, 429
  of ZonedDateTime, 433
isCancelled method (Future), 351
isDirectory method (Files), 314, 316
isDone method
  of CompletableFuture, 354
  of Future, 351
isEmpty method
  of BitSet, 261
  of Collection, 249
  of Map, 257
isEnum method (Class), 172
isEqual method (Predicate), 129–130
isFinite, isInfinite methods (Double), 13
isInstance method (Class), 173
isInterface method (Modifier), 173, 179
isInterrupted method (Thread), 382
isLeapYear method (LocalDate), 426
isLocalClass method (Class), 172
isLoggable method
  of Filter, 213
  of System.Logger, 209
isMemberClass method (Class), 172
isNamePresent method (Parameter), 183
isNative method (Modifier), 173, 179
isNegative method (Duration), 424
isNull method (Objects), 125
ISO 8601 format, 408
ISO 8859-1 encoding, 306, 309
isPresent method (Optional), 283–285
isPrimitive method (Class), 172
isPrivate, isProtected, isPublic methods (Modifier), 173, 179
isRecord method (Class), 172
isRegularFile method (Files), 314, 316
isSealed method (Class), 172
isStatic, isStrict, isSynchronized methods (Modifier), 173, 179
isSynthetic method (Class), 172
isVolatile method (Modifier), 173, 179
isZero method (Duration), 424
Iterable interface, 252–253, 314
  iterator method, 252
iterate method (Stream), 274, 279, 294, 367
Iterator interface
  next, hasNext methods, 252
  remove, removeIf methods, 253
iterator method
  of Collection, 249
  of ServiceLoader, 178
  of Stream, 286
iterators, 252–253, 286
  converting to streams, 275, 296
  invalid, 253
  traversing, 178
  weakly consistent, 368

J

j.u.l. *See* java.util.logging package
JAR files, 85
  dependencies in, 497
  for split packages, 488
  manifest for, 490
  modular, 488–489
  processing order of, 87
  resources in, 174, 455
  scanning for deprecated elements, 407
jar program, 85
  -C option, 488
  -d option, 488
  --module-version option, 488
Java EE platform, 353
Java Persistence Architecture, 397
Java Platform Module System, 475
  layers in, 489
  migration to, 489–491
  no support for versioning in, 477, 480, 488
  service loading in, 496–497
java program, 4
  --add-exports, --add-opens options, 492
  --add-module option, 489
  -cp (--class-path, -classpath) option, 86–87
  -da (-disableassertions) option, 205
  -ea (-enableassertions) option, 205
  -esa (-enablesystemassertions) option, 205
  --illegal-access option, 492

-m, -p (--module, --module-path) options, 479, 488
option files for, 492–493
option names in, 85
specifying locales in, 446
Java programming language
compatibility with older versions of, 156–157, 228
online API documentation on, 29–31
strongly typed, 15
Unicode support in, 31–33
uniformity of, 3, 116
java.awt package, 88, 477
java.awt.geom package, 336
java.base module, 481
java.class.path system property, 260
java.desktop module, 480
java.home system property, 260
java.io.tmpdir system property, 260
java.lang, java.lang.annotation packages, 406
java.lang.reflect package, 179
java.logging module, 494
java.sql package, 437
java.time package, 421–437
java.util package, 7, 368
java.util.concurrent package, 368, 371
java.util.concurrent.atomic package, 373
java.util.logging package, 206–211
java.util.random package, 106
java.version system property, 260
JavaBeans, 183–184
javac program, 4
-author option, 99
-cp (--class-path, -classpath) option, 86
-d option, 85, 99
-encoding option, 458
-link, -linksource options, 99
-parameters option, 180
-processor option, 413
-version option, 99
-XprintRounds option, 417
JavaCompiler.getTask method, 464–465
javadoc program, 95–100
including annotations in, 408
JavaFileObject interface, 464
JavaFX platform, 121, 359
java*n*.log files, 211
JavaScript programming language
accessing classes of, from Java, 471
delimiters in, 470
semicolons in, 470
JavaServer Faces framework, 258
javax.annotation package, 406
javax.swing package, 480
JAXB (Java Architecture for XML Binding), 485
jconsole program, 211
jdeprscan program, 407
jdeps program, 497
JDK (Java Development Kit), 4
obsolete features in, 476
JEP 246 (platform logging API), 206
jlink program, 498
jmod program, 499
job scheduling, 263
join method
of String, 25
of Thread, 381
joining method (Collectors), 286–287
JPA (Java Persistence API), 485
JShell (Java Shell tool), 7–11
imported packages in, 10–11
loading modules into, 489
JSON (JavaScript Object Notation), 153–156
JSP (JavaServer Pages), 472
JSR 223 support, 468
JUnit framework, 397–398

K

K formatting symbol (date/time), 436
\k, in regular expressions, 327
key/value pairs
adding new keys to, 255
in annotations, 398–399, 405
removed by garbage collector, 263
values of, 255
keys method (Preferences), 460
keySet method
of ConcurrentHashMap, 372
of Map, 257, 264

keywords, 15
  contextual, 156

L

L suffix, 12
[L prefix, 171
L64X128MixRandom algorithm, 106
lambda expressions, 121–124
  annotating targets for, 408
  capturing variables in, 132–134
  executing, 127
  for loggers, 208
  generic types and, 225
  parameters of, 122
  processing, 127–131
  return type of, 123
  scope of, 131–132
  this reference in, 132
  throwing exceptions in, 196
  using with streams, 276, 366
language codes, 289, 443–444
language model API, 414–415
last method (SortedSet), 255
lastDayOf*Xxx* methods (TemporalAdjusters), 428
lastIndexOf method
  of List, 250
  of String, 29
lastIndexOfSubList method (Collections), 251
lastInMonth method (TemporalAdjusters), 428
lazy operations, 273, 276, 279, 332
leap seconds, 422
leap years, 426
legacy code, 436–437
length method
  of arrays, 47
  of RandomAccessFile, 311
  of String, 6, 32
.level suffix, 210
lib/modules file, 499
limit method (Stream), 278, 296
line feed, 34
  character literal for, 14
  formatting for output, 37
  in regular expressions, 328
line.separator system property, 260
lines method
  of BufferedReader, 308
  of Files, 275, 297, 307
@link tag (javadoc), 98
linked lists, 248, 253
LinkedBlockingQueue class, 371, 381
LinkedHashMap class, 258
LinkedList class, 248
List interface, 226, 248–249
  add, addAll, get, indexOf, lastIndexOf, listIterator methods, 250
  of method, 49, 51, 250, 264
  remove, replaceAll, set, sort methods, 250
  subList method, 250, 265
list method (Files), 316–317
ListIterator interface, 253
ListResourceBundle class, 457
lists
  converting to streams, 296
  mutable, 265
  printing elements of, 125
  removing null values from, 125
  sublists of, 265
  unmodifiable views of, 266
literals
  character, 14
  floating-point, 13
  integer, 12
  string, 26–27, 33
little-endian format, 305
load balancing, 334
load method (ServiceLoader), 178, 497
loadClass method (ClassLoader), 175
local classes, 137–138
local date/time, 424–430
local variables, 45–46
  annotating, 400–401
  vs. instance, 74
LocalDate class, 63
  datesUntil method, 426–427
  get*Xxx* methods, 63, 425–427
  is*Xxx* methods, 426
  legacy classes and, 437
  minus, minus*Xxx* methods, 425, 427
  now method, 72, 82, 425

of method, 63, 72, 425–426
ofInstant method, 425
parse method, 450
plus, plusXxx methods, 63–64, 66, 425, 427
toEpochSecond method, 426
until method, 426–427
withXxx methods, 425
LocalDateTime class, 430
atZone method, 430
legacy classes and, 437
parse method, 450
Locale class, 288
forLanguageTag method, 446
getAvailableLocales method, 445
getCountry method, 289
getDefault method, 445–446
getDisplayDefault method, 456
getDisplayName method, 446
getISOXxx methods, 445
predefined fields, 445
setDefault method, 445–446
locales, 287–291, 442–447
date/time formatting for, 449–451
default, 434, 445–446, 449–450, 456
displaying names of, 446
first day of week in, 451
for template strings, 453–454
formatting styles for, 435, 450
sorting words for, 451–452
specifying, 443–445
weekdays and months in, 435
LocalTime class, 429–430
final, 151
getXxx, isXxx methods, 429
legacy classes and, 437
minus, minusXxx, now, of, ofInstant methods, 429
parse method, 450
plus, plusXxx, toXxx, withXxx methods, 429
lock method
of FileChannel, 312
of ReentrantLock, 376
locks, 364
error-prone, 365
intrinsic, 377–379
reentrant, 375–377
releasing, 199, 362
log handlers, 211–213
default, 211
filtering/formatting, 213
levels of, 211
Log4j framework, 206
Logback framework, 206
Logger class (java.util.logging), 494
Logger interface (System), 207–209
getName method, 209
isLoggable method, 209
log method, 207–209
loggers
filtering/formatting, 213
hierarchy of, 210
naming, 207
logging, 206–214
configuring, 209–211
failures, 201
levels of, 208–211
overriding methods for, 151
LogRecord class, methods of, 214
Long class, 49
MAX_VALUE, MIN_VALUE constants, 11
unsigned division in, 12
xxxUnsigned methods, 21
long indicator, in string templates, 453
long type, 11–12
atomic operations on, 374–375
functional interfaces for, 130
streams of, 294
type conversions of, 21–22
LongAccumulator class, 374
accumulate, get methods, 374
LongAdder class, 374–375
add, increment, sum methods, 374
threadsafe, 377
LongConsumer, LongXxxOperator, LongPredicate, LongSupplier, LongToXxxFunction interfaces, 130
LongFunction interface, 130, 232
longs method (RandomGenerator), 294
LongStream class, 294–295
LongSummaryStatistics class, 287, 295
long-term persistence, 340

Lookup class, 487
lookup method (MethodHandles), 487
loops, 41–43
  exiting, 43–44
  infinite, 43
lower method (NavigableSet), 255

M

m, M formatting symbols (date/time), 436
main method, 2, 6
  decomposing, 56–57
  string array parameter of, 52
ManagedExecutorService class, 353
Map interface, 250
  clear method, 257
  compute method, 256
  computeIf*Xxx* methods, 256–257
  contains*Xxx* methods, 257
  entrySet method, 257–258
  forEach method, 257
  get, getOrDefault methods, 255–256
  isEmpty method, 257
  keySet method, 257, 264
  merge method, 256
  of method, 257, 264
  ofEntries method, 264
  put method, 255–256
  putAll method, 257
  putIfAbsent method, 256
  remove method, 257
  replace, replaceAll methods, 257
  size method, 257
  values method, 257, 264
map method
  of Optional, 282
  of Stream, 276
mapMulti method (Stream), 278
mapping method (Collectors), 290
maps, 255–258
  concurrent, 257, 288
  empty, 257
  iterating over, 258
  of stream elements, 287–288, 296
  order of elements in, 258
  views of, 257
    unmodifiable, 266
mapToInt method (Stream), 293
mapToObj method (IntStream), 277
mapTo*Xxx* methods (*Xxx*Stream), 294
marker interfaces, 165
Matcher class, 329–331
  methods of, 332
matcher, matches methods (Pattern), 329
MatchResult interface, 330–332
Math class
  E constant, 20
  floorMod method, 20
  max, min methods, 20
  PI constant, 20, 80, 89
  pow method, 20, 81, 89
  round method, 22
  sqrt method, 20
  *xxx*Exact methods, 20, 22
max method
  of Stream, 280
  of *Xxx*Stream, 295
MAX_VALUE constant (integer classes), 11
maxBy method
  of BinaryOperator, 129
  of Collectors, 290
medium indicator, in string templates, 453
memory
  allocating, 364
  caching, 361
  concurrent access to, 361
memory-mapped files, 311
merge method
  of ConcurrentHashMap, 369–370
  of Map, 256
Message class, 165–166
MessageFormat class, 453–454
meta-annotations, 404–410
META-INF/MANIFEST.MF file, 490
META-INF/services directory, 496
method calls, 6
  receiver of, 69
Method class, 179–180
  getModifiers, getName methods, 179
  invoke method, 182–183
method expressions, 124, 150
method references, 124–126, 233
  annotating, 402

MethodHandles.lookup method, 487
methods, 2
  abstract, 123, 151–152
  accessor, 64, 77
  annotating, 236, 400
  atomic, 369
  body of, 68
  chaining calls of, 64
  clashes of, 236–237
  compatible, 162
  declarations of, 67
  default, 114–116
  deprecated, 97, 406–407
  documentation comments for, 95, 97
  enumerating, 179–180
  factory, 72, 83
  final, 150, 366
  for throwing exceptions, 203–204
  header of, 67
  inlining, 151
  instance, 68–69
  invoking, 182
  modifying functions, 135
  mutator, 64, 266, 366
  naming, 15–16, 77
  native, 81
  overloading, 73, 125
  overriding, 114, 145–147, 151, 195–196, 406–407
  parameters of, 180
    null checks for, 203
  passing arrays into, 56
  private, 117
  proxied, 187
  public, 108–109, 179
  restricted to subclasses, 152–153
  return value of, 2, 68
  returning functions, 135
  static, 56, 81–83, 90, 113–114
  storing in variables, 7
  symmetric, 161
  synchronized, 377–380
  used for serialization, 406–407
  utility, 87
  variable number of arguments of, 57
Microsoft Notepad, 306
Microsoft Windows
  line ending in, 34
  path separator in, 86, 260
  registry in, 459
min method
  of Math, 20
  of Stream, 280
  of *Xxx*Stream, 295
MIN_VALUE constant (integer classes), 11
minBy method
  of BinaryOperator, 129
  of Collectors, 290
minus, minus*Xxx* methods
  of Duration, 424
  of Instant, 424
  of LocalDate, 425, 427
  of LocalTime, 429
  of ZonedDateTime, 432
Modifier interface
  is*Xxx* methods, 173, 179
  toString method, 173
modifiers, checking, 179
module keyword, 479
module path, 479, 488, 490–491
Module.getResourceAsStream method, 487
module-info.class file, 479, 488
module-info.java file, 479
modules, 475
  aggregator, 494
  annotating, 480
  automatic, 489–491
  bundling up the minimal set of, 498
  declaration of, 478–479
  documentation comments for, 95, 99
  explicit, 491
  illegal access to, 492
  inspecting files in, 499
  loading into JShell, 489
  naming, 478, 490
  open, 486
  reflective access for, 180–181
  required, 480–482, 493–495
  tools for, 497–499
  transitive, 493–495
  unnamed, 491
  versioning and, 477, 480, 488

monitors (classes), 379
Month enumeration, 425–426, 432
  getDisplayName method, 435, 450
MonthDay class, 427
move method (Files), 315–316
multiplication, 19
multipliedBy method (Duration), 424
mutators, 64
  unmodifiable views and, 266

N

n
  conversion character, 37
  formatting symbol (date/time), 436
\n (line feed)
  for character literals, 14
  in property files, 259–260
  in regular expressions, 325–326, 333
name method (Enum), 167
NaN (not a number), 13
native methods, 81
naturalOrder method (Comparator), 136
navigable maps/sets, 266
NavigableMap interface, 372
NavigableSet interface, 249, 254, 265
  methods of, 255
nCopies method (Collections), 249, 251
negate method (Predicate, BiPredicate), 129
negated method (Duration), 424
negateExact method (Math), 20
NEGATIVE_INFINITY value (Double), 13
negative values, 11
nested classes, 90–95
  annotating, 402
  enumerating, 179–180
  inner, 91–93
  public, 91
  static, 90–91
new operator, 7, 15, 18, 72
  as constructor reference, 126
  for anonymous classes, 138
  for arrays, 46–47, 54
newBufferedReader method (Files), 308, 468
newBufferedWriter method (Files), 308, 316
newBuilder method (HttpClient), 321, 353
newCachedThreadPool method (Executors), 349
newFileSystem method (FileSystems), 319
newHttpClient method (HttpClient), 321, 353
newInputStream method (Files), 302, 316, 334
newInstance method
  of Array, 186
  of Class, 182, 239
  of Constructor, 182–183
newKeySet method (ConcurrentHashMap), 372
newline. *See* line feed
newOutputStream method (Files), 302, 316, 334
newProxyInstance method (Proxy), 187
next method
  declaring, 107
  of Iterator, 252
  of Scanner, 35
  of TemporalAdjusters, 428
nextClearBit method (BitSet), 261
nextDouble method
  common for all generators, 106
  of Scanner, 35, 308
nextInt method
  common for all generators, 106
  of Random, 7, 41
  of Scanner, 35
nextLine method (Scanner), 35
nextOrSame method (TemporalAdjusters), 428
nextSetBit method (BitSet), 261
nominal typing, 128
noneMatch method (Stream), 280
noneOf method (EnumSet), 262
noninterference, of stream operations, 275
@NonNull annotation, 401
non-sealed modifier, 156
normalize method (Path), 313
Normalizer class, 453
NoSuchElementException, 283, 371
notify, notifyAll methods (Object), 380–381
now method
  of Instant, 423
  of LocalDate, 72, 82, 425
  of LocalTime, 429
  of ZonedDateTime, 432

null value, 27, 66
as default value, 73, 76
checking parameters for, 203
comparing against, 160
converting to strings, 159
NullPointerException, 27, 48, 66, 74, 194, 203, 256
vs. Optional, 280
nullsFirst, nullsLast methods (Comparator), 136
Number class, 448
number indicator, in string templates, 453
NumberFormat class
getXxxInstance methods, 83, 447
parse method, 448
setCurrency method, 448
NumberFormatException, 194
numbers
average of, 108–109
big, 24
comparing, 118
converting to strings, 28
default value of, 73, 76
even or odd, 19
formatting, 37, 442, 447, 453
from grouped elements, 290
in regular expressions, 326
non-negative, 205, 260
printing, 36
random, 7, 41, 106, 274, 278, 294, 384
reading/writing, 308, 310–311
rounding, 14, 22
type conversions of, 21–22
unsigned, 12, 21
with fractional parts, 13–14

O

o conversion character, 37
Object class, 157–166
clone method, 153, 158, 163–166, 182
equals method, 158–162
finalize method, 158
getClass method, 151, 158, 160, 170, 233, 239
hashCode method, 158, 160, 162–163
notify, notifyAll methods, 380–381
toString method, 158–159
wait method, 379–381
object references, 65–66
attempting to change, 71
comparing, 159
default value of, 73, 76
null, 66
passed by value, 71
serialization and, 335
ObjectInputStream class, 334–335
defaultReadObject method, 337, 341
readDouble method, 337
readFields method, 341
readObject method, 335–343
ObjectInputValidation interface, 342–343
object-oriented programming, 61–102
encapsulation in, 475–476
ObjectOutputStream class, 334
defaultWriteObject method, 336–337
writeDouble method, 337
writeObject method, 334–337
object-relational mappers, 485
objects, 2, 62–66
calling methods on, 7
casting, 110–111
cloning, 163–166
comparing, 50, 159–162
constructing, 7, 71–76, 182–183
converting:
to JSON, 486
to strings, 158–159
deep/shallow copies of, 164–165
deserialized, 339–341
immutable, 64
initializing variables with, 15
inspecting, 180–181
invoking static methods on, 82
mutable, 75
serializable, 334–335
sorting, 117–119
state of, 62
Objects class
checkIndex method, 204
converting to streams, 274
equals method, 161
hash method, 163

isNull method, 125
requireNonNull, requireNonNullXxx methods, 203–204
ObjXxxConsumer interfaces, 130
octal numbers, 12
formatting for output, 37
octonions, 32
odd numbers, 19
of method
of EnumSet, 262
of IntStream, 294
of List, 49, 51, 250, 264
of LocalDate, 63, 72, 425–426
of LocalTime, 429
of Map, 257, 264
of Optional, 284
of Path, 312, 314, 319
of ProcessHandle, 389
of Set, 264
of Stream, 273–274
of ZonedDateTime, 430–432
ofDateAdjuster method (TemporalAdjusters), 428
ofDays method
of Duration, 423–424, 426, 431
of Period, 431
ofEntries method (Map), 264
offer method (BlockingQueue), 371
offsetByCodePoints method (String), 33
OffsetDateTime class, 433
ofHours method (Duration), 423–424
ofInstant method
of LocalDate, 425
of LocalTime, 429
of ZonedDateTime, 432
ofLocalizedXxx methods (DateTimeFormatter), 433, 449
ofMillis, ofMinutes, ofNanos methods (Duration), 423–424
ofNullable method
of Optional, 284
of Stream, 274, 286
ofPattern method (DateTimeFormatter), 435
ofSeconds method (Duration), 423–424
ofString method (HttpResponse), 322
ofYears method (Period), 426
onExit method
of Process, 389
of ProcessHandle, 390
open keyword, 487
open method (FileChannel), 311
openConnection method (URL), 320
opens keyword, 486
qualified, 495
openStream method (URL), 302
Operation interface, 169
operations
associative, 292
atomic, 364, 369, 373–375, 379
bulk, 370
lazy, 273, 276, 279, 332
parallel, 366–368
performed optimistically, 374
stateless, 295
threadsafe, 368–372
operators, 17–24
cast, 22
precedence of, 18
option files, 492–493
Optional class, 280–285
creating values of, 284
empty method, 284
filter method, 282
flatMap method, 284–285
for empty streams, 292
for processes, 390
get method, 283–285
ifPresent method, 281
ifPresentOrElse method, 281
isPresent method, 283–285
map method, 282
of, ofNullable methods, 284
or method, 282
orElse method, 280
orElseThrow method, 281, 283
proper usage of, 283
stream method, 285–286
OptionalXxx classes, 295
or method
of BitSet, 261
of Predicate, BiPredicate, 129
Oracle JDK, 468

order method (ByteBuffer), 311
ordinal method (Enum), 167
orElseThrow method (Optional), 281, 283
org.omg.corba package, 476
orTimeout method (CompletableFuture), 357
os.arch, os.name, os.version system properties, 260
OSGi (Open Service Gateway Initiative), 477
output
  formatted, 36–38
  redirecting, 469
  setting locales for, 447
  writing, 308–310
output streams, 302
  closing, 304
  obtaining, 302
  writing to, 304
OutputStream class, 334
  write method, 304
OutputStreamWriter class, 308
outputWriter method (Process), 387
@Override annotation, 146, 336, 339, 406–407
overriding, 145–147
  for logging/debugging, 151
overview.html file, 99

P

\p, \P, in regular expressions, 326
package declarations, 83–85
package statement, 84
package-info.java file, 99, 400
packages, 3, 83–90
  accessing, 87–88, 153, 476, 483–484, 486, 490
  adding classes to, 88
  annotating, 400–401
  default, 84
  documentation comments for, 95, 99
  exporting, 482–485, 487
  naming, 83
  not nesting, 84
  split, 488
parallel method (*Xxx*Stream), 295
parallel streams, 366–367
parallelStream method (Collection), 249, 272–273, 295, 366
parallel*Xxx* methods (Arrays), 52, 367
@param tag (javadoc), 97
Parameter class, 183
parameter variables, 70
  annotating, 400
  scope of, 45
ParameterizedType interface, 240
parentLogger method (Driver), 494
parse method
  of DateTimeFormatter, 435
  of Local*Xxx*, ZonedDateTime, 450
  of NumberFormat, 448
Parse.quote method, 324
parseDouble method (Double), 28
ParseException, 448
parseInt method (Integer), 28, 194
partitioning, 365
partitioningBy method (Collectors), 289, 292
Pascal triangle, 54
passwords, 36
Path interface, 114, 312–314
  get method, 314
  get*Xxx* methods, 314
  normalize method, 313
  of method, 312, 314, 319
  relativize method, 313
  resolve, resolveSibling methods, 313
  subpath method, 314
  toAbsolutePath method, 314
  toFile method, 314
path separators, 313
path.separator system property, 260
paths, 312
  absolute vs. relative, 312–314
  combining, 314
  filtering, 317
  resolving, 313
  taking apart, 314
Paths class, 114
Pattern class
  asMatchPredicate, asPredicate methods, 329
  compile method, 329, 333

flags, 333
matcher, matches methods, 329
split method, 331
splitAsStream method, 275, 332
pattern variables, 213
PECS (producer extends, consumer super), 226
peek method
of BlockingQueue, 371
of Stream, 279
percent indicator, in string templates, 453
performance
atomic operations and, 374
big numbers and, 24
combined operators and, 20
memory caching and, 361
Period class
ofDays method, 431
ofYears method, 426
plusYears method, 426
permits keyword, 155–156
@Persistent annotation, 409
PHP, scripting engine for, 468
PI constant (Math), 20, 80, 89
placeholders, 453–454
platform class loader, 174
platform encoding, 306, 458
platform logging API, 206–210
plugins, loading, 175
plus, plus*Xxx* methods
of Duration, 424
of Instant, 424
of LocalDate, 63–64, 66, 425, 427
of LocalTime, 429
of ZonedDateTime, 431–432
plusYears method (Period), 426
Point class, 158–159
poll method (BlockingQueue), 371
poll*Xxx* methods (NavigableSet), 255
pools, for parallel streams, 297
pop method (ArrayDeque), 262
POSITIVE_INFINITY value (Double), 13
POST requests, 322
postVisitDirectory method (FileVisitor), 318
pow method (Math), 20, 81, 89
predefined character classes, 324, 326, 328
predicate functions, 289
Predicate interface, 124, 129
and method, 129
isEqual method, 129–130
or, negate methods, 129
test method, 129, 225
Preferences class, 459–461
childrenNames method, 460
exportSubtree method, 460
get, get*Xxx* methods, 460
importPreferences method, 461
keys method, 460
put, put*Xxx* methods, 460
remove, removeNode methods, 460
system*Xxx* methods, 459–460
user*Xxx* methods, 459–460
previous method
of ListIterator, 253
of TemporalAdjusters, 428
previousClearBit method (BitSet), 261
previousOrSame method (TemporalAdjusters), 428
previousSetBit method (BitSet), 261
preVisitDirectory method (FileVisitor), 318
primitive types, 11–14
comparing, 161
converting to strings, 159
functions interfaces for, 130
passed by value, 71
streams of, 293–294
type parameters and, 231
variables of, no updating for, 70
wrapper classes for, 49–50
printStackTrace method (Throwable), 203
PrintStream class, 6, 159, 309
print method, 6, 36, 206, 309
printf method, 36–37, 57, 309
println method, 6, 35–36, 52, 125, 309
PrintWriter class, 309
close method, 197–198
print method, 309
printf method, 309, 447
println method, 309
priority queues, 263

private modifier, 3, 87
  for enum constructors, 168
Process class, 386–390
  destroy, destroyForcibly methods, 389
  errorReader method, 387
  exitValue method, 389
  getErrorStream method, 386–387
  getInputStream, getOutputStream methods, 386
  inputReader method, 387
  isAlive method, 389
  onExit method, 389
  outputWriter method, 387
  supportsNormalTermination method, 389
  toHandle method, 389
  waitFor method, 388–389
ProcessBuilder class, 386–390
  directory method, 386
  redirect*Xxx* methods, 387
  start, startPipeline methods, 388
processes, 386–390
  building, 386–388
  getting info about, 389–390
  killing, 389
  running, 388–389
ProcessHandle interface, 389–390
  allProcesses method, 389
  current method, 389
  destroy, destroyForcibly methods, 390
  info method, 390
  isAlive method, 390
  of method, 389
  onExit method, 390
  supportsNormalTermination method, 390
processing pipeline, 355, 388
Processor interface, 413
Programmer's Day, 426
programming languages
  functional, 105
  object-oriented, 2
  scripting, 467
programs
  compiling, 4
  configuration options for, 259
  localizing, 441–461
  packaging, 499
  responsive, 358
  running, 4
  testing, 204
promises (in concurrent libraries), 354
properties, 183–184
  loading from file, 259
  naming, 184
  read-only/write-only, 184
  testing for, 225
Properties class, 259–260
.properties extension, 455
property files
  encoding, 259, 457
  generating, 417
  localizing, 455–457
protected modifier, 152–153
Provider.get, Provider.type methods, 178
provides keyword, 496
Proxy class, 186–188
  newProxyInstance method, 187
public modifier, 3, 87
  for interface methods, 108–109
  method overriding and, 147
push method (ArrayDeque), 262
put method
  of BlockingQueue, 371
  of FileChannel, 311
  of Map, 255–256
  of Preferences, 460
putAll method (Map), 257
putBoolean method
  of FileChannel, 311
  of Preferences, 460
putByte method (FileChannel), 311
putByteArray method (Preferences), 460
putChar method (FileChannel), 311
putDouble, putFloat methods
  of FileChannel, 311
  of Preferences, 460
putIfAbsent method
  of ConcurrentHashMap, 369
  of Map, 256
putInt, putLong methods
  of FileChannel, 311
  of Preferences, 460
putShort method (FileChannel), 311

Q

\Q, in regular expressions, 324–325
qualified exports, 495
Queue interface, 250, 262
  synchronizing methods in, 379
  using ArrayDeque with, 262
quote method (Parse), 324
quoteReplacement method (Matcher), 332

R

R language, scripting engine for, 468
\r (carriage return)
  for character literals, 14
  in property files, 260
\r, \R, in regular expressions, 325, 328
race conditions, 295, 362–364
Random class, 7, 106
  nextInt method, 7, 41
  threadsafe, 384
random numbers, 7, 41, 106
  in multiple threads, 384
  streams of, 274, 278, 294
RandomAccess interface, 249
RandomAccessFile class, 310–311
  getFilePointer method, 311
  length method, 311
  seek method, 310–311
RandomGenerator interface, 107
  methods of, 294
RandomNumbers class, 82
range method
  of EnumSet, 262
  of *Xxx*Stream, 294
rangeClosed method (*Xxx*Stream), 294
ranges, 265
  converting to streams, 296
raw types, 229, 232–233
read method
  of Files, 304
  of InputStream, 303
  of InputStreamReader, 307
readAll*Xxx* methods (Files), 303, 307
readByte, readChar methods (DataInput), 310
readDouble method
  of DataInput, 310
  of ObjectInputStream, 337
Reader class, 307
readers, 302
readExternal method (Externalizable), 338–339
readFields method (ObjectInputStream), 341
readFloat, readFully methods (DataInput), 310
readInt method (DataInput), 310–311
readLine method
  of BufferedReader, 308
  of Console, 36
readLong method (DataInput), 310
readNBytes method (Files), 303
readObject method
  of HashSet, 337
  of ObjectInputStream, 335–343
readPassword method (Console), 36
readResolve method (Serializable), 339–340
readShort method (DataInput), 310
readUnsigned*Xxx*, readUTF methods (DataOutput), 310
receiver parameters, 69, 403
records, 76–79
  serializable, 342
redirection syntax, 36
redirect*Xxx* methods (ProcessBuilder), 387
reduce method (Stream), 292–293
reduce*Xxx* methods (ConcurrentHashMap), 370
reducing method (Collectors), 291
reductions, 280, 292–293
ReentrantLock class, 375–377
  lock, unlock methods, 376
reflection, 179–188
  generic types and, 234, 238–241
  module system and, 180–181, 485, 492
  processing annotations with, 410–413
  security and, 343
ReflectiveOperationException, 171
regular expressions, 323–333
  flags for, 333
  groups in, 330–331
  replacing matches with, 332
  splitting input with, 331
  testing matches of, 329–330
  turning into predicates, 329

relational operators, 22–23
relativize method (Path), 313
remainderUnsigned method (Integer, Long), 21
remove method
  of ArrayDeque, 262
  of ArrayList, 49
  of BlockingQueue, 371
  of Collection, 249
  of Iterator, 253
  of List, 250
  of Map, 257
  of Preferences, 460
removeAll method (Collection), 249
removeIf method
  of ArrayList, 124
  of Collection, 249
  of Iterator, 253
removeNode method (Preferences), 460
@Repeatable annotation, 407, 409–410
replace method
  of Map, 257
  of String, 29
replaceAll method
  of Collections, 251
  of List, 250
  of Map, 257
  of Matcher, 332
  of String, 332
replaceFirst method (Matcher), 332
requireNonNull, requireNonNull*Xxx* methods (Objects), 203–204
requires keyword, 479, 482–485, 490, 493–495
resolve, resolveSibling methods (Path), 313
resource bundles, 455–458
ResourceBundle class
  extending, 457
  getBundle method, 456–458
  getObject method, 457
  getString method, 456
resources, 170–179
  loading, 174, 487
  managing, 197
resume method (Thread, deprecated), 382
retainAll method (Collection), 249
@Retention annotation, 404, 407
return statement, 56, 68
  in finally blocks, 199
  in lambda expressions, 122
@return tag (javadoc), 97
return types, covariant, 147, 231
return values
  as arrays, 56
  missing, 280
  providing type of, 56
reverse domain name convention, 83, 478
reverse method (Collections), 52, 252
reversed method (Comparator), 136
reverseOrder method (Comparator), 137
RFC 822, RFC 1123 formats, 434
Rhino JavaScript engine, 468, 470
rotate method (Collections), 252
round method (Math), 22
RoundEnvironment interface, 414
roundoff errors, 14
RowSetProvider class, 492
rt.jar file, 499
Ruby, scripting engine for, 468
runAfter*Xxx* methods (CompletableFuture), 357–358
Runnable interface, 120, 129, 349, 351
  executing on the UI thread, 359
  run method, 129, 348, 381, 383
  using class literals with, 171
runtime
  raw types at, 232–233
  safety checks at, 229
Runtime class
  availableProcessors method, 349
  exec method, 386
runtime image file, 499
RuntimeException, 194

S

s formatting symbol (date/time), 436
s, S conversion characters, 37
\s, \S, in regular expressions, 326
safety checks, as runtime, 229
@SafeVarargs annotation, 236, 406, 408
sample code, 5

Scala programming language, 227
Scanner class, 35
  findAll method, 330
  hasNext, hasNext*Xxx* methods, 35, 308
  next, next*Xxx* methods, 35, 308
  tokens method, 275, 308
  useLocale method, 447
scheduling applications
  computing dates for, 428–429
  time zones and, 425, 430
Scheme, scripting engine for, 468
ScriptContext interface, 469
ScriptEngine interface
  createBindings method, 469
  eval method, 468–471
  getFactory method, 469
ScriptEngineFactory interface, 471
ScriptEngineManager class
  getEngine*Xxx* methods, 468
  visibility of bindings in, 469
scripting engines, 468
  compiling code in, 471
  implementing Java interfaces in, 471
scripting languages, 467
  invoking functions in, 470
scripts
  compiling, 471
  evaluating, 468
sealed modifier, 154–156
sealed types, 153–156
search*Xxx* methods (ConcurrentHashMap), 370
security, 88, 342–344
SecurityException, 181
@see tag (javadoc), 98
seek method (RandomAccessFile), 311
sequences, producing, 274
@Serial annotation, 336, 339, 406–407
serial numbers, 335
Serializable interface, 334–335
  readResolve, writeReplace methods, 339–340
serialization, 333–344
  filters for, 343
serialVersionUID instance variable, 341
server-side software, 334
ServiceLoader class, 177–179, 496
  iterator method, 178
  load method, 178, 497
ServiceLoader.Provider interface, 178
services
  configurable, 177
  loading, 177–179, 496–497
ServletException, 201–202
Set interface, 249, 372
  of method, 264
  working with EnumSet, 262
set method
  of Array, 186
  of ArrayList, 49
  of BitSet, 261
  of Field, 183
  of List, 250
  of ListIterator, 253
setAccessible method (AccessibleObject), 181, 183
setAll method (Arrays), 127
setBoolean, setByte, setChar methods
  of Array, 186
  of Field, 183
setClassAssertionStatus method (ClassLoader), 206
setContextClassLoader method (Thread), 176–177
setCurrency method (NumberFormat), 448
setDaemon method (Thread), 385
setDecomposition method (Collator), 452
setDefault method (Locale), 445–446
setDefaultAssertionStatus method (ClassLoader), 206
setDefaultUncaughtExceptionHandler method (Thread), 202
setDoOutput method (URLConnection), 320
setDouble, setFloat, setInt, setLong methods
  of Array, 186
  of Field, 183
setOut method (System), 81
setPackageAssertionStatus method (ClassLoader), 206
setProperty method (System), 210
setReader method (ScriptContext), 469

setRequestProperty method (URLConnection), 320
sets, 254–255
  immutable, 365
  threadsafe, 372
  unmodifiable views of, 266
setShort method
  of Array, 186
  of Field, 183
setStrength method (Collator), 452
setUncaughtExceptionHandler method (Thread), 381
setWriter method (ScriptContext), 469
shallow copies, 164–165
shared variables, 362–365
  atomic mutations of, 373–375
  locking, 375–377
shell
  redirection syntax of, 36
  scripts for, generating, 417
shift operators, 23–24
Shift_JIS encoding, 306
short circuit evaluation, 23
Short class, 49
  MAX_VALUE, MIN_VALUE constants, 11
short indicator, in string templates, 453
short type, 11–12
  streams of, 294
  type conversions of, 21
short-term persistence, 340
shuffle method (Collections), 52, 252
SimpleDateFormat class, 384–385
SimpleFileVisitor class, 318
SimpleJavaFileObject class, 466
@since tag (javadoc), 97
singletons, 339
size method
  of ArrayList, 49
  of Collection, 249
  of Map, 257
skip method (Stream), 278
skipNBytes method (Files), 304
sleep method (Thread), 381, 383
SLF4J (Simple Logging Fasade for Java), 206, 478
SOAP protocol, 477
SocketHandler class, 211
sort method
  of Arrays, 52, 119, 123–124
  of Collections, 52, 226–227, 241, 252
  of List, 250
sorted maps, 265–266
sorted method (Stream), 279
sorted sets, 249, 265
  traversing, 254
  unmodifiable views of, 266
sorted streams, 296
SortedMap interface, 265
SortedSet interface, 249, 254
  first method, 255
  headSet method, 255, 265
  last method, 255
  subSet, tailSet methods, 255, 265
sorting
  array lists, 52
  arrays, 52, 117–119
  chaining comparators for, 136
  changing order of, 135
  streams, 279
  strings, 27–28, 124, 451–452
source code, generating, 406, 408, 415–417
source files
  documentation comments for, 99
  encoding of, 458
  placing, in a file system, 84
  reading from memory, 465
space flag (for output), 38
spaces
  in regular expressions, 326
  removing, 29
split method
  of Pattern, 331
  of String, 26, 332
splitAsStream method (Pattern), 275, 332
spliterator method (Collection), 249
Spliterators.spliteratorUnknownSize method, 275
SQL (Structured Query Language), 34
sqrt method (Math), 20
square root, computing, 284
Stack class, 262

stack trace, 202–203
StackWalker class, 203
standard output, 3
StandardCharsets class, 306
StandardJavaFileManager interface, 464–466
start method
  of Matcher, MatchResult, 330–331
  of ProcessBuilder, 388
  of Thread, 381
startPipeline method (ProcessBuilder), 388
startsWith method (String), 29
stateless operations, 295
statements, combining, 46
static constants, 80–81
static imports, 89–90
static initialization, 175
static methods, 56, 81–83
  accessing static variables from, 82
  importing, 90
  in interfaces, 113–114
static modifier, 2, 16, 56, 79–83, 169
  for modules, 494
static nested classes, 90–91
static variables, 79–80
  accessing from static methods, 82
  importing, 90
  visibility of, 361
stop method (Thread, deprecated), 382
Stream interface
  anyMatch method, 280
  collect method, 286–287, 293
  concat method, 278
  count method, 273, 280
  distinct method, 279, 296
  dropWhile method, 278
  empty method, 274
  filter method, 273–276, 280
  findAny method, 280
  findFirst method, 179, 280
  flatMap method, 277
  forEach, forEachOrdered methods, 286
  generate method, 274, 294
  iterate method, 274, 279, 294, 367
  iterator method, 286
  limit method, 278, 296
  map method, 276
  mapMulti method, 278
  mapToInt method, 293
  max, min methods, 280
  noneMatch method, 280
  of method, 273–274
  ofNullable method, 274, 286
  peek method, 279
  reduce method, 292–293
  skip method, 278
  sorted method, 279
  takeWhile method, 278
  toArray method, 126, 286
  toList method, 275
  unordered method, 296
stream method
  of Arrays, 274, 294
  of BitSet, 261
  of Collection, 249, 272–273
  of Optional, 285–286
  of StreamSupport, 275
streams, 271–275
  collecting elements of, 286–288
  combining, 278
  computing values from, 292–293
  converting to/from arrays, 274, 286, 296, 368
  creating, 273–276
  debugging, 279
  empty, 274, 280, 292
  filtering, 285
  finite, 274
  flattening, 277, 285
  infinite, 273–274, 278–279
  intermediate operations for, 273
  locating services with, 178
  noninterference of, 275
  of primitive type values, 293–294
  of random numbers, 294
  ordered, 296
  parallel, 272, 280, 286, 288–289, 292, 295–297, 366–367
  processed lazily, 273, 276, 279
  reductions of, 280
  removing duplicates from, 279
  returned by Files.lines, 297
  sorting, 279

splitting, 278
summarizing, 287, 295
terminal operation for, 273, 280
transformations of, 276–278, 294
vs. collections, 273
StreamSupport.stream method, 275
String class, 6, 28
charAt method, 32
codePoints method, 32–33, 276–278
codePoint*Xxx* methods, 32
compareTo method, 27–28, 117, 451
compareToIgnoreCase method, 124
contains method, 29
endsWith method, 29
equals method, 26–27
equalsIgnoreCase method, 27
final, 151
format method, 447
formatted method, 38
hash codes, 162
immutability of, 29, 365
indexOf, lastIndexOf methods, 29
join method, 25
length method, 6, 32
offsetByCodePoints method, 33
replace method, 29
replaceAll method, 332
split method, 26, 332
startsWith method, 29
strip method, 448
substring method, 26
toLowerCase method, 29, 276, 447
toUpperCase method, 29, 447
StringBuilder class, 25
strings, 6, 25–34
comparing, 26–28
concatenating, 25, 159
converting:
from byte arrays, 307
from objects, 158–159
to code points, 276
to numbers, 28
empty, 27–28, 159
formatting for output, 37
internal representation of, 33
normalized, 452
sorting, 27–28, 124, 451–452
splitting, 26, 275
templates for, 453–454
transforming to lower/uppercase, 276, 447
StringSource class, 465
StringWriter class, 309
strip method (String), 29, 448
strong element (HTML), 96
subclasses, 145
anonymous, 149–150, 169
calling toString method in, 158
constructors for, 147
inheriting annotations, 407
initializing instance variables in, 147
methods in, 145
preventing, 151
public, 147
superclass assignments in, 147
subList method (List), 250, 265
subMap method (SortedMap), 265
subpath method (Path), 314
subSet method
of NavigableSet, 255
of SortedSet, 255, 265
substring method (String), 26
subtractExact method (Math), 20
subtraction, 19
accurate, 24
not associative, 292
subtypes, 110
wildcards for, 224
sum method
of LongAdder, 374
of *Xxx*Stream, 295
summarizing*Xxx* methods (Collectors), 287, 291
summaryStatistics method (*Xxx*Stream), 295
summing*Xxx* methods (Collectors), 290
super keyword, 116, 146–147, 150, 225–227
superclasses, 145
annotating, 401
calling equals method on, 161
default methods of, 157

methods of, 145–147
public, 147
serializability of, 335
supertypes, 110, 113
wildcards for, 225–226
Supplier interface, 129, 354
supplyAsync method (CompletableFuture), 353–355
supportsNormalTermination method
of Process, 389
of ProcessHandle, 390
@SuppressWarnings annotation, 232, 406–408, 480
suspend method (Thread, deprecated), 382
swap method (Collections), 252
Swing GUI toolkit, 120–121, 359
SwingConstants interface, 113
SwingWorker class (Swing), 359
switch statement, 39–41
fall-through variant of, 40
using enumerations in, 170
with pattern matching, 154
symbolic links, 316–317
synchronized keyword, 376–380
synchronized views, 266
synchronized*Xxx* methods (Collections), 252
System class
getLogger method, 207–208
getProperties method, 260
getProperty method, 175, 204, 259
setOut method, 81
setProperty method, 210
system class loader, 174, 176
system classes, enabling/disabling assertions for, 205
system properties, 260
System.err constant, 202, 211, 385, 464
System.in constant, 35
System.Logger interface, 207–209
getName method, 209
isLoggable method, 209
log method, 207–209
System.Logger.Level enumeration, 208
System.out constant, 6, 17, 35–38, 52, 57, 81, 125, 206, 309, 464
system*Xxx* methods (Preferences), 459–460

T

T, in dates, 434
t, T conversion characters, 37
\t
in regular expressions, 325
tab, for character literals, 14
%t pattern variable, 213
tab completion, 9–10
tagging interfaces, 165
tailMap method (SortedMap), 265
tailSet method
of NavigableSet, 255
of SortedSet, 255, 265
take method (BlockingQueue), 371
takeWhile method (Stream), 278
tar program, 85
@Target annotation, 404–406
tasks, 348–353
canceling, 351–352
combining results from, 351–353
computationally intensive, 349
coordinating work between, 370–372
defining, 120
executing, 120, 349
groups of, 385
long-running, 358–359
running, 348–350
short-lived, 349
submitting, 351
vs. threads, 349
working simultaneously, 354
teeing method (Collectors), 291
Temporal interface, 428
TemporalAdjuster.ofDateAdjuster method, 428
TemporalAdjusters class, 428
terminal window, 4
test method
of BiPredicate, 129
of Predicate, 129, 225
of *Xxx*Predicate, 130
@Test annotation, 398–399, 404
text blocks, 33–34
TextStyle enumeration, 451
thenAccept method (CompletableFuture), 353, 357

thenAcceptBoth method (CompletableFuture), 357–358
thenApply, thenApplyAsync methods (CompletableFuture), 355–357
thenCombine method (CompletableFuture), 357–358
thenComparing method (Comparator), 136–137
thenCompose method (CompletableFuture), 356–357
thenRun method (CompletableFuture), 357
third-party libraries, 489–490
this reference, 69–70
  annotating, 403
  capturing, 125
  in constructors, 73, 366
  in lambda expressions, 132
Thread class
  getContextClassLoader method, 176–177
  interrupted, isInterrupted methods, 382
  join method, 381
  properties, 385
  resume method (deprecated), 382
  setContextClassLoader method, 176–177
  setDaemon method, 385
  setDefaultUncaughtExceptionHandler method, 202
  setUncaughtExceptionHandler method, 381
  sleep method, 381, 383
  start method, 381
  stop, suspend methods (deprecated), 382
ThreadLocal class, 384–385
ThreadLocalRandom.current method, 384
threads, 349, 381–385
  atomic mutations in, 373–375
  creating, 120
  daemon, 385
  groups of, 385
  interrupting, 351, 382–383
  local variables in, 384–385
  locking, 375–377
  names of, 385
  priorities of, 385
  race conditions in, 295, 362–364
  running tasks in, 120
  starting, 381–382
  states of, 385
  temporarily inactive, 383
  terminating, 349–350
  uncaught exception handlers of, 385
  visibility and, 360–362, 378
  vs. tasks, 349
  waiting on conditions, 379
  worker, 358–359
throw statement, 193
Throwable class, 193
  in assertions, 205
  initCause method, 202
  no generic subtypes for, 237
  printStackTrace method, 203
throws keyword, 195
  type variables in, 237–238
@throws tag (javadoc), 97, 196
time
  current, 422
  formatting, 433–436, 449–451
  measuring, 423
  parsing, 435
Time class, 436–437
time indicator, in string templates, 453
time zones, 430–433
TimeoutException, 351
Timestamp class, 162, 436–437
timestamps, 423, 433
TimeZone class, 437
™ (trademark symbol), 452–453
toAbsolutePath method (Path), 314
toArray method
  of Collection, 249
  of Stream, 126, 286
  of *Xxx*Stream, 295
toByteArray method
  of BitSet, 261
  of ByteArrayOutputStream, 302–303
toCollection method (Collectors), 286
toConcurrentMap method (Collectors), 288
toDays method (Duration), 423
ToDoubleFunction interface, 130, 232
toEpochSecond method
  of LocalDate, 426
  of LocalTime, 429
toFile method (Path), 314
toFormat method (DateTimeFormatter), 435

toGenericString method (Class), 172
toHandle method (Process), 389
toHours method (Duration), 423
toInstant method
  of Date, 436
  of ZonedDateTime, 431, 433
toIntExact method (Math), 22
ToIntFunction interface, 130, 232
tokens method (Scanner), 275, 308
toList method (Stream), 275
toLocal*Xxx* methods (ZonedDateTime), 433
toLongArray method (BitSet), 261
ToLongFunction interface, 130, 232
toLowerCase method (String), 29, 276, 447
toMap method (Collectors), 287–288
toMillis, toMinutes (Duration), 423
toNanoOfDay method (LocalTime), 429
toNanos method (Duration), 423
ToolProvider.getSystemJavaCompiler method, 464
toPath method (File), 314
toSecondOfDay method (LocalTime), 429
toSeconds method (Duration), 423
toSet method (Collectors), 286, 290
toString method
  calling from subclasses, 158
  of Arrays, 52, 159
  of BitSet, 261
  of Class, 172
  of Double, 28
  of Enum, 167
  of Integer, 28
  of Modifier, 173
  of Object, 158–159
  of Point, 158–159
  of records, 77
toUnsignedInt method (Byte), 12
toUpperCase method (String), 29, 447
toZonedDateTime method (GregorianCalendar), 436–437
transferTo method (InputStream), 304
transient modifier, 336
transitive keyword, 493–495
TreeMap class, 255, 288
TreeSet class, 254
true value (boolean), 14
try statement, 196–200
  for visiting directories, 316
tryLock method (FileChannel), 312
trySetAccessible method (AccessibleObject), 181
try-with-resources statement, 197–199
  closing output streams with, 304
  for file locking, 312
type bounds, 222–223, 241
  annotating, 402
type erasure, 228–231, 236
  clashes after, 236–237
Type interface, 240
type method (ServiceLoader.Provider), 178
type parameters, 117, 220–221
  annotating, 400
  primitive types and, 221, 231
type variables
  exceptions and, 237–238
  in static context, 236
  no instantiating of, 233–235
  wildcards with, 226–227
TypeElement interface, 414
TypeVariable interface, 240

U

U+, for code points, 32
\u
  for character literals, 14, 457–458
  in regular expressions, 325
%u pattern variable, 213
UnaryOperator interface, 129
uncaught exception handlers, 381, 385
unchecked exceptions, 194
  documenting, 196
  generic types and, 238
UncheckedIOException, 308
Unicode, 31–33, 294, 305
  normalization forms in, 452
  replacement character in, 309
unit tests, 397
Unix operating system
  executable files in, 5
  path separator in, 86, 260
  specifying locales in, 446
  wildcard in classpath in, 86

unlock method (ReentrantLock), 376
unmodifiable*Xxx* methods (Collections), 252
unordered method (Stream), 296
until method (LocalDate), 426–427
updateAndGet method (Atomic*Xxx*), 373
URI class, 322
URL class, 322
  final, 151
  getInputStream method, 320
  openConnection method, 320
  openStream method, 302
URLClassLoader class, 175
URLConnection class, 320–321
  connect method, 320
  getHeaderFields method, 320
  getInputStream method, 321
  getOutputStream method, 320
  setDoOutput method, 320
  setRequestProperty method, 320
URLs, reading from, 302, 320
useLocale method (Scanner), 447
user directory, 314
user interface. *See* GUI
user preferences, 459–461
user.dir, user.home, user.name system properties, 260
user*Xxx* methods (Preferences), 459–460
uses keyword, 497
UTC (coordinated universal time), 431
UTF-8 encoding, 305
  for source files, 458
  modified, 310
UTF-16 encoding, 14, 32, 294, 305
  in regular expressions, 325
Util.createInstance method, 176–177
utility classes, 87, 176

V

V formatting symbol (date/time), 436
\v, \V, in regular expressions, 326
validateObject method (ObjectInputValidation), 342–343
valueOf method
  of BitSet, 261
  of Enum, 166–167
values method
  of Enum, 167
  of Map, 257, 264
var keyword, 15–16
varargs parameters
  declaring, 57
  safety of, 406, 408
VarHandle class, 487
variable handles, 487
VariableElement interface, 414
variables, 7, 14–17
  atomic mutations of, 373–375
  capturing, in lambda expressions, 132–134
  declaring, 15–16
  defined in interfaces, 113
  deprecated, 97, 406–407
  documentation comments for, 95, 97
  effectively final, 133–134
  final, 361, 365
  holding object references, 65–66
  initializing, 15–17
  instance, 67, 69, 72–75, 77–78, 82, 147, 152, 161, 336, 339–341
  local, 45–46
  naming, 15–16
  parameter, 70
  private, 67, 88
  public static final, 113
  redefining, 46
  scope of, 45, 88
  shared, 362–365, 375–377
  static final. *See* constants
  static, 79–80, 82, 90, 361
  thread-local, 384–385
  using an abstract class as type of, 152
  visibility of, 360–362, 378
  volatile, 361–362
@version tag (javadoc), 96, 99
versioning, 340–342
views, 264–266
virtual machine, 4
  instruction reordering in, 361
visibility, 360–362
  guaranteed with locks, 378

visitFile, visitFileFailed methods (FileVisitor), 318
void keyword, 2, 56
  using class literals with, 171
volatile modifier, 362

W

\w, \W, in regular expressions, 326
wait method (Object), 379–381
waitFor method (Process), 388–389
waiting on a condition, 380
walk method (Files), 316–319
walkFileTree method (Files), 316, 318
warning method (Logger), 406
warnings, suppressing, 232, 236, 407
weak references, 263
weaker access privilege, 147
WeakHashMap class, 263
weakly consistent iterators, 368
WeakReference class, 264
web pages
  extracting links from, 355
  reading, 356, 358
whenComplete method (CompletableFuture), 354, 356–357
while statement, 41–43
  breaking/continuing, 43–44
  declaring variables for, 45
white space
  in regular expressions, 326
  in text blocks, 34
  removing, 29
wildcards
  annotating, 402
  capturing, 228
  for annotation processors, 413
  for types, 224–226
  in class path, 86
  unbounded, 227
  with imported classes, 88–89
  with type variables, 226–227
WildcardType interface, 240
Window class, 88
WindowAdapter class, 114
WindowListener interface, 114
with method (Temporal), 428
withDayOf*Xxx* methods
  of LocalDate, 425
  of ZonedDateTime, 432
withHour method
  of LocalTime, 429
  of ZonedDateTime, 432
withLocale method (DateTimeFormatter), 434, 450
withMinute, withNano, withSecond methods
  of LocalTime, 429
  of ZonedDateTime, 432
withMonth method
  of LocalDate, 425
  of LocalTime, 429
  of ZonedDateTime, 432
withYear method
  of LocalDate, 425
  of ZonedDateTime, 432
withZoneSame*Xxx* methods (ZonedDateTime), 432
words
  in regular expressions, 326
  reading from a file, 308
  sorting alphabetically, 451–452
working directory, 386
wrapper classes, 49–50
write method
  of Files, 309, 316
  of OutputStream, 304
  of Writer, 308
writeByte, writeChar methods (DataOutput), 310
writeDouble method
  of DataOutput, 310
  of ObjectOutputStream, 337
writeExternal method (Externalizable), 338–339
writeFloat, writeFully methods (DataOutput), 310
writeInt method (DataOutput), 310–311
writeLong method (DataOutput), 310
writeObject method
  of HashSet, 337
  of ObjectOutputStream, 334–337
Writer class, 308–309
  write method, 308

writeReplace method (Serializable), 339–340
writers, 302
writeShort, writeUnsigned*Xxx*, writeUTF methods (DataOutput), 310

X

x, X
  conversion characters, 37
  formatting symbols (date/time), 436
\x, in regular expressions, 325
XML descriptors, generating, 417
xor method (BitSet), 261
Xoroshiro128PlusPlus algorithm, 106

Y

y formatting symbol (date/time), 436
Year, YearMonth classes, 427
yield statement, 40–41

Z

z, Z formatting symbols (date/time), 434, 436
\z, \Z, in regular expressions, 328
ZIP file systems, 319
ZipInputStream, ZipOutputStream classes, 319
zoned time, 424–427, 430–433
ZonedDateTime class, 430–433
  get*Xxx*, is*Xxx* methods, 432–433
  legacy classes and, 437
  minus, minus*Xxx*, now methods, 432
  of method, 430–432
  ofInstant method, 432
  parse method, 450
  plus, plus*Xxx* methods, 431–432
  toInstant method, 431, 433
  toLocal*Xxx* methods, 433
  with*Xxx* methods, 432
ZoneId class, 430