Core Java® SE 9 for the Impatient

Second Edition
This page intentionally left blank
To Chi—the most patient person in my life.
This page intentionally left blank
# Contents

Preface xxi  
Acknowledgments xxiii  
About the Author xxv  

## 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  3  
1.1.3 Method Calls  6  
1.1.4 JShell  7  

1.2 Primitive Types  10  
1.2.1 Signed Integer Types  10  
1.2.2 Floating-Point Types  12  
1.2.3 The char Type  13  
1.2.4 The boolean Type  14  

1.3 Variables  14  
1.3.1 Variable Declarations  14  
1.3.2 Names  14
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.3</td>
<td>Initialization</td>
<td>15</td>
</tr>
<tr>
<td>1.3.4</td>
<td>Constants</td>
<td>15</td>
</tr>
<tr>
<td>1.4</td>
<td>Arithmetic Operations</td>
<td>17</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Assignment</td>
<td>18</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Basic Arithmetic</td>
<td>18</td>
</tr>
<tr>
<td>1.4.3</td>
<td>Mathematical Methods</td>
<td>19</td>
</tr>
<tr>
<td>1.4.4</td>
<td>Number Type Conversions</td>
<td>20</td>
</tr>
<tr>
<td>1.4.5</td>
<td>Relational and Logical Operators</td>
<td>22</td>
</tr>
<tr>
<td>1.4.6</td>
<td>Big Numbers</td>
<td>23</td>
</tr>
<tr>
<td>1.5</td>
<td>Strings</td>
<td>24</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Concatenation</td>
<td>24</td>
</tr>
<tr>
<td>1.5.2</td>
<td>Substrings</td>
<td>25</td>
</tr>
<tr>
<td>1.5.3</td>
<td>String Comparison</td>
<td>25</td>
</tr>
<tr>
<td>1.5.4</td>
<td>Converting Between Numbers and Strings</td>
<td>27</td>
</tr>
<tr>
<td>1.5.5</td>
<td>The String API</td>
<td>28</td>
</tr>
<tr>
<td>1.5.6</td>
<td>Code Points and Code Units</td>
<td>30</td>
</tr>
<tr>
<td>1.6</td>
<td>Input and Output</td>
<td>32</td>
</tr>
<tr>
<td>1.6.1</td>
<td>Reading Input</td>
<td>32</td>
</tr>
<tr>
<td>1.6.2</td>
<td>Formatted Output</td>
<td>33</td>
</tr>
<tr>
<td>1.7</td>
<td>Control Flow</td>
<td>36</td>
</tr>
<tr>
<td>1.7.1</td>
<td>Branches</td>
<td>36</td>
</tr>
<tr>
<td>1.7.2</td>
<td>Loops</td>
<td>38</td>
</tr>
<tr>
<td>1.7.3</td>
<td>Breaking and Continuing</td>
<td>39</td>
</tr>
<tr>
<td>1.7.4</td>
<td>Local Variable Scope</td>
<td>41</td>
</tr>
<tr>
<td>1.8</td>
<td>Arrays and Array Lists</td>
<td>43</td>
</tr>
<tr>
<td>1.8.1</td>
<td>Working with Arrays</td>
<td>43</td>
</tr>
<tr>
<td>1.8.2</td>
<td>Array Construction</td>
<td>44</td>
</tr>
<tr>
<td>1.8.3</td>
<td>Array Lists</td>
<td>45</td>
</tr>
<tr>
<td>1.8.4</td>
<td>Wrapper Classes for Primitive Types</td>
<td>46</td>
</tr>
<tr>
<td>1.8.5</td>
<td>The Enhanced for Loop</td>
<td>47</td>
</tr>
<tr>
<td>1.8.6</td>
<td>Copying Arrays and Array Lists</td>
<td>47</td>
</tr>
<tr>
<td>1.8.7</td>
<td>Array Algorithms</td>
<td>49</td>
</tr>
<tr>
<td>1.8.8</td>
<td>Command-Line Arguments</td>
<td>49</td>
</tr>
<tr>
<td>1.8.9</td>
<td>Multidimensional Arrays</td>
<td>50</td>
</tr>
</tbody>
</table>
1.9 Functional Decomposition 52
  1.9.1 Declaring and Calling Static Methods 53
  1.9.2 Array Parameters and Return Values 53
  1.9.3 Variable Arguments 53

Exercises 54

2 OBJECT-ORIENTED PROGRAMMING 59
  2.1 Working with Objects 60
    2.1.1 Accessor and Mutator Methods 62
    2.1.2 Object References 63
  2.2 Implementing Classes 65
    2.2.1 Instance Variables 65
    2.2.2 Method Headers 65
    2.2.3 Method Bodies 66
    2.2.4 Instance Method Invocations 66
    2.2.5 The this Reference 67
    2.2.6 Call by Value 68
  2.3 Object Construction 69
    2.3.1 Implementing Constructors 69
    2.3.2 Overloading 70
    2.3.3 Calling One Constructor from Another 71
    2.3.4 Default Initialization 71
    2.3.5 Instance Variable Initialization 72
    2.3.6 Final Instance Variables 73
    2.3.7 The Constructor with No Arguments 73
  2.4 Static Variables and Methods 74
    2.4.1 Static Variables 74
    2.4.2 Static Constants 75
    2.4.3 Static Initialization Blocks 76
    2.4.4 Static Methods 77
    2.4.5 Factory Methods 78
  2.5 Packages 78
    2.5.1 Package Declarations 79
    2.5.2 The jar Command 80
2.5.3  The Class Path  81
2.5.4  Package Access  83
2.5.5  Importing Classes  83
2.5.6  Static Imports  85
2.6  Nested Classes  85
  2.6.1  Static Nested Classes  85
  2.6.2  Inner Classes  87
  2.6.3  Special Syntax Rules for Inner Classes  89
2.7  Documentation Comments  90
  2.7.1  Comment Insertion  90
  2.7.2  Class Comments  91
  2.7.3  Method Comments  92
  2.7.4  Variable Comments  92
  2.7.5  General Comments  92
  2.7.6  Links  93
  2.7.7  Package, Module, and Overview Comments  94
  2.7.8  Comment Extraction  94

Exercises  95

3  INTERFACES AND LAMBDA EXPRESSIONS  99
3.1  Interfaces  100
  3.1.1  Declaring an Interface  100
  3.1.2  Implementing an Interface  101
  3.1.3  Converting to an Interface Type  103
  3.1.4  Casts and the instanceof Operator  103
  3.1.5  Extending Interfaces  104
  3.1.6  Implementing Multiple Interfaces  105
  3.1.7  Constants  105
3.2  Static, Default, and Private Methods  105
  3.2.1  Static Methods  105
  3.2.2  Default Methods  106
  3.2.3  Resolving Default Method Conflicts  107
  3.2.4  Private Methods  109
3.3 Examples of Interfaces 109
3.3.1 The Comparable Interface 109
3.3.2 The Comparator Interface 111
3.3.3 The Runnable Interface 112
3.3.4 User Interface Callbacks 112

3.4 Lambda Expressions 113
3.4.1 The Syntax of Lambda Expressions 114
3.4.2 Functional Interfaces 115

3.5 Method and Constructor References 116
3.5.1 Method References 117
3.5.2 Constructor References 118

3.6 Processing Lambda Expressions 119
3.6.1 Implementing Deferred Execution 119
3.6.2 Choosing a Functional Interface 120
3.6.3 Implementing Your Own Functional Interfaces 123

3.7 Lambda Expressions and Variable Scope 124
3.7.1 Scope of a Lambda Expression 124
3.7.2 Accessing Variables from the Enclosing Scope 124

3.8 Higher-Order Functions 127
3.8.1 Methods that Return Functions 127
3.8.2 Methods That Modify Functions 128
3.8.3 Comparator Methods 128

3.9 Local and Anonymous Classes 129
3.9.1 Local Classes 129
3.9.2 Anonymous Classes 130

Exercises 131

4 INHERITANCE AND REFLECTION 135
4.1 Extending a Class 136
4.1.1 Super- and Subclasses 136
4.1.2 Defining and Inheriting Subclass Methods 137
4.1.3 Method Overriding 137
4.1.4 Subclass Construction 139
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.5</td>
<td>Superclass Assignments</td>
<td>139</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Casts</td>
<td>140</td>
</tr>
<tr>
<td>4.1.7</td>
<td>Final Methods and Classes</td>
<td>141</td>
</tr>
<tr>
<td>4.1.8</td>
<td>Abstract Methods and Classes</td>
<td>141</td>
</tr>
<tr>
<td>4.1.9</td>
<td>Protected Access</td>
<td>142</td>
</tr>
<tr>
<td>4.1.10</td>
<td>Anonymous Subclasses</td>
<td>143</td>
</tr>
<tr>
<td>4.1.11</td>
<td>Inheritance and Default Methods</td>
<td>144</td>
</tr>
<tr>
<td>4.1.12</td>
<td>Method Expressions with <code>super</code></td>
<td>145</td>
</tr>
<tr>
<td>4.2</td>
<td><code>Object</code>: The Cosmic Superclass</td>
<td>145</td>
</tr>
<tr>
<td>4.2.1</td>
<td>The <code>toString</code> Method</td>
<td>146</td>
</tr>
<tr>
<td>4.2.2</td>
<td>The <code>equals</code> Method</td>
<td>148</td>
</tr>
<tr>
<td>4.2.3</td>
<td>The <code>hashCode</code> Method</td>
<td>150</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Cloning Objects</td>
<td>151</td>
</tr>
<tr>
<td>4.3</td>
<td>Enumerations</td>
<td>154</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Methods of Enumerations</td>
<td>155</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Constructors, Methods, and Fields</td>
<td>156</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Bodies of Instances</td>
<td>157</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Static Members</td>
<td>157</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Switching on an Enumeration</td>
<td>158</td>
</tr>
<tr>
<td>4.4</td>
<td>Runtime Type Information and Resources</td>
<td>159</td>
</tr>
<tr>
<td>4.4.1</td>
<td>The Class <code>Class</code></td>
<td>159</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Loading Resources</td>
<td>162</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Class Loaders</td>
<td>163</td>
</tr>
<tr>
<td>4.4.4</td>
<td>The Context Class Loader</td>
<td>164</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Service Loaders</td>
<td>166</td>
</tr>
<tr>
<td>4.5</td>
<td>Reflection</td>
<td>168</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Enumerating Class Members</td>
<td>168</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Inspecting Objects</td>
<td>169</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Invoking Methods</td>
<td>171</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Constructing Objects</td>
<td>171</td>
</tr>
<tr>
<td>4.5.5</td>
<td>JavaBeans</td>
<td>172</td>
</tr>
<tr>
<td>4.5.6</td>
<td>Working with Arrays</td>
<td>174</td>
</tr>
<tr>
<td>4.5.7</td>
<td>Proxies</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>177</td>
</tr>
</tbody>
</table>
5  EXCEPTIONS, ASSERTIONS, AND LOGGING  181
      5.1  Exception Handling  182
          5.1.1  Throwing Exceptions  182
          5.1.2  The Exception Hierarchy  183
          5.1.3  Declaring Checked Exceptions  185
          5.1.4  Catching Exceptions  186
          5.1.5  The Try-with-Resources Statement  187
          5.1.6  The finally Clause  189
          5.1.7  Rethrowing and Chaining Exceptions  190
          5.1.8  Uncaught Exceptions and the Stack Trace  192
          5.1.9  The Objects.requireNonNull Method  193
      5.2  Assertions  193
          5.2.1  Using Assertions  194
          5.2.2  Enabling and Disabling Assertions  194
      5.3  Logging  195
          5.3.1  Using Loggers  195
          5.3.2  Loggers  196
          5.3.3  Logging Levels  197
          5.3.4  Other Logging Methods  197
          5.3.5  Logging Configuration  199
          5.3.6  Log Handlers  200
          5.3.7  Filters and Formatters  202

Exercises  203

6  GENERIC PROGRAMMING  207
       6.1  Generic Classes  208
       6.2  Generic Methods  209
       6.3  Type Bounds  210
       6.4  Type Variance and Wildcards  211
           6.4.1  Subtype Wildcards  212
           6.4.2  Supertype Wildcards  213
           6.4.3  Wildcards with Type Variables  214
           6.4.4  Unbounded Wildcards  215
           6.4.5  Wildcard Capture  216
6.5 Generics in the Java Virtual Machine 216
   6.5.1 Type Erasure 217
   6.5.2 Cast Insertion 217
   6.5.3 Bridge Methods 218

6.6 Restrictions on Generics 220
   6.6.1 No Primitive Type Arguments 220
   6.6.2 At Runtime, All Types Are Raw 220
   6.6.3 You Cannot Instantiate Type Variables 221
   6.6.4 You Cannot Construct Arrays of Parameterized Types 223
   6.6.5 Class Type Variables Are Not Valid in Static Contexts 224
   6.6.6 Methods May Not Clash after Erasure 224
   6.6.7 Exceptions and Generics 225

6.7 Reflection and Generics 226
   6.7.1 The Class\langle T\rangle Class 227
   6.7.2 Generic Type Information in the Virtual Machine 227

Exercises 229

7 COLLECTIONS 235

7.1 An Overview of the Collections Framework 236
7.2 Iterators 240
7.3 Sets 242
7.4 Maps 243
7.5 Other Collections 247
   7.5.1 Properties 247
   7.5.2 Bit Sets 248
   7.5.3 Enumeration Sets and Maps 250
   7.5.4 Stacks, Queues, Deques, and Priority Queues 250
   7.5.5 Weak Hash Maps 251

7.6 Views 252
   7.6.1 Small Collections 252
   7.6.2 Ranges 253
9.1.8 Random-Access Files 296
9.1.9 Memory-Mapped Files 297
9.1.10 File Locking 297

9.2 Paths, Files, and Directories 298
9.2.1 Paths 298
9.2.2 Creating Files and Directories 300
9.2.3 Copying, Moving, and Deleting Files 301
9.2.4 Visiting Directory Entries 302
9.2.5 ZIP File Systems 305

9.3 HTTP Connections 306
9.3.1 The URLConnection and HttpURLConnection Classes 306
9.3.2 The HTTP Client API 307

9.4 Regular Expressions 310
9.4.1 The Regular Expression Syntax 310
9.4.2 Finding One Match 314
9.4.3 Finding All Matches 315
9.4.4 Groups 316
9.4.5 Splitting along Delimiters 317
9.4.6 Replacing Matches 317
9.4.7 Flags 318

9.5 Serialization 319
9.5.1 The Serializable Interface 319
9.5.2 Transient Instance Variables 321
9.5.3 The readObject and writeObject Methods 321
9.5.4 The readResolve and writeReplace Methods 322
9.5.5 Versioning 324

Exercises 325

10 CONCURRENT PROGRAMMING 329

10.1 Concurrent Tasks 330
10.1.1 Running Tasks 330
10.1.2 Futures 333

10.2 Asynchronous Computations 335
10.2.1 Completable Futures 335
10.2.2 Composing Completable Futures 337
10.2.3 Long-Running Tasks in User-Interface Callbacks 340
10.3 Thread Safety 341
10.3.1 Visibility 342
10.3.2 Race Conditions 344
10.3.3 Strategies for Safe Concurrency 346
10.3.4 Immutable Classes 347
10.4 Parallel Algorithms 348
10.4.1 Parallel Streams 348
10.4.2 Parallel Array Operations 349
10.5 Threadsafe Data Structures 350
10.5.1 Concurrent Hash Maps 350
10.5.2 Blocking Queues 352
10.5.3 Other Threadsafe Data Structures 354
10.6 Atomic Counters and Accumulators 354
10.7 Locks and Conditions 357
10.7.1 Locks 357
10.7.2 The synchronized Keyword 358
10.7.3 Waiting on Conditions 360
10.8 Threads 362
10.8.1 Starting a Thread 363
10.8.2 Thread Interruption 364
10.8.3 Thread-Local Variables 365
10.8.4 Miscellaneous Thread Properties 366
10.9 Processes 366
10.9.1 Building a Process 367
10.9.2 Running a Process 368
10.9.3 Process Handles 370
Exercises 371

II ANNOTATIONS 377
11.1 Using Annotations 378
11.1.1 Annotation Elements 378
11.1.2 Multiple and Repeated Annotations 380
11.1.3 Annotating Declarations 380
11.1.4 Annotating Type Uses 381
11.1.5 Making Receivers Explicit 382
11.2 Defining Annotations 383
11.3 Standard Annotations 386
  11.3.1 Annotations for Compilation 387
  11.3.2 Annotations for Managing Resources 388
  11.3.3 Meta-Annotations 389
11.4 Processing Annotations at Runtime 391
11.5 Source-Level Annotation Processing 394
  11.5.1 Annotation Processors 394
  11.5.2 The Language Model API 395
  11.5.3 Using Annotations to Generate Source Code 395
Exercises 398

I2  THE DATE AND TIME API 401
12.1 The Time Line 402
12.2 Local Dates 404
12.3 Date Adjusters 407
12.4 Local Time 409
12.5 Zoned Time 410
12.6 Formatting and Parsing 413
12.7 Interoperating with Legacy Code 416
Exercises 417

I3  INTERNATIONALIZATION 421
13.1 Locales 422
  13.1.1 Specifying a Locale 423
  13.1.2 The Default Locale 426
  13.1.3 Display Names 426
13.2 Number Formats 427
13.3 Currencies 428
13.4 Date and Time Formatting 429
13.5 Collation and Normalization 431
13.6 Message Formatting 433
13.7 Resource Bundles 435
  13.7.1 Organizing Resource Bundles 435
  13.7.2 Bundle Classes 437
13.8 Character Encodings 438
13.9 Preferences 439
Exercises 441

14 COMPILING AND SCRIPTING 443
14.1 The Compiler API 444
  14.1.1 Invoking the Compiler 444
  14.1.2 Launching a Compilation Task 444
  14.1.3 Reading Source Files from Memory 445
  14.1.4 Writing Byte Codes to Memory 446
  14.1.5 Capturing Diagnostics 447
14.2 The Scripting API 448
  14.2.1 Getting a Scripting Engine 448
  14.2.2 Bindings 449
  14.2.3 Redirecting Input and Output 449
  14.2.4 Calling Scripting Functions and Methods 450
  14.2.5 Compiling a Script 452
14.3 The Nashorn Scripting Engine 452
  14.3.1 Running Nashorn from the Command Line 452
  14.3.2 Invoking Getters, Setters, and Overloaded Methods 453
  14.3.3 Constructing Java Objects 454
  14.3.4 Strings in JavaScript and Java 455
  14.3.5 Numbers 456
  14.3.6 Working with Arrays 457
  14.3.7 Lists and Maps 458
  14.3.8 Lambdas 458
  14.3.9 Extending Java Classes and Implementing Java Interfaces 459
  14.3.10 Exceptions 461
14.4 Shell Scripting with Nashorn 461
   14.4.1 Executing Shell Commands 462
   14.4.2 String Interpolation 462
   14.4.3 Script Inputs 463

Exercises 464

15 THE JAVA PLATFORM MODULE SYSTEM 469

   15.1 The Module Concept 470
   15.2 Naming Modules 472
   15.3 The Modular “Hello, World!” Program 472
   15.4 Requiring Modules 474
   15.5 Exporting Packages 476
   15.6 Modules and Reflective Access 479
   15.7 Modular JARs 482
   15.8 Automatic Modules and the Unnamed Module 484
   15.9 Command-Line Flags for Migration 485
   15.10 Transitive and Static Requirements 487
   15.11 Qualified Exporting and Opening 489
   15.12 Service Loading 490
   15.13 Tools for Working with Modules 491

Exercises 494

Index 497
Java is now over twenty years old, and 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 well 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, with Java 9, 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, Objective C, 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, including the powerful concepts of lambda expressions and streams. I tell you where to find out more about old-fashioned concepts that you might still see in legacy code, but I don’t dwell on them.

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 and leave a comment. On that page, you will also find a link to an archive file containing all code examples from the book.
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 SE 9. 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 both editions who spotted many errors and gave thoughtful suggestions for improvement. They are: Andres Almiray, Gail Anderson, Paul Anderson, Marcus Biel, Brian Goetz, Marty Hall, Mark Lawrence, Doug Lea, Simon Ritter, Yoshiki Shibata, and Christian Ullenboom.

*Cay Horstmann*

*San Francisco*

*July 2017*
Cay S. Horstmann is the author of *Java SE 8 for the Really Impatient* and *Scala for the Impatient* (both from Addison-Wesley), is principal author of *Core Java™, Volumes I and II, Tenth Edition* (Prentice Hall, 2016), and has written a dozen other books for professional programmers and computer science students. He is a professor of computer science at San Jose State University and is a Java Champion.
Interfaces and Lambda Expressions

Topics in This Chapter

- 3.1 Interfaces — page 100
- 3.2 Static, Default, and Private Methods — page 105
- 3.3 Examples of Interfaces — page 109
- 3.4 Lambda Expressions — page 113
- 3.5 Method and Constructor References — page 116
- 3.6 Processing Lambda Expressions — page 119
- 3.7 Lambda Expressions and Variable Scope — page 124
- 3.8 Higher-Order Functions — page 127
- 3.9 Local and Anonymous Classes — page 129
- Exercises — page 131
Java was designed as an object-oriented programming language in the 1990s when object-oriented programming was the principal paradigm for software development. Interfaces are a key feature of object-oriented programming: They let you specify what should be done, without having to provide an implementation.

Long before there was object-oriented programming, there were functional programming languages, such as Lisp, in which functions and not objects are the primary structuring mechanism. Recently, functional programming has risen in importance because it is well suited for concurrent and event-driven (or “reactive”) programming. Java supports function expressions that provide a convenient bridge between object-oriented and functional programming. In this chapter, you will learn about interfaces and lambda expressions.

The key points of this chapter are:

1. An interface specifies a set of methods that an implementing class must provide.
2. An interface is a supertype of any class that implements it. Therefore, one can assign instances of the class to variables of the interface type.
3. An interface can contain static methods. All variables of an interface are automatically public, static, and final.
4. An interface can contain default methods that an implementing class can inherit or override.
5. An interface can contain private methods that cannot be called or overridden by implementing classes.
6. The `Comparable` and `Comparator` interfaces are used for comparing objects.
7. A functional interface is an interface with a single abstract method.
8. A lambda expression denotes a block of code that can be executed at a later point in time.
9. Lambda expressions are converted to functional interfaces.
10. Method and constructor references refer to methods or constructors without invoking them.
11. Lambda expressions and local classes can access effectively final variables from the enclosing scope.

### 3.1 Interfaces

An *interface* is a mechanism for spelling out a contract between two parties: the supplier of a service and the classes that want their objects to be usable with the service. In the following sections, you will see how to define and use interfaces in Java.

#### 3.1.1 Declaring an Interface

Consider a service that works on sequences of integers, reporting the average of the first `n` values:

```java
public static double average(IntSequence seq, int n)
```

Such sequences can take many forms. Here are some examples:

- A sequence of integers supplied by a user
- A sequence of random integers
- The sequence of prime numbers
- The sequence of elements in an integer array
- The sequence of code points in a string
- The sequence of digits in a number

We want to implement a *single mechanism* for dealing with all these kinds of sequences.
First, let us spell out what is common between integer sequences. At a minimum, one needs two methods for working with a sequence:

- Test whether there is a next element
- Get the next element

To declare an interface, you provide the method headers, like this:

```java
public interface IntSequence {
    boolean hasNext();
    int next();
}
```

You need not implement these methods, but you can provide default implementations if you like—see Section 3.2.2, “Default Methods” (page 106). If no implementation is provided, we say that the method is abstract.

NOTE: All methods of an interface are automatically public. Therefore, it is not necessary to declare `hasNext` and `next` as public. Some programmers do it anyway for greater clarity.

The methods in the interface suffice to implement the `average` method:

```java
public static double average(IntSequence seq, int n) {
    int count = 0;
    double sum = 0;
    while (seq.hasNext() && count < n) {
        count++;
        sum += seq.next();
    }
    return count == 0 ? 0 : sum / count;
}
```

### 3.1.2 Implementing an Interface

Now let’s look at the other side of the coin: the classes that want to be usable with the `average` method. They need to `implement` the `IntSequence` interface. Here is such a class:

```java
public class SquareSequence implements IntSequence {
    private int i;

    public boolean hasNext() {
        return true;
    }
}
There are infinitely many squares, and an object of this class delivers them all, one at a time. (To keep the example simple, we ignore integer overflow—see Exercise 6.)

The implements keyword indicates that the SquareSequence class intends to conform to the IntSequence interface.

CAUTION: The implementing class must declare the methods of the interface as public. Otherwise, they would default to package access. Since the interface requires public access, the compiler would report an error.

This code gets the average of the first 100 squares:

```java
SquareSequence squares = new SquareSequence();
double avg = average(squares, 100);
```

There are many classes that can implement the IntSequence interface. For example, this class yields a finite sequence, namely the digits of a positive integer starting with the least significant one:

```java
public class DigitSequence implements IntSequence {
    private int number;

    public DigitSequence(int n) {
        number = n;
    }

    public boolean hasNext() {
        return number != 0;
    }

    public int next() {
        int result = number % 10;
        number /= 10;
        return result;
    }

    public int rest() {
        return number;
    }
}
```
An object `new DigitSequence(1729)` delivers the digits 9 2 7 1 before `hasNext` returns false.

**NOTE:** The `SquareSequence` and `DigitSequence` classes implement all methods of the `IntSequence` interface. If a class only implements some of the methods, then it must be declared with the `abstract` modifier. See Chapter 4 for more information on abstract classes.

### 3.1.3 Converting to an Interface Type

This code fragment computes the average of the digit sequence values:

```java
IntSequence digits = new DigitSequence(1729);
double avg = average(digits, 100);
// Will only look at the first four sequence values
```

Look at the `digits` variable. Its type is `IntSequence`, not `DigitSequence`. A variable of type `IntSequence` refers to an object of some class that implements the `IntSequence` interface. You can always assign an object to a variable whose type is an implemented interface, or pass it to a method expecting such an interface.

Here is a bit of useful terminology. A type S is a *supertype* of the type T (the *subtype*) when any value of the subtype can be assigned to a variable of the supertype without a conversion. For example, the `IntSequence` interface is a supertype of the `DigitSequence` class.

**NOTE:** Even though it is possible to declare variables of an interface type, you can never have an object whose type is an interface. All objects are instances of classes.

### 3.1.4 Casts and the `instanceof` Operator

Occasionally, you need the opposite conversion—from a supertype to a subtype. Then you use a *cast*. For example, if you happen to know that the object stored in an `IntSequence` is actually a `DigitSequence`, you can convert the type like this:

```java
IntSequence sequence = ...;
DigitSequence digits = (DigitSequence) sequence;
System.out.println(digits.rest());
```

In this scenario, the cast was necessary because `rest` is a method of `DigitSequence` but not `IntSequence`. 
See Exercise 2 for a more compelling example.

You can only cast an object to its actual class or one of its supertypes. If you are wrong, a compile-time error or class cast exception will occur:

```java
String digitString = (String) sequence;
// Cannot possibly work—IntSequence is not a supertype of String
RandomSequence randoms = (RandomSequence) sequence;
// Could work, throws a class cast exception if not
```

To avoid the exception, you can first test whether the object is of the desired type, using the `instanceof` operator. The expression

```java
object instanceof Type
```

returns `true` if `object` is an instance of a class that has `Type` as a supertype. It is a good idea to make this check before using a cast.

```java
if (sequence instanceof DigitSequence) {
    DigitSequence digits = (DigitSequence) sequence;
    ...
}
```

**NOTE:** The `instanceof` operator is null-safe: The expression `obj instanceof Type` is `false` if `obj` is null. After all, `null` cannot possibly be a reference to an object of any given type.

### 3.1.5 Extending Interfaces

An interface can `extend` another, requiring or providing additional methods on top of the original ones. For example, `Closeable` is an interface with a single method:

```java
public interface Closeable {
    void close();
}
```

As you will see in Chapter 5, this is an important interface for closing resources when an exception occurs.

The `Channel` interface extends this interface:

```java
public interface Channel extends Closeable {
    boolean isOpen();
}
```

A class that implements the `Channel` interface must provide both methods, and its objects can be converted to both interface types.
3.1.6 Implementing Multiple Interfaces

A class can implement any number of interfaces. For example, a FileSequence class that reads integers from a file can implement the Closeable interface in addition to IntSequence:

```java
public class FileSequence implements IntSequence, Closeable {
    // ...
}
```

Then the FileSequence class has both IntSequence and Closeable as supertypes.

3.1.7 Constants

Any variable defined in an interface is automatically `public static final`.

For example, the SwingConstants interface defines constants for compass directions:

```java
public interface SwingConstants {
    int NORTH = 1;
    int NORTH_EAST = 2;
    int EAST = 3;
    // ...
}
```

You can refer to them by their qualified name, `SwingConstants.NORTH`. If your class chooses to implement the SwingConstants interface, you can drop the `SwingConstants` qualifier and simply write `NORTH`. However, this is not a common idiom. It is far better to use enumerations for a set of constants; see Chapter 4.

---

**NOTE:** You cannot have instance variables in an interface. An interface specifies behavior, not object state.

3.2 Static, Default, and Private Methods

In earlier versions of Java, all methods of an interface had to be abstract—that is, without a body. Nowadays you can add three kinds of methods with a concrete implementation: static, default, and private methods. The following sections describe these methods.

3.2.1 Static Methods

There was never a technical reason why an interface could not have static methods, but they did not fit into the view of interfaces as abstract
specifications. That thinking has now evolved. In particular, factory methods
make a lot of sense in interfaces. For example, the \texttt{IntSequence} interface can
have a static method \texttt{digitsOf} that generates a sequence of digits of a given
integer:
\begin{verbatim}
IntSequence digits = IntSequence.digitsOf(1729);
\end{verbatim}
The method yields an instance of some class implementing the \texttt{IntSequence}
interface, but the caller need not care which one it is.
\begin{verbatim}
public interface IntSequence {
...
    static IntSequence digitsOf(int n) {
        return new DigitSequence(n);
    }
}
\end{verbatim}

\textbf{NOTE:} In the past, it had been common to place static methods in a
companion class. You find pairs of interfaces and utility classes, such
as \texttt{Collection/Collections} or \texttt{Path/Paths}, in the Java API. This split is no
longer necessary.

3.2.2 Default Methods

You can supply a \texttt{default} implementation for any interface method. You must
tag such a method with the \texttt{default} modifier.
\begin{verbatim}
public interface IntSequence {
    default boolean hasNext() { return true; }
    // By default, sequences are infinite
    int next();
}
\end{verbatim}
A class implementing this interface can choose to override the \texttt{hasNext} method
or to inherit the default implementation.

\textbf{NOTE:} Default methods put an end to the classic pattern of providing
an interface and a companion class that implements most or all of
its methods, such as \texttt{Collection/AbstractCollection} or \texttt{WindowListener/
WindowAdapter} in the Java API. Nowadays you should just implement the
methods in the interface.

An important use for default methods is \textit{interface evolution}. Consider for ex-
ample the \texttt{Collection} interface that has been a part of Java for many years. Suppose
that way back when, you provided a class
Later, in Java 8, a `stream` method was added to the interface. Suppose the `stream` method was not a default method. Then the `Bag` class no longer compiles since it doesn’t implement the new method. Adding a nondefault method to an interface is not `source-compatible`. But suppose you don’t recompile the class and simply use an old JAR file containing it. The class will still load, even with the missing method. Programs can still construct `Bag` instances, and nothing bad will happen. (Adding a method to an interface is `binary-compatible`.) However, if a program calls the `stream` method on a `Bag` instance, an `AbstractMethodError` occurs.

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

### 3.2.3 Resolving Default Method Conflicts

If a class implements two interfaces, one of which has a default method and the other a method (default or not) with the same name and parameter types, then you must resolve the conflict. This doesn’t happen very often, and it is usually easy to deal with the situation.

Let’s look at an example. Suppose we have an interface `Person` with a `getId` method:

```java
public interface Person {
    String getName();
    default int getId() { return 0; }
}
```

And suppose there is an interface `Identified`, also with such a method.

```java
public interface Identified {
    default int getId() { return Math.abs(hashCode()); }
}
```

You will see what the `hashCode` method does in Chapter 4. For now, all that matters is that it returns some integer that is derived from the object.

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

```java
public class Employee implements Person, Identified {
    ...
}
```

The class inherits two `getId` methods provided by the `Person` and `Identified` interfaces. There is no way for the Java compiler to choose one over the other. The compiler reports an error and leaves it up to you to resolve the ambiguity.
Provide a `getId` method in the `Employee` class and either implement your own ID scheme, or delegate to one of the conflicting methods, like this:

```java
public class Employee implements Person, Identified {
    public int getId() { return Identified.super.getId(); }
    ...
}
```

**NOTE:** The `super` keyword lets you call a supertype method. In this case, we need to specify which supertype we want. The syntax may seem a bit odd, but it is consistent with the syntax for invoking a superclass method that you will see in Chapter 4.

Now assume that the `Identified` interface does not provide a default implementation for `getId`:

```java
interface Identified {
    int getId();
}
```

Can the `Employee` class inherit the default method from the `Person` interface? At first glance, this might seem reasonable. But how does the compiler know whether the `Person.getId` method actually does what `Identified.getId` is expected to do? After all, it might return the level of the person’s Freudian id, not an ID number.

The Java designers decided in favor of safety and uniformity. It doesn’t matter how two interfaces conflict; if at least one interface provides an implementation, the compiler reports an error, and it is up to the programmer to resolve the ambiguity.

**NOTE:** If neither interface provides a default for a shared method, then there is no conflict. An implementing class has two choices: implement the method, or leave it unimplemented and declare the class as abstract.

**NOTE:** If a class extends a superclass (see Chapter 4) and implements an interface, inheriting the same method from both, the rules are easier. In that case, only the superclass method matters, and any default method from the interface is simply ignored. This is actually a more common case than conflicting interfaces. See Chapter 4 for the details.
3.2.4 Private Methods

As of Java 9, methods in an interface can be private. A private method can be static or an instance method, but it cannot be a default method since that can be overridden. As private methods can only be used in the methods of the interface itself, their use is limited to being helper methods for the other methods of the interface.

For example, suppose the IntSequence class provides methods

static of(int a)
static of(int a, int b)
static of(int a, int b, int c)

Then each of these methods could call a helper method

private static IntSequence makeFiniteSequence(int... values) { ... }

3.3 Examples of Interfaces

At first glance, interfaces don’t seem to do very much. An interface is just a set of methods that a class promises to implement. To make the importance of interfaces more tangible, the following sections show you four examples of commonly used interfaces from the Java API.

3.3.1 The Comparable Interface

Suppose you want to sort an array of objects. A sorting algorithm repeatedly compares elements and rearranges them if they are out of order. Of course, the rules for doing the comparison are different for each class, and the sorting algorithm should just call a method supplied by the class. As long as all classes can agree on what that method is called, the sorting algorithm can do its job. That is where interfaces come in.

If a class wants to enable sorting for its objects, it should implement the Comparable interface. There is a technical point about this interface. We want to compare strings against strings, employees against employees, and so on. For that reason, the Comparable interface has a type parameter:

```java
public interface Comparable<T> {
    int compareTo(T other);
}
```

For example, the String class implements Comparable<String> so that its compareTo method has the signature

```java
int compareTo(String other)
```
NOTE: A type with a type parameter such as Comparable or ArrayList is a generic type. You will learn all about generic types in Chapter 6.

When calling x.compareTo(y), the compareTo method returns an integer value to indicate whether x or y should come first. A positive return value (not necessarily 1) indicates that x should come after y. A negative integer (not necessarily -1) is returned when x should come before y. If x and y are considered equal, the returned value is 0.

Note that the return value can be any integer. That flexibility is useful because it allows you to return a difference of integers. That is handy, provided the difference cannot produce integer overflow.

```java
public class Employee implements Comparable<Employee> {
    public int compareTo(Employee other) {
        return getId() - other.getId(); // Ok if IDs always ≥ 0
    }
}
```

CAUTION: Returning a difference of integers does not always work. The difference can overflow for large operands of opposite sign. In that case, use the Integer.compare method that works correctly for all integers. However, if you know that the integers are non-negative, or their absolute value is less than Integer.MAX_VALUE / 2, then the difference works fine.

When comparing floating-point values, you cannot just return the difference. Instead, use the static Double.compare method. It does the right thing, even for ±∞ and NaN.

Here is how the Employee class can implement the Comparable interface, ordering employees by salary:

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

NOTE: It is perfectly legal for the compare method to access other.salary. In Java, a method can access private features of any object of its class.
The `String` class, as well as over a hundred other classes in the Java library, implements the `Comparable` interface. You can use the `Arrays.sort` method to sort an array of `Comparable` objects:

```java
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends); // friends is now ["Mary", "Paul", "Peter"]
```

**NOTE:** Strangely, the `Arrays.sort` method does not check at compile time whether the argument is an array of `Comparable` objects. Instead, it throws an exception if it encounters an element of a class that doesn’t implement the `Comparable` interface.

### 3.3.2 The `Comparator` Interface

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

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

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

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

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

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

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

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

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

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

Now the array is either `("Paul", "Mary", "Peter")` or `("Mary", "Paul", "Peter")`.

You will see in Section 3.4.2, “Functional Interfaces” (page 115) how to use a `Comparator` much more easily, using a lambda expression.

### 3.3.3 The Runnable Interface

At a time when just about every processor has multiple cores, you want to keep those cores busy. You may want to run certain tasks in a separate thread, or give them to a thread pool for execution. To define the task, you implement the `Runnable` interface. This interface has just one method.

```java
class HelloTask implements Runnable {
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.println("Hello, World!");
        }
    }
}
```

If you want to execute this task in a new thread, create the thread from the `Runnable` and start it.

```java
Runnable task = new HelloTask();
Thread thread = new Thread(task);
thread.start();
```

Now the `run` method executes in a separate thread, and the current thread can proceed with other work.

**NOTE:** In Chapter 10, you will see other ways of executing a `Runnable`.

**NOTE:** There is also a `Callable<T>` interface for tasks that return a result of type `T`.

### 3.3.4 User Interface Callbacks

In a graphical user interface, you have to specify actions to be carried out when the user clicks a button, selects a menu option, drags a slider, and so on. These actions are often called *callbacks* because some code gets called back when a user action occurs.
In Java-based GUI libraries, interfaces are used for callbacks. For example, in JavaFX, the following interface is used for reporting events:

```java
public interface EventHandler<T> {
    void handle(T event);
}
```

This too is a generic interface where \( T \) is the type of event that is being reported, such as an `ActionEvent` for a button click.

To specify the action, implement the interface:

```java
class CancelAction implements EventHandler<ActionEvent> {
    public void handle(ActionEvent event) {
        System.out.println("Oh noes!" preferring\);\n    }
}
```

Then, make an object of that class and add it to the button:

```java
Button cancelButton = new Button("Cancel");
cancelButton.setOnAction(new CancelAction());
```

**NOTE:** Since Oracle positions JavaFX as the successor to the Swing GUI toolkit, I use JavaFX in these examples. (Don't worry—you need not know any more about JavaFX than the couple of statements you just saw.) The details don't matter; in every user interface toolkit, be it Swing, JavaFX, or Android, you give a button some code that you want to run when the button is clicked.

Of course, this way of defining a button action is rather tedious. In other languages, you just give the button a function to execute, without going through the detour of making a class and instantiating it. The next section shows how you can do the same in Java.

### 3.4 Lambda Expressions

A *lambda expression* is a block of code that you can pass around so it can be executed later, once or multiple times. In the preceding sections, you have seen many situations where it is useful to specify such a block of code:

- To pass a comparison method to `Arrays.sort`
- To run a task in a separate thread
- To specify an action that should happen when a button is clicked
However, Java is an object-oriented language where (just about) everything is an object. There are no function types in Java. Instead, functions are expressed as objects, instances of classes that implement a particular interface. Lambda expressions give you a convenient syntax for creating such instances.

### 3.4.1 The Syntax of Lambda Expressions

Consider again the sorting example from Section 3.3.2, “The Comparator Interface” (page 111). We pass code that checks whether one string is shorter than another. We compute

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

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

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

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

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

```java
λfirst. λsecond. first.length() - second.length()
```

**NOTE:** Why the letter λ? Did Church run out of letters of the alphabet? Actually, the venerable *Principia Mathematica* (see [http://plato.stanford.edu/entries/principia-mathematica](http://plato.stanford.edu/entries/principia-mathematica)) used the ^ accent to denote function parameters, which inspired Church to use an uppercase lambda Λ. But in the end, he switched to the lowercase version. Ever since, an expression with parameter variables has been called a lambda expression.

If the body of a lambda expression carries out a computation that doesn’t fit in a single expression, write it exactly like you would have written a method: enclosed in `{}` and with explicit `return` statements. For example,
(String first, String second) -> {
    int difference = first.length() < second.length();
    if (difference < 0) return -1;
    else if (difference > 0) return 1;
    else return 0;
}

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

Runnable task = () -> { for (int i = 0; i < 1000; i++) doWork(); }

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

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

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

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

EventHandler<ActionEvent> listener = event ->
    System.out.println("Oh noes!");
    // Instead of (event) -> or (ActionEvent event) ->

You never specify the result type of a lambda expression. However, the compiler infers it from the body and checks that it matches the expected type. For example, the expression

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

can be used in a context where a result of type int is expected (or a compatible type such as Integer, long, or double).

3.4.2 Functional Interfaces

As you already saw, there are many interfaces in Java that express actions, such as Runnable or Comparator. Lambda expressions are compatible with these interfaces.

You can supply a lambda expression whenever an object of an interface with a single abstract method is expected. Such an interface is called a functional interface.
To demonstrate the conversion to a functional interface, consider the `Arrays.sort` method. Its second parameter requires an instance of `Comparator`, an interface with a single method. Simply supply a lambda:

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

Behind the scenes, the second parameter variable of the `Arrays.sort` method receives an object of some class that implements `Comparator<String>`. Invoking the `compare` method on that object executes the body of the lambda expression. The management of these objects and classes is completely implementation-dependent and highly optimized.

In most programming languages that support function literals, you can declare function types such as `(String, String) -> int`, declare variables of those types, put functions into those variables, and invoke them. In Java, there is only one thing you can do with a lambda expression: put it in a variable whose type is a functional interface, so that it is converted to an instance of that interface.

**NOTE:** You cannot assign a lambda expression to a variable of type `Object`, the common supertype of all classes in Java (see Chapter 4). `Object` is a class, not a functional interface.

The Java API provides a large number of functional interfaces (see Section 3.6.2, “Choosing a Functional Interface,” page 120). One of them is

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

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

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

### 3.5 Method and Constructor References

Sometimes, there is already a method that carries out exactly the action that you’d like to pass on to some other code. There is special syntax for a `method reference` that is even shorter than a lambda expression calling the method. A similar shortcut exists for constructors. You will see both in the following sections.
3.5.1 Method References

Suppose you want to sort strings regardless of letter case. You could call

    Arrays.sort(strings, (x, y) -> x.compareToIgnoreCase(y));

Instead, you can pass this method expression:

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

The expression String::compareToIgnoreCase is a *method reference* that is equivalent to the lambda expression (x, y) -> x.compareToIgnoreCase(y).

Here is another example. The Objects class defines a method `isNull`. The call `Objects.isNull(x)` simply returns the value of `x == null`. It seems hardly worth having a method for this case, but it was designed to be passed as a method expression. The call

    list.removeIf(Objects::isNull);

removes all null values from a list.

As another example, suppose you want to print all elements of a list. The ArrayList class has a method `forEach` that applies a function to each element. You could call

    list.forEach(x -> System.out.println(x));

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

    list.forEach(System.out::println);

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

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

In the first case, the first parameter becomes the receiver of the method, and any other parameters are passed to the method. For example, `String::compareToIgnoreCase` is the same as `(x, y) -> x.compareToIgnoreCase(y).

In the second case, all parameters are passed to the static method. The method expression `Objects::isNull` is equivalent to `x -> Objects.isNull(x).

In the third case, the method is invoked on the given object, and the parameters are passed to the instance method. Therefore, `System.out::println` is equivalent to `x -> System.out.println(x).`
NOTE: When there are multiple overloaded methods with the same name, the compiler will try to find from the context which one you mean. For example, there are multiple versions of the `println` method. When passed to the `forEach` method of an `ArrayList<String>`, the `println(String)` method is picked.

You can capture the `this` parameter in a method reference. For example, `this::equals` is the same as `x -> this.equals(x)`.

NOTE: In an inner class, you can capture the `this` reference of an enclosing class as `EnclosingClass.this::method`. You can also capture `super`—see Chapter 4.

3.5.2 Constructor References

Constructor references are just like method references, except that the name of the method is `new`. For example, `Employee::new` is a reference to an `Employee` constructor. If the class has more than one constructor, then it depends on the context which constructor is chosen.

Here is an example for using such a constructor reference. Suppose you have a list of strings

```java
List<String> names = ...;
```

You want a list of employees, one for each name. As you will see in Chapter 8, you can use streams to do this without a loop: Turn the list into a stream, and then call the `map` method. It applies a function and collects all results.

```java
Stream<Employee> stream = names.stream().map(Employee::new);
```

Since `names.stream()` contains `String` objects, the compiler knows that `Employee::new` refers to the constructor `Employee(String)`.

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

Array constructor references are useful to overcome a limitation of Java: It is not possible to construct an array of a generic type. (See Chapter 6 for details.)

For that reason, methods such as `Stream.toArray` return an `Object` array, not an array of the element type:

```java
Object[] employees = stream.toArray();
```
But that is unsatisfactory. The user wants an array of employees, not objects. To solve this problem, another version of `toArray` accepts a constructor reference:

```java
Employee[] buttons = stream.toArray(Employee[]::new);
```

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

### 3.6 Processing Lambda Expressions

Up to now, you have seen how to produce lambda expressions and pass them to a method that expects a functional interface. In the following sections, you will see how to write your own methods that can consume lambda expressions.

#### 3.6.1 Implementing Deferred Execution

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

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

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

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

To accept the lambda, we need to pick (or, in rare cases, provide) a functional interface. In this case, we can just use `Runnable`:

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

Note that the body of the lambda expression is executed when `action.run()` is called.
Now let’s make this example a bit more sophisticated. We want to tell the action in which iteration it occurs. For that, we need to pick a functional interface that has a method with an int parameter and a void return. Instead of rolling your own, I strongly recommend that you use one of the standard ones described in the next section. The standard interface for processing int values is

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

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

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

And here is how you call it:

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

### 3.6.2 Choosing a Functional Interface

In most functional programming languages, function types are **structural**. To specify a function that maps two strings to an integer, you use a type that looks something like `Function2<String, String, Integer>` or `(String, String) -> int`. In Java, you instead declare the intent of the function using a functional interface such as `Comparator<String>`. In the theory of programming languages this is called **nominal** typing.

Of course, there are many situations where you want to accept “any function” without particular semantics. There are a number of generic function types for that purpose (see Table 3-1), and it’s a very good idea to use one of them when you can.

For example, suppose you write a method to process files that match a certain criterion. Should you use the descriptive `java.io.FileFilter` class or a `Predicate<File>`? I strongly recommend that you use the standard `Predicate<File>`. The only reason not to do so would be if you already have many useful methods producing `FileFilter` instances.
### Table 3-1 Common Functional Interfaces

<table>
<thead>
<tr>
<th>Functional Interface</th>
<th>Parameter types</th>
<th>Return type</th>
<th>Abstract method name</th>
<th>Description</th>
<th>Other methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runnable</td>
<td>none</td>
<td>void</td>
<td>run</td>
<td>Runs an action without arguments or return value</td>
<td></td>
</tr>
<tr>
<td>Supplier&lt;T&gt;</td>
<td>none</td>
<td>T</td>
<td>get</td>
<td>Supplies a value of type T</td>
<td></td>
</tr>
<tr>
<td>Consumer&lt;T&gt;</td>
<td>T</td>
<td>void</td>
<td>accept</td>
<td>Consumes a value of type T</td>
<td>andThen</td>
</tr>
<tr>
<td>BiConsumer&lt;T, U&gt;</td>
<td>T, U</td>
<td>void</td>
<td>accept</td>
<td>Consumes values of types T and U</td>
<td>andThen</td>
</tr>
<tr>
<td>Function&lt;T, R&gt;</td>
<td>T</td>
<td>R</td>
<td>apply</td>
<td>A function with argument of type T</td>
<td>compose, andThen, identity</td>
</tr>
<tr>
<td>BiFunction&lt;T, U, R&gt;</td>
<td>T, U</td>
<td>R</td>
<td>apply</td>
<td>A function with arguments of types T and U</td>
<td>andThen</td>
</tr>
<tr>
<td>UnaryOperator&lt;T&gt;</td>
<td>T</td>
<td>T</td>
<td>apply</td>
<td>A unary operator on the type T</td>
<td>compose, andThen, identity</td>
</tr>
<tr>
<td>BinaryOperator&lt;T&gt;</td>
<td>T, T</td>
<td>T</td>
<td>apply</td>
<td>A binary operator on the type T</td>
<td>andThen, maxBy, minBy</td>
</tr>
<tr>
<td>Predicate&lt;T&gt;</td>
<td>T</td>
<td>boolean</td>
<td>test</td>
<td>A boolean-valued function</td>
<td>and, or, negate, isEqual</td>
</tr>
<tr>
<td>BiPredicate&lt;T, U&gt;</td>
<td>T, U</td>
<td>boolean</td>
<td>test</td>
<td>A boolean-valued function with two arguments</td>
<td>and, or, negate</td>
</tr>
</tbody>
</table>
NOTE: Most of the standard functional interfaces have nonabstract methods for producing or combining functions. For example, Predicate.isEqual(a) is the same as a::equals, but it also works if a is null. There are default methods and, or, negate for combining predicates. For example,

```
Predicate.isEqual(a).or(Predicate.isEqual(b))
```

is the same as

```
x -> a.equals(x) || b.equals(x)
```

Table 3-2 lists the 34 available specializations for primitive types int, long, and double. It is a good idea to use these specializations to reduce autoboxing. For that reason, I used an IntConsumer instead of a Consumer<Integer> in the example of the preceding section.

**Table 3-2** Functional Interfaces for Primitive Types

<table>
<thead>
<tr>
<th>Functional Interface</th>
<th>Parameter types</th>
<th>Return type</th>
<th>Abstract method name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BooleanSupplier</td>
<td>none</td>
<td>boolean</td>
<td>getAsBoolean</td>
</tr>
<tr>
<td>PSupplier</td>
<td>none</td>
<td>p</td>
<td>getAsP</td>
</tr>
<tr>
<td>PConsumer</td>
<td>p</td>
<td>void</td>
<td>accept</td>
</tr>
<tr>
<td>ObjPConsumer&lt;T&gt;</td>
<td>T, p</td>
<td>void</td>
<td>accept</td>
</tr>
<tr>
<td>PFunction&lt;T&gt;</td>
<td>p</td>
<td>T</td>
<td>apply</td>
</tr>
<tr>
<td>PToQFunction</td>
<td>p</td>
<td>q</td>
<td>applyAsQ</td>
</tr>
<tr>
<td>ToPFunction&lt;T&gt;</td>
<td>T</td>
<td>p</td>
<td>applyAsP</td>
</tr>
<tr>
<td>ToPBiFunction&lt;T, U&gt;</td>
<td>T, U</td>
<td>p</td>
<td>applyAsP</td>
</tr>
<tr>
<td>PUnaryOperator</td>
<td>p</td>
<td>p</td>
<td>applyAsP</td>
</tr>
<tr>
<td>PBinaryOperator</td>
<td>p, p</td>
<td>p</td>
<td>applyAsP</td>
</tr>
<tr>
<td>PPredicate</td>
<td>p</td>
<td>boolean</td>
<td>test</td>
</tr>
</tbody>
</table>
3.6.3 Implementing Your Own Functional Interfaces

Ever so often, you will be in a situation where none of the standard functional interfaces work for you. Then you need to roll your own.

Suppose you want to fill an image with color patterns, where the user supplies a function yielding the color for each pixel. There is no standard type for a mapping \((\text{int}, \text{int}) \rightarrow \text{Color}\). You could use \(\text{BiFunction< Integer, Integer, Color>}\), but that involves autoboxing.

In this case, it makes sense to define a new interface

```java
@FunctionalInterface
public interface PixelFunction {
    Color apply(int x, int y);
}
```

**NOTE:** It is a good idea to tag functional interfaces with the `@FunctionalInterface` annotation. This has two advantages. First, the compiler checks that the annotated entity is an interface with a single abstract method. Second, the javadoc page includes a statement that your interface is a functional interface.

Now you are ready to implement a method:

```java
BufferedImage createImage(int width, int height, PixelFunction f) {
    BufferedImage image = new BufferedImage(width, height,
                                            BufferedImage.TYPE_INT_RGB);

    for (int x = 0; x < width; x++)
        for (int y = 0; y < height; y++) {
            Color color = f.apply(x, y);
            image.setRGB(x, y, color.getRGB());
        }
    return image;
}
```

To call it, supply a lambda expression that yields a color value for two integers:

```java
BufferedImage frenchFlag = createImage(150, 100,
                                        (x, y) -> x < 50 ? Color.BLUE : x < 100 ? Color.WHITE : Color.RED);
```
3.7 Lambda Expressions and Variable Scope

In the following sections, you will learn how variables work inside lambda expressions. This information is somewhat technical but essential for working with lambda expressions.

3.7.1 Scope of a Lambda Expression

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

```java
int first = 0;
Comparator<String> comp = (first, second) -> first.length() - second.length();
// Error: Variable first already defined
```

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

As another consequence of the “same scope” rule, the this keyword in a lambda expression denotes the this parameter of the method that creates the lambda. For example, consider

```java
public class Application() {
    public void doWork() {
        Runnable runner = () -> { ...; System.out.println(this.toString()); ...};
        ...
    }
}
```

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

3.7.2 Accessing Variables from the Enclosing Scope

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

```java
public static void repeatMessage(String text, int count) {
    Runnable r = () -> {
        for (int i = 0; i < count; i++) {
            System.out.println(text);
        }
    };  
    new Thread(r).start();
}
```
Note that the lambda expression accesses the parameter variables defined in the enclosing scope, not in the lambda expression itself.

Consider a call

```java
repeatMessage("Hello", 1000); // Prints Hello 1000 times in a separate thread
```

Now look at the variables `count` and `text` inside the lambda expression. If you think about it, something nonobvious is going on here. The code of the lambda expression may run long after the call to `repeatMessage` has returned and the parameter variables are gone. How do the `text` and `count` variables stay around when the lambda expression is ready to execute?

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

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

In our example, the lambda expression has two free variables, `text` and `count`. The data structure representing the lambda expression must store the values for these variables—in our case, "Hello" and 1000. We say that these values have been captured by the lambda expression. (It's an implementation detail how that is done. For example, one can translate a lambda expression into an object with a single method, so that the values of the free variables are copied into instance variables of that object.)

**NOTE:** The technical term for a block of code together with the values of free variables is a closure. In Java, lambda expressions are closures.

As you have seen, a lambda expression can capture the value of a variable in the enclosing scope. To ensure that the captured value is well defined, there is an important restriction. In a lambda expression, you can only reference variables whose value doesn't change. This is sometimes described by saying that lambda expressions capture values, not variables. For example, the following is a compile-time error:

```java
for (int i = 0; i < n; i++) {
    new Thread(() -> System.out.println(i)).start();
    // Error—cannot capture i
}
```

The lambda expression tries to capture `i`, but this is not legal because `i` changes. There is no single value to capture. The rule is that a lambda
expression can only access local variables from an enclosing scope that are 
*effectively final*. An effectively final variable is never modified—it either is or 
could be declared as final.

**NOTE:** The same rule applies to variables captured by local classes (see 
Section 3.9, “Local and Anonymous Classes,” page 129). In the past, the 
rule was more draconian and required captured variables to actually be 
declared final. This is no longer the case.

**NOTE:** The variable of an enhanced *for* loop is effectively final since its 
scope is a single iteration. The following is perfectly legal:

```java
for (String arg : args) {
    new Thread(() -> System.out.println(arg)).start();
    // OK to capture arg
}
```

A new variable `arg` is created in each iteration and assigned the next 
value from the `args` array. In contrast, the scope of the variable `i` in the 
preceding example was the entire loop.

As a consequence of the “effectively final” rule, a lambda expression cannot 
mutate any captured variables. For example,

```java
public static void repeatMessage(String text, int count, int threads) {
    Runnable r = () -> {
        while (count > 0) {
            count--; // Error: Can’t mutate captured variable
            System.out.println(text);
        }
    };
    for (int i = 0; i < threads; i++) new Thread(r).start();
}
```

This is actually a good thing. As you will see in Chapter 10, if two threads 
update `count` at the same time, its value is undefined.

**NOTE:** Don’t count on the compiler to catch all concurrent access errors. 
The prohibition against mutation only holds for local variables. If `count` 
is an instance variable or static variable of an enclosing class, then no 
error is reported even though the result is just as undefined.
CAUTION: One can circumvent the check for inappropriate mutations by using an array of length 1:

```java
int[] counter = new int[1];
button.setOnAction(event -> counter[0]++);
```

The `counter` variable is effectively final—it is never changed since it always refers to the same array, so you can access it in the lambda expression.

Of course, code like this is not threadsafe. Except possibly for a callback in a single-threaded UI, this is a terrible idea. You will see how to implement a threadsafe shared counter in Chapter 10.

### 3.8 Higher-Order Functions

In a functional programming language, functions are first-class citizens. Just like you can pass numbers to methods and have methods that produce numbers, you can have arguments and return values that are functions. Functions that process or return functions are called higher-order functions. This sounds abstract, but it is very useful in practice. Java is not quite a functional language because it uses functional interfaces, but the principle is the same. In the following sections, we will look at some examples and examine the higher-order functions in the `Comparator` interface.

#### 3.8.1 Methods that Return Functions

Suppose sometimes we want to sort an array of strings in ascending order and other times in descending order. We can make a method that produces the correct comparator:

```java
public static Comparator<String> compareInDirection(int direction) {
    return (x, y) -> direction * x.compareTo(y);
}
```

The call `compareInDirection(1)` yields an ascending comparator, and the call `compareInDirection(-1)` a descending comparator.

The result can be passed to another method (such as `Arrays.sort`) that expects such an interface.

```java
Arrays.sort(friends, compareInDirection(-1));
```

In general, don’t be shy to write methods that produce functions (or, technically, instances of classes that implement a functional interface). This is useful to generate custom functions that you pass to methods with functional interfaces.
3.8.2 Methods That Modify Functions

In the preceding section, you saw a method that yields an increasing or decreasing string comparator. We can generalize this idea by reversing any comparator:

```java
public static Comparator<String> reverse(Comparator<String> comp) {
    return (x, y) -> comp.compare(y, x);
}
```

This method operates on functions. It receives a function and returns a modified function. To get case-insensitive descending order, use

```java
reverse(String::compareToIgnoreCase)
```

**NOTE:** The Comparator interface has a default method reversed that produces the reverse of a given comparator in just this way.

3.8.3 Comparator Methods

The Comparator interface has a number of useful static methods that are higher-order functions generating comparators.

The comparing method takes a “key extractor” function that maps a type T to a comparable type (such as String). The function is applied to the objects to be compared, and the comparison is then made on the returned keys. For example, suppose a Person class has a method getLastName. Then you can sort an array of Person objects by last name like this:

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

You can chain comparators with the thenComparing method to break ties. For example, sort an array of people by last name, then use the first name for people with the same last name:

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

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

```java
Arrays.sort(people, Comparator.comparing(Person::getLastName,
            (s, t) -> s.length() - t.length()));
```

Moreover, both the comparing and thenComparing methods have variants that avoid boxing of int, long, or double values. An easier way of sorting by name length would be
Arrays.sort(people, Comparator.comparingInt(p -> p.getLastName().length()));

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

```java
Comparator.comparing(Person::getMiddleName, Comparator.nullsFirst(...));
```

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

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

The static reverseOrder method gives the reverse of the natural order.

### 3.9 Local and Anonymous Classes

Long before there were lambda expressions, Java had a mechanism for concisely defining classes that implement an interface (functional or not). For functional interfaces, you should definitely use lambda expressions, but once in a while, you may want a concise form for an interface that isn't functional. You will also encounter the classic constructs in legacy code.

#### 3.9.1 Local Classes

You can define a class inside a method. Such a class is called a *local class*. You would do this for classes that are just tactical. This occurs often when a class implements an interface and the caller of the method only cares about the interface, not the class.

For example, consider a method

```java
public static IntSequence randomInts(int low, int high)
```

that generates an infinite sequence of random integers with the given bounds. Since IntSequence is an interface, the method must return an object of some class implementing that interface. The caller doesn’t care about the class, so it can be declared inside the method:
private static Random generator = new Random();

public static IntSequence randomInts(int low, int high) {
    class RandomSequence implements IntSequence {
        public int next() { return low + generator.nextInt(high - low + 1); }
        public boolean hasNext() { return true; }
    }

    return new RandomSequence();
}

NOTE: A local class is not declared as public or private since it is never accessible outside the method.

There are two advantages of making a class local. First, its name is hidden in the scope of the method. Second, the methods of the class can access variables from the enclosing scope, just like the variables of a lambda expression.

In our example, the next method captures three variables: low, high, and generator. If you turned RandomInt into a nested class, you would have to provide an explicit constructor that receives these values and stores them in instance variables (see Exercise 16).

3.9.2 Anonymous Classes

In the example of the preceding section, the name RandomSequence was used exactly once: to construct the return value. In this case, you can make the class anonymous:

```
public static IntSequence randomInts(int low, int high) {
    return new IntSequence() {
        public int next() { return low + generator.nextInt(high - low + 1); }
        public boolean hasNext() { return true; }
    }
}
```

The expression

```
new Interface() { methods }
```

means: Define a class implementing the interface that has the given methods, and construct one object of that class.

NOTE: As always, the () in the new expression indicate the construction arguments. A default constructor of the anonymous class is invoked.
Before Java had lambda expressions, anonymous inner classes were the most concise syntax available for providing runnables, comparators, and other functional objects. You will often see them in legacy code.

Nowadays, they are only necessary when you need to provide two or more methods, as in the preceding example. If the IntSequence interface has a default hasNext method, as in Exercise 16, you can simply use a lambda expression:

```java
public static IntSequence randomInts(int low, int high) {
    return () -> low + generator.nextInt(high - low + 1);
}
```

### Exercises

1. Provide an interface Measurable with a method double getMeasure() that measures an object in some way. Make Employee implement Measurable. Provide a method double average(Measurable[] objects) that computes the average measure. Use it to compute the average salary of an array of employees.

2. Continue with the preceding exercise and provide a method Measurable largest(Measurable[] objects). Use it to find the name of the employee with the largest salary. Why do you need a cast?

3. What are all the supertypes of String? Of Scanner? Of ImageOutputStream? Note that each type is its own supertype. A class or interface without declared supertype has supertype Object.

4. Implement a static of method of the IntSequence class that yields a sequence with the arguments. For example, IntSequence.of(3, 1, 4, 1, 5, 9) yields a sequence with six values. Extra credit if you return an instance of an anonymous inner class.

5. Add a static method with the name constant of the IntSequence class that yields an infinite constant sequence. For example, IntSequence.constant(1) yields values 1 1 1..., ad infinitum. Extra credit if you do this with a lambda expression.

6. The SquareSequence class doesn’t actually deliver an infinite sequence of squares due to integer overflow. Specifically, how does it behave? Fix the problem by defining a Sequence<T> interface and a SquareSequence class that implements Sequence<BigInteger>.

7. In this exercise, you will try out what happens when a method is added to an interface. In Java 7, implement a class DigitSequence that implements Iterator<Integer>, not IntSequence. Provide methods hasNext, next, and a do-nothing remove. Write a program that prints the elements of an instance.
In Java 8, the Iterator class gained another method, forEachRemaining. Does your code still compile when you switch to Java 8? If you put your Java 7 class in a JAR file and don't recompile, does it work in Java 8? What if you call the forEachRemaining method? Also, the remove method has become a default method in Java 8, throwing an UnsupportedOperationException. What happens when remove is called on an instance of your class?

8. Implement the method void luckySort(ArrayList<String> strings, Comparator<String> comp) that keeps calling Collections.shuffle on the array list until the elements are in increasing order, as determined by the comparator.

9. Implement a class Greeter that implements Runnable and whose run method prints n copies of "Hello, " + target, where n and target are set in the constructor. Construct two instances with different messages and execute them concurrently in two threads.

10. Implement methods

   ```java
   public static void runTogether(Runnable... tasks)
   public static void runInOrder(Runnable... tasks)
   ```

   The first method should run each task in a separate thread and then return. The second method should run all methods in the current thread and return when the last one has completed.

11. Using the listFiles(FileFilter) and isDirectory methods of the java.io.File class, write a method that returns all subdirectories of a given directory. Use a lambda expression instead of a FileFilter object. Repeat with a method expression and an anonymous inner class.

12. Using the list(FilenameFilter) method of the java.io.File class, write a method that returns all files in a given directory with a given extension. Use a lambda expression, not a FilenameFilter. Which variable from the enclosing scope does it capture?

13. Given an array of File objects, sort it so that directories come before files, and within each group, elements are sorted by path name. Use a lambda expression to specify the Comparator.

14. Write a method that takes an array of Runnable instances and returns a Runnable whose run method executes them in order. Return a lambda expression.

15. Write a call to Arrays.sort that sorts employees by salary, breaking ties by name. Use Comparator.thenComparing. Then do this in reverse order.

16. Implement the RandomSequence in Section 3.9.1, “Local Classes” (page 129) as a nested class, outside the randomInts method.
Symbols and Numbers

- (minus sign)
  - flag (for output), 35
  - in dates, 414
  - in regular expressions, 310
  - operator, 17–18

--
  - in command-line options, 81
  - in shell scripts, 463
  - operator, 17, 19

->, in lambda expressions, 114, 117

–, in string templates, 434

_ (underscore)
  - in number literals, 12
  - in variable names, 14, 67

, (comma)
  - flag (for output), 35
  - in numbers, 422, 428, 433
  - normalizing, 317
  - trailing, in arrays, 44

; (semicolon)
  - in Java vs. JavaScript, 450
  - path separator (Windows), 81, 248

: (colon)
  - in assertions, 194
  - in dates, 414
  - in switch statement, 37
  - path separator (Unix), 81, 248

:: operator, 117, 145

! (exclamation sign)
  - comments, in property files, 247
  - operator, 17, 22

!= operator, 17, 22
  - for wrapper classes, 47

? (quotation mark)
  - in regular expressions, 310–311, 313
  - replacement character, 295, 438
  - wildcard, for types, 212–216, 227

? : operator, 17, 22

/ (slash)
  - file separator (Unix), 248, 298
  - in javac path segments, 5
  - operator, 17–18
  - root component, 298

//, /*...*/ comments, 3

/**...*/ comments, 90–91

/* operator, 17
Index

. (period)
in method calls, 6
in numbers, 422, 428, 433
in package names, 5, 79
in regular expressions, 310–311, 319
operator, 17
..., parent directory, 299
...(ellipsis), for varargs, 54
`...` (back quotes), in shell scripts, 462
^ (caret)
for function parameters, 114
in regular expressions, 310–313, 318
operator, 17, 23
^= operator, 17
~ (tilde), operator, 17, 23
'.' (single quotes)
for character literals, 13–14
in JavaScript, 450
in string templates, 434
"..." (double quotes)
for strings, 6
in javadoc hyperlinks, 94
in shell scripts, 462
"" (empty string), 26–27, 147
( (left parenthesis), in formatted output, 35
(...)(parentheses)
empty, for anonymous classes, 130
for anonymous functions (JavaScript), 458
for casts, 21, 103
in regular expressions, 310–313, 316–317
operator, 17
[...](square brackets)
for arrays, 43–44, 50
in JavaScript, 454, 457–458
in regular expressions, 310–312
operator, 17
{...} (curly braces)
in annotation elements, 379
in lambda expressions, 114
in regular expressions, 310–313, 318
in string templates, 433
with arrays, 44
{"..."}, double brace initialization, 144
@ (at)
in java command, 487
in javadoc comments, 91
$ (dollar sign)
currency symbol, 433
flag (for output), 36
in JavaScript function calls, 455, 460
in regular expressions, 310–311, 313, 318
in variable names, 14
${...}, in shell scripts, 462–463
€ currency symbol, 428, 433
* (asterisk)
for annotation processors, 394
in documentation comments, 92
in regular expressions, 310–313, 317
operator, 17–18
wildcard:
in class path, 81
in option files, 487

\ (backslash)
character literal, 14
file separator (Windows), 248, 298
in option files, 487
in regular expressions, 310–311, 318
\= operator, 17
\= (double ampersand)
in regular expressions, 312
operator, 17, 22
\= operator, 17
# (number sign)
comments, in property files, 247
flag (for output), 35
in javadoc hyperlinks, 93
in option files, 487
in string templates, 434
#!, in shell scripts, 464
% (percent sign)
conversion character, 34–35
operator, 17–18
%% pattern variable, 202
%= operator, 17
+ (plus sign)
flag (for output), 35
in regular expressions, 310–313
operator, 17–18
   for strings, 24–25, 27, 147
++ operator, 17, 19
+= operator, 17
< (left angle bracket)
   flag (for output), 36
   in shell syntax, 33
   in string templates, 434
operator, 22, 456
<< operator, 17, 23
<<= operator, 17
<= operator, 17, 22
<< operator, 17, 22
<s...%>, <%=...%> delimiters (JSP), 464
s, in string templates, 434
:o (diamond syntax)
   for array lists, 45
   for constructors of generic classes, 209
...> (angle brackets)
   for type parameters, 109, 208
   in javadoc hyperlinks, 93
   in regular expressions, 313
=, -= operators, 17–18
== operator, 17, 22, 149
   for class objects, 160
   for enumerations, 155
   for strings, 26
   for wrapper classes, 47
=>, in JavaScript, 459
> (right angle bracket)
   in shell syntax, 33
operator, 22
>>, >>> operators, 17, 22–23
>>> operators, 17
| (vertical bar)
   in regular expressions, 310–312
   in string templates, 434
operator, 17, 23
|= operator, 17
|| operator, 17, 22
0 (zero)
   as default value, 71, 74
   flag (for output), 35
   formatting symbol (date/time), 416
   prefix, for octal literals, 11
\0, in regular expressions, 311
0b prefix, 11
0x prefix, 11, 35
0xFEFF byte order mark, 291

A
a formatting symbol (date/time), 416
a, A conversion characters, 34
\a, \A, in regular expressions, 311, 314
abstract classes, 141–142
abstract methods, of an interface, 115
abstract modifier, 103, 141–142
AbstractCollection class, 106
AbstractMethodError, 107
AbstractProcessor class, 394
accept methods (Consumer, XxxConsumer), 121–122
acceptEither method (CompletableFuture), 339–340
AccessibleObject class, 172
   setAccessible method, 170, 172
   trySetAccessible method, 170
accessors, 62
accumulate method (LongAccumulator), 356
accumulateAndGet method (AtomicXxx), 355
accumulator functions, 278
add method
   of ArrayDeque, 250
   of ArrayList, 46, 62
   of BlockingQueue, 353
   of Collection, 236
   of List, 238
   of ListIterator, 241
   of LongAdder, 356
addAll method
   of Collection, 214, 236
   of Collections, 239
   of List, 238
addExact method (Math), 20
addHandler method (Logger), 200
addition, 18
   identity for, 278
addSuppressed method (IOException), 189
aggregators, 488
allMatch method (Stream), 267
allOf method
   of CompletableFuture, 339–340
   of EnumSet, 250
allProcesses method (ProcessHandle), 370
and, andNot methods (BitSet), 249
and, andThen methods (functional interfaces), 121
Android, 341
AnnotatedConstruct interface, 395
AnnotatedElement interface, 392–393
annotation interfaces, 383–386
annotation processors, 394
annotations
accessing, 384
from a different module, 489
and modifiers, 382
container, 389, 392
declaration, 380–381
documented, 389
generating source code with, 395–398
inherited, 389, 392
key/value pairs in. See elements
meta, 384–390
multiple, 380
processing:
at runtime, 391–393
source-level, 394–398
repeatable, 380, 389, 392
standard, 386–390
type use, 381–382
anonymous classes, 130–131
anyMatch method (Stream), 267
anyOf method (CompletableFuture), 339–340
Apache Commons CSV, 484
API documentation, 28–30
generating, 90
Applet class, 163
applications. See programs
apply, applyAsXxx methods (functional interfaces), 121–122
applyToEither method (CompletableFuture), 339–340
$ARG$, in shell scripts, 463
arguments array (jjs), 463
arithmetic operations, 17–24
Array class, 174–175
array list variables, 45
array lists, 45–46
anonymous, 144
checking for nulls, 215
constructing, 45–46
converting between, 212
copying, 48
filling, 49
instantiating with type variables, 222
size of, 46
sorting, 49
visiting all elements of, 47
working with elements of, 46–47
array variables
assigning values to, 45
copying, 47
declaring, 43–44
initializing, 43
ArrayBlockingQueue class, 353
ArrayDeque class, 250
ArrayList class, 45–46, 237
add method, 46, 62
clear method, 154
forEach method, 117
generating Class objects for, 160
get, remove, set, size methods, 46
removeIf method, 116
arrays, 43–45
accessing nonexisting elements in, 43
allocating, 222
annotating, 381
casting, 174
checking, 174
comparing, 149
computing values of, 349
constructing, 43–44
constructor references with, 118
converting:
to a reference of type Object, 146
to/from streams, 271, 281, 350
copying, 48
covariant, 212
filling, 44, 49
generating Class objects for, 160
growing, 174–175
hash codes of, 151
in JavaScript, 457–458
length of, 43, 45, 127
multidimensional, 50–52, 147
of bytes, 288–289
of generic types, 118, 223
of objects, 44, 349
of primitive types, 349
of strings, 317
passing into methods, 53
printing, 49, 52, 147
serializable, 319
sorting, 49, 109–111, 349–350
superclass assignment in, 140
using class literals with, 160
Arrays class
asList method, 253
copyOf method, 48
deepToString method, 147
equals method, 149
fill method, 49
hashCode method, 151
parallelXXX methods, 49, 349
sort method, 49, 111–112, 116–117
stream method, 262, 279
toString method, 49, 147
ArrayStoreException, 140, 212, 223
ASCII, 30, 290
for property files, 437
for source files, 438
ASM tool, 398
assert statement, 194
AssertionError, 194
assertions, 193–195
checking, 381
enabling/disabling, 194–195
assignment operators, 18
associative operations, 278
asString method (HttpResponse), 308
asSubclass method (Class), 227
asynchronous computations, 335–341
AsyncTask class (Android), 341
atomic operations, 346, 351, 354–357, 360
and performance, 355
AtomicXXX classes, 355
atZone method (LocalDateTime), 410
@author tag (javadoc), 91, 95
autoboxing, 46, 123
AutoCloseable interface, 187, 210
close method, 188
availableCharsets method (Charset), 292
availableProcessors method (Runtime), 331
average method (XXXStream), 280
B
b, B conversion characters, 35
\b (backspace), 14
\b, \B, in regular expressions, 314
bash scripts (Unix), 461
BasicFileAttributes class, 303
batch files (Windows), 461
BeanInfo class, 173
between method (Duration), 403
BiConsumer interface, 121
BiFunction interface, 121, 123
BigDecimal class, 13, 23–24
big-endian format, 291, 296–297
BigInteger class, 11, 23–24
binary data, reading/writing, 296
binary numbers, 11, 13
binary trees, 242
BinaryOperator interface, 121
binarySearch method (Collections), 240
Bindings interface, 449
BiPredicate interface, 121
BitSet class, 248–249
collecting streams into, 279
methods of, 249
bitwise operators, 23
block statement, labeled, 41
blocking queues, 352–353
BlockingQueue interface, 353
Boolean class, 46
boolean type, 14
default value of, 71, 74
formatting for output, 35
reading/writing, 296
streams of, 279
BooleanSupplier interface, 122
bootstrap class loader, 163
boxed method (XXXStream), 280
branches, 36–38
break statement, 37, 39–41
bridge methods, 218–219
  clashes of, 225
BufferedReader class, 294
build method (HttpClient), 308
bulk operations, 352
Byte class, 46
  MIN_VALUE, MAX_VALUE constants, 11
toUnsignedInt method, 12
byte codes, 4
  writing to memory, 446–447
byte order mark, 291
byte type, 10–12, 289
  streams of, 279
  type conversions of, 21
ByteArrayClass class, 446
ByteArrayClassLoader class, 447
ByteArrayXxxStream classes, 288–289
ByteBuffer class, 297
bytes
  arrays of, 288–289
  converting to strings, 292
  reading, 289
  writing, 290

c
  c, C conversion characters, 34
  C: \root component, 298
C/C++ programming languages
    #include directive in, 84
    allocating memory in, 346
    integer types in, 11
    pointers in, 63
C# programming language, type parameters in, 215
\c, in regular expressions, 311
CachedRowSetImpl class, 486
calculators, 157
Calendar class, 401
  getFirstDayOfWeek method, 431
  weekends in, 407
calendars, 60
call method (CompilationTask), 445
call by reference, 69
Callable interface, 112
  call method, 333
  extending, 445
callbacks, 112–113, 337
  registering, 335
camel case, 15
cancel method
    of CompletableFuture, 337
    of Future, 333
cancellation requests, 364
CancellationException, 337
cardinality method (BitSet), 249
carriage return, character literal for, 14
case label, 37
cast method (class), 227
casts, 21, 103–104, 140
  and generic types, 220
  annotating, 382
  inserting, 217–218
catch statement, 186–187
  annotating parameters of, 380
  in JavaScript, 461
  in try-with-resources, 189
  no type variables in, 225
ceiling method (NavigableSet), 243
Channel interface, 104
channels, 297
char type, 13–14
  streams of, 279
  type conversions of, 21
Character class, 46
character classes, 310
character encodings, 290–293
  detecting, 292
  localizing, 438
  partial, 292, 295
  platform, 292, 438
character literals, 13–14
characters, 288
  combined, 432
  formatting for output, 34
  normalized, 432–433
  reading/writing, 296
charAt method (String), 31
CharSequence interface, 28
  chars, codePoints methods, 279
  splitting by regular expressions, 263
Charset class
   availableCharsets method, 292
defaultCharset method, 292, 438
displayName method, 438
   forName method, 292
checked exceptions, 183–186
   and generic types, 226
   and no-argument constructors, 171
   combining in a superclass, 185
declaring, 185–186
documenting, 186
   in lambda expressions, 186
   not allowed in a method, 191
   rethrowing, 190
cHECKED views, 221, 254
cHECKEDXXX methods (Collections), 240, 254
Checker Framework, 381
childrenNames method (Preferences), 440
choice indicator, in string templates, 434
Church, Alonzo, 114, 404
Class class, 159–162, 228
   asSubclass, cast methods, 227
   comparing objects of, 160
   forName method, 160–161, 164–165, 184, 192, 447
generic, 227
   getCanonicalName method, 160–161
getClassLoader method, 161
getComponentType method, 161, 174
getConstructor(s) methods, 162, 168, 171, 227
getDeclaredConstructor(s) methods, 162, 168, 227
   getDeclaredField(s) methods, 162
   getDeclaredMethod(s) methods, 162, 171
   getDeclaringClass method, 161
getEnclosingXXX methods, 161
getEnumConstants method, 227
getField(s) methods, 162, 168
   getInterfaces method, 161
   getMethod(s) methods, 162, 168, 171
   getModifiers method, 161
   getName method, 159–161
getPackage, getPackageName methods, 161
getResource method, 163, 435
   getSimpleName method, 161
   getSuperclass method, 161, 227
   getTypeName method, 161
   getTypeParameters method, 228
isXXX methods, 161, 174
newInstance method, 171, 227
toString, toGenericString methods, 161
class declarations
   annotations in, 380, 389
   initialization blocks in, 72–73
class files, 4, 163
   paths of, 79
   processing annotations in, 398
class literals, 160
   no annotations for, 382
   no type variables in, 221
class loaders, 163–164
class objects, 160
class path, 80–81
   problems with, 471
   .class suffix, 160
   ClassCastException, 104, 221
classes, 2, 60
   abstract, 103, 108, 141–142
   accessing from a different module, 489
   adding functionality to, 77
   adding to packages, 83
   anonymous, 130–131
   companion, 106
   compiling on the fly, 446
   constructing objects of, 14
deprecated, 93
   documentation comments for, 90–92
   enumerating members of, 158, 168–169
evolving, 324
   extending, 136–145
      in JavaScript, 459–460
   fields of, 135
   final, 141
generic, 45
   immutable, 28, 347
   implementing, 65–69, 153
   importing, 83–84
   inner, 87–89
classes (cont.)
instances of, 6, 65, 78
loading, 169
local, 129–130
members of, 135
naming, 14–15, 78, 159
nested, 85–90, 382
not known at compile time, 160, 175
protected, 142–143
public, 83, 476
static initialization of, 164
system, 195
testing, 83
utility, 83, 165
wrapper, 46–47
classes win rule, 151
classifier functions, 274
ClassLoader class
defineClass method, 486
extending, 447
findClass, loadClass methods, 164
setXxxAssertionStatus methods, 195
classloader inversion, 165
ClassNotFoundException, 184
CLASSPATH environment variable, 82
clear method
of BitSet, 249
of Collection, 236
of Map, 246
close method
of AutoCloseable, 188
of PrintWriter, 187–188
throwing exceptions, 188
Closeable interface, 104
close method, 188
closures, 125
code element (HTML), in documentation
comments, 91
code generator tools, 388
code points, 31, 290
  turning a string into, 263
code units, 13, 31, 279
  in regular expressions, 311
codePoints method (CharSequence), 279
codePoints, codePointXXX methods (String),
  31–32
Collator class, 27
  methods of, 432
collect method (Stream), 271–272, 279
Collection interface, 106, 236
  add method, 236
  addAll method, 214, 236
  clear method, 236
  contains, containsAll, isEmpty methods, 237
  iterator, spliterator methods, 237
  parallelStream method, 237, 260–261, 280, 348
  remove, removeAll, retainAll methods, 236
  size method, 237
  stream method, 237, 260–261
toArray method, 237
collections, 235–254
generic, 254
  iterating over elements of, 260–261
mutable, 253
processing, 239
serializable, 319
threadsafe, 354
unmodifiable views of, 253–254
  vs. streams, 261
  with given elements, 252
Collections class, 106, 239
  addAll method, 239
  binarySearch method, 240
  copy method, 239
  disjoint method, 240
  fill method, 49, 239
  frequency method, 239
  indexOfSubList, lastIndexOfSubList
    methods, 240
  nCopies method, 237, 239
replaceAll method, 239
reverse, shuffle methods, 49, 240
rotate, swap methods, 240
sort method, 49, 215, 240
synchronized Xxx, unmodifiable Xxx methods, 240
Collector interface, 272
Collectors class, 85
counting method, 275
collecting method, 277
flatMapting method, 276
groupingBy method, 274–277
groupingByConcurrent method, 275, 282
joining method, 272
mapping method, 276
maxBy, minBy methods, 276
partitioningBy method, 275, 277
reducing method, 277
summarizing Xxx methods, 272, 276
summing Xxx methods, 276
toCollection, tolist methods, 272
toConcurrentMap method, 274
toMap method, 273–274
toSet method, 272, 275
collected global object (JavaScript), 454
command-line arguments, 49–50
comments, 3
documentation, 90–95
commonPool method (ForkJoinPool), 335
companion classes, 106
Comparable interface, 109–111, 155, 215, 242
compascal method, 109
streams of, 265
with priority queues, 251
Comparator interface, 85, 111–112, 127–129, 242
comparing, comparing Xxx methods, 128–129
naturalOrder method, 129
nullsFirst, nullsLast methods, 129
reversed method, 128
reverseOrder method, 129
streams of, 265
thenComparing method, 128–129
with priority queues, 251
compareTo method
of Enum, 156
of Instant, 403
of String, 26–27, 109, 431
compareToIgnoreCase method (String), 117
calculateUnsigned method (Integer, Long), 20
compatibility, drawbacks of, 216
Compilable interface, 452
compilation, 4
CompilationTask interface, 444
call method, 445
compile method (Pattern), 315, 318
compiler
instruction reordering in, 343
invoking, 444
compile-time errors, 15
completable futures, 335–340
combining, 340
composing, 337–340
interrupting, 337
CompletableFuture class, 335–340
acceptEither, applyToEither methods, 339–340
allOf, anyOf methods, 339–340
cancel method, 337
complete, completeExceptionally methods, 336
exceptionally method, 338–339
handle method, 339
isDone method, 336
runAfterXxx methods, 339–340
supplyAsync method, 335–337
thenAccept method, 335, 339
thenAcceptBoth, thenCombine methods, 339–340
thenApply, thenApplyAsync, thenCompose methods, 337–339
thenRun method, 339
thenComplete method, 336, 338–339
CompletionStage interface, 339
compose method (functional interfaces), 121
computations
asynchronous, 335–341
mutator, 62
reproducible floating-point, 20
computations (cont.)  
with arbitrary precision, 13  
compute, computeIfXXX methods (Map), 245  
concat method (Stream), 265  
concatenation, 24–25  
objects with strings, 147  
concurrent access errors, 126  
concurrent programming, 329–369  
for scripts, 449  
strategies for, 346  
ConcurrentHashMap class, 350–352, 362  
compute method, 350–352  
computeIfXXX, forEachXXX, merge, putIfAbsent, reduceXXX, searchXXX methods, 351  
keySet, newKeySet methods, 354  
no null values in, 244  
ConcurrentModificationException, 241, 350  
ConcurrentSkipListXXX classes, 354  
conditional operator, 22  
configuration files, 439–441  
editing, 199–200  
locating, 162  
resolving paths for, 299  
confinement, 346  
connect method (URLConnection), 306  
Console class, 33  
console, displaying fonts on, 438  
ConsoleHandler class, 200, 203  
constants, 15–16, 105  
  naming, 15  
  static, 75–76  
  using in another class, 16  
Constructor interface, 168–169  
  getModifiers, getName methods, 168  
  newInstance method, 171–172  
constructor references, 118–119  
  annotating, 382  
constructors, 69–74  
  annotating, 224, 380–381  
  documentation comments for, 90  
  executing, 70  
  for subclasses, 139  
  implementing, 69–70  
  in abstract classes, 142  
  invoking another constructor from, 71  
  no-argument, 73, 139, 171  
overloading, 70–71  
public, 70, 168  
  references in, 348  
Consumer interface, 121  
contains method (String), 28  
contains, containsAll methods (Collection), 237  
containsXXX methods (Map), 246  
Content-Type header, 292  
context class loaders, 164–166  
continue statement, 40–41  
control flow, 36–43  
conversion characters, 34–35  
cooperative cancellation, 364  
copy method  
  of Collections, 239  
  of Files, 290, 301–302, 305  
copyOf method (Arrays), 48  
CopyOnWriteArrayXXX classes, 354  
CORBA (Common Object Request Broker Architecture), 470  
count method (Stream), 261, 266  
counters  
  atomic, 354–357  
  de/incrementing, 189  
counting method (Collectors), 275  
country codes, 275, 424  
covariance, 211  
createBindings method (ScriptEngine), 449  
createInstance method (Util), 165  
createTempXXX methods (Files), 301  
createXXX methods (Files), 300  
critical sections, 346, 357, 363  
Crockford, Douglas, 451  
currencies, 428–429  
  formatting, 433  
Currency class, 428  
current method (ProcessHandle), 370  
D  
d  
  conversion character, 34  
  formatting symbol (date/time), 416  
D suffix, 12  
\d, \D, in regular expressions, 312  
daemon threads, 366
databases, 377
  annotating access to, 388
  persisting objects in, 479
DataInput/Output interfaces, 295–296
  read/writeXxx methods, 296–297, 322
DataXxxStream classes, 296
Data class, 401, 416–417
DateFormat class, 429
  dates
    computing, 407–408
    formatting, 413–416, 422, 429–431, 433
    local, 404–407
    nonexistent, 407, 412, 430
  parsing, 415
datesUntil method (LocalDate), 406–407
DateTimeFormat class, 429–431
dateFormatter class, 413–416
  and legacy classes, 417
  format method, 413, 430
  ofLocalizedXxx methods, 413, 430
  ofPattern method, 415
  parse method, 415
  toFormat method, 415
  withLocale method, 413, 430
DateTimeParseException, 430
daylight savings time, 410–413
DayOfWeek enumeration, 61, 406–407, 411
  getDisplayName method, 415, 430
dayOfWeekInMonth method
    (TemporalAdjusters), 408
deadlocks, 346, 358, 362–363
debugging
  messages for, 183
  overriding methods for, 141
  primary arrays for, 49
  streams, 266
  threads, 366
  using anonymous subclasses for, 143–144
  with assertions, 194
DecimalFormat class, 78
  number format patterns of, 433
decomposition (for classes), 52–54
decomposition modes (for characters), 432
decrement operator, 19
decrementExact method (Math), 20
deep copies, 153
depToStream method (Arrays), 147
default label (in switch), 37, 159
default methods, 106–108
  conflicts of, 107–108, 144–145
  in interfaces, 151
default modifier, 106, 385
defaultCharset method (Charset), 292, 438
defaultXxxObject methods (ObjectXxxStream), 322
defensive programming, 193
deferred execution, 119–120
defineClass method (ClassLoader), 486
delete, deleteIfExists methods (Files), 301
delimiters, for scanners, 294
@Deprecated annotation, 93, 386–387
@deprecated tag (javadoc), 93, 387
decast interface, 238, 250
destroy, destroyForcibly methods
  of Process, 369
  of ProcessHandle, 370
DiagnosticCollector class, 447
DiagnosticListener interface, 447
diamond syntax (<>)
  for array lists, 45
  for constructors of generic classes, 209
directories, 298
  checking for existence, 300, 302
  creating, 300–302
  deleting, 301, 304–305
  moving, 301
  temporary, 301
  user, 299
  visiting, 302–305
  working, 367
directory method (ProcessBuilder), 367
disjoint method (Collections), 239
displayName method (Charset), 438
distinct method (Stream), 265, 282
dividedBy method (Duration), 404
divideUnsigned method (Integer, Long), 20
division, 18
do statement, 38
doc-files directory, 91
documentation comments, 90–95
@Documented annotation, 387, 389
domain names
  for modules, 472
  for packages, 79
dot notation, 6, 16
double brace initialization, 144
Double class, 46
  compare method, 110
  equals method, 149
  isFinite, isnInfinite methods, 13
  NaN, NEGATIVE_INFINITY, POSITIVE_INFINITY values, 13
  parseDouble method, 27
toString method, 27
double type, 12–13
  atomic operations on, 357
  functional interfaces for, 122
  streams of, 279
type conversions of, 20–22
DoubleAccumulator, DoubleAdder classes, 357
DoubleConsumer, DoubleXxxOperator,
  DoublePredicate, DoubleSupplier,
  DoubleToXxxFunction interfaces, 122
DoubleFunction interface, 122, 220
doubles method (Random), 280
DoubleStream class, 279–280
DoubleSummaryStatistics class, 272, 280
doubleValue method (Number), 428
downstream collectors, 275–277, 282
Driver.parentLogger method, 488
dropWhile method (Stream), 265
Duration class
  between method, 403
dividedBy, isZero, isNegative, minus,
    minusXxx, multipliedBy, negated, plus,
    plusXxx methods, 404
  immutability of, 347, 404
  ofXxx methods, 403, 405, 413
toXxx methods, 403
dynamic method lookup, 139–140,
  218–219
dynamically typed languages, 456
E
E constant (Math), 20
e, E
  conversion characters, 34
  formatting symbols (date/time), 416
  \e, \E, in regular expressions, 311
Eclipse, 5
ECMAScript standard, 452, 459
edu global object (JavaScript), 454
effectively final variables, 126
efficiency, and final modifier, 141
Element interface, 395
element method (BlockingQueue), 353
elements (in annotations), 378–379, 385
else statement, 36
em element (HTML), in documentation comments, 91
Emacs text editor, 453
empty method
  of Optional, 269
  of Stream, 262
empty string, 26, 147
  concatenating, 27
encapsulation, 60, 469–471, 479
encodings. See character encodings
end method (Matcher, MatchResult), 315–316
<<END, in shell scripts, 463
endsWith method (String), 28
engine scope, 449
enhanced for loop, 47, 52, 126
  for collections, 241
  for enumerations, 155
  for iterators, 167
  for paths, 300
entering, exiting methods (Logger), 197
Entry class, 217
entrySet method (Map), 246
Enum class, 155–156
enum instances
  adding methods to, 157
  construction, 156
  referred by name, 159
enum keyword, 16, 154
enumeration sets, 250
enumerations, 154–159
  annotating, 380
comparing, 155
constructing, 156
defining, 16
nested inside classes, 158
serialization of, 323
static members of, 157–158
traversing instances of, 155
using in switch, 158

EnumMap, EnumSet classes, 250
$ENV, in shell scripts, 463
environment variables, modifying, 368
equivalents, testing for, 22
equals method
  final, 150
  for subclasses, 149
  for values from different classes, 149
null-safe, 149
of Arrays, 149
of Double, 149
of Instant, 403
of Object, 146, 148–150
of Objects, 149
of String, 25–26
of wrapper classes, 47
overriding, 148–150
symmetric, 150
equalsIgnoreCase method (String), 26
$ERR, in shell scripts, 462
Error class, 183
error messages, for generic methods, 210
eval method (ScriptEngine), 449–451
even numbers, 18
EventHandler interface, 113
Exception class, 183
exceptionally method (CompletableFuture), 338–339
exceptions, 182–193
  and generic types, 225–226
annotating, 382
catching, 186–190
  in JavaScript, 461
chaining, 190–191
checked, 171, 183–186
combining in a superclass, 185
creating, 184
documenting, 186
hierarchy of, 183–185
logging, 198
rethrowing, 189–191
suppressed, 189
throwing, 182–183
uncaught, 192
unchecked, 183
exec method (Runtime), 367
Executable class
  getModifiers method, 172
  getParameters method, 169
ExecutableElement interface, 395
ExecutionException, 333
Executor interface, 338
executor services, 331
default, 335
ExecutorCompletionService class, 334
Executors class, 331
ExecutorService interface, 445
  execute method, 331
    invokeAll, invokeAny methods, 334
exists method (Files), 300, 302
exit function (shell scripts), 464
$EXIT, in shell scripts, 462
exitValue method (Process), 369
exports keyword, 473, 476–479
  qualified, 489
exportSubtree method (Preferences), 440
extends keyword, 104, 136, 210–214
Externalizable interface, 322

F
f conversion character, 34
F suffix, 12
\f, in regular expressions, 311
factory methods, 70, 78
failures, logging, 190
falling through, 37
false value (boolean), 14
  as default value, 71, 74
Field interface, 168–169
  get, getXXX, set, setXXX methods, 170, 172
    getModifiers, getName method, 168, 172
    getType method, 168
fields (instance and static variables), 135
  enumerating, 168–169
final, 344
  provided, 143
public, 168
  retrieving values of, 169–171
  setting, 170
transient, 321
File class, 300
file attributes
  copying, 301
  filtering paths by, 303
file handlers
  configuring, 201–202
  default, 200
file managers, 446
file pointers, 296
file.encoding system property, 292
file.separator system property, 248
FileChannel class
  get, getXXX, open, put, putXXX methods, 297
  lock, tryLock methods, 298
FileFilter class, 120
FileHandler class, 200–203
FileNotFoundException, 183
files
  archiving, 305
  channels to, 297
  checking for existence, 183, 300–302
  closing, 187
  copying/moving, 301–302
  creating, 299–302
  deleting, 301
  empty, 300
  encoding of, 290–291
  locking, 297–298
  memory-mapped, 297
  missing, 447
  random-access, 296–297
  reading from/writing to, 33, 183, 289
  temporary, 301
Files class
  copy method, 290, 301–302, 305
  createTempXXX methods, 301
  createXXX methods, 300
delete, deleteIfExists methods, 301
  exists method, 300, 302
  find method, 302–303
  isXXX methods, 300, 302
  lines method, 263, 293
  list method, 302–303
  move method, 301–302
newBufferedReader method, 294, 448
newBufferedWriter method, 294, 302
newXXXStream methods, 288, 302, 320
read, readNBytes methods, 289
readAllBytes method, 289, 293
readAllLines method, 293
walk method, 302–305
walkFileTree method, 302, 304
write method, 295, 302
FileSystem, FileSystems classes, 305
FileTime class, 417
FileVisitor interface, 304
fill method
  of Arrays, 49
  of Collections, 49, 239
filter method (Stream), 261–263, 267
Filter interface, 202
filtering method (Collectors), 277
final fields, 344
final methods, 347
final modifier, 15, 73, 141
final variables, 343, 347
finalize method
  of Enum, 156
  of Object, 146
finally statement, 189–190
  for locks, 358
financial calculations, 13
find method (Files), 302–303
findAll method (Scanner), 316
findAny method (Stream), 267
findClass method (ClassLoader), 164
findFirst method (Stream), 168, 267
fine method (Logger), 197
first method (SortedSet), 243
first day of week, 431
firstDayOfXXX methods (TemporalAdjusters), 408
flag bits, sequences of, 248
flatMap method
  general concept of, 264
  of Optional, 269–271
  of Stream, 264
flatMap method (Collectors), 276
flip method (BitSet), 249
Floor class, 46
float type, 12–13
  streams of, 279
  type conversions of, 20–22
floating-point types, 12–13
  and binary number system, 13
  comparing, 110
  division of, 18
  formatting for output, 34
  in hexadecimal notation, 13
  type conversions of, 20–22
floor method (NavigableSet), 243
floorMod method (Math), 19
fonts, missing, 438
for statement, 39
  declaring variables for, 42
  enhanced, 47, 52, 126, 155, 241, 300
  multiple variables in, 39
for each loop (JavaScript), 458
forEach method
  of ArrayList, 117
  of Map, 246
forEach, forEachOrdered methods (Stream), 271
forEachXxx methods (ConcurrentHashMap), 352
ForkJoinPool class, 338
  commonPool method, 335
forLanguageTag method (Locale), 426
format method
  of DateTimeFormatter, 413, 430
  of MessageFormat, 433–435
  of String, 427
Format class, 417
format specifiers, 34
formatted output, 33–36
Format class, 203
formatters, for date/time values, 414–415
forms, posting data from, 307–309
forName method
  of Charset, 292
  of Class, 160–161, 164–165, 184, 192, 447
frequency method (Collections), 239
from method (Instant, ZonedDateTime), 416
full indicator, in string templates, 433
Function interface, 121, 273
function keyword (JavaScript), 458
function types, 114
  structural, 120
functional interfaces, 115–116
  as method parameters, 213–214
  common, 121
  contravariant in parameter types, 214
  for primitive types, 122
  implementing, 123
@FunctionalInterface annotation, 123, 386, 388–389
functions, 60
  higher-order, 127–129
Future interface, 334
  cancel, isCancelled, isDone methods, 333
  get method, 333, 335
futures, 333–335
  comappable, 333–340
G
  G formatting symbol (date/time), 416
  g, G conversion characters, 34
  \g, in regular expressions, 314
  %g pattern variable, 202
garbage collector, 251
generate method (Stream), 262, 279
  @Generated annotation, 387–388
generators, converting to streams, 281
generic classes, 45, 208–209
  constructing objects of, 209
  information available at runtime, 227
  instantiating, 209
generic collections, 254
generic constructors, 228
generic methods, 209–210
  calling, 210
  declaring, 210
  information available at runtime, 227
generic type declarations, 228–229
generic types, 110
and exceptions, 225–226
and lambda expressions, 213
and reflection, 226–229
annotating, 381
arrays of, 118
casting, 220
in JVM, 216–219
invariant, 212–213
restrictions on, 220–226
GenericArrayType interface, 228
get method
of Array, 174
of ArrayList, 46
of BitSet, 249
of Field, 170, 172
of FileChannel, 297
of Future, 170, 172
of FileChannel, 297
of Future, 333, 335
of List, 238
of LongAccumulator, 356
of Map, 243, 245
of Optional, 269–271
of Path, 298–300
of Preferences, 440
of ServiceLoader.Provider, 167
of Supplier, 121
of ThreadLocal, 365
GET requests, 308
getAndXxx methods (AtomicXxx), 355
getAnnotation, getAnnotationsByType methods
(AnnotatedConstruct), 395
getAnnotationXxx methods
(AnnotatedElement), 392–393
getAsXxx methods
of OptionalXxx, 280
of XxxSupplier, 122
getAudioClip method (Applet), 163
getAvailableCurrencies method (Currency), 428
getAvailableIds method (ZoneId), 410
getAvailableLocales method (Locale), 425
getAverage method (XxxSummaryStatistics), 272
getBundle method (ResourceBundle), 436–438
getCanonicalName method (Class), 160–161
getClass method (Object), 141, 146, 149, 159, 221, 227
getClassLoader method (Class), 161
getComponentType method (Class), 161, 174
getConstructor(s) methods (Class), 162, 168, 171, 227
getContents method (ListResourceBundle), 437
getContextClassLoader method (Thread), 165
getCountry method (Locale), 274
getCurrencyInstance method (NumberFormat), 427
getDayOfXxx methods
of LocalDate, 61, 406–407
of LocalTime, 409
of ZonedDateTime, 411
getDeclaredAnnotationXxx methods
(AnnotatedElement), 392–393
getDeclaredConstructor(s) methods (Class), 162, 168, 227
getDeclaredField(s) methods (Class), 162
getDeclaredMethod(s) methods (Class), 162, 171
getDeclaringClass method
of Class, 161
of Enum, 156
getDefault method (Locale), 426, 436
getDisplayName method
of Currency, 429
of DayOfWeek, Month, 415, 430
of Locale, 426
getElementsAnnotatedWith method
(RoundEnvironment), 395
getEnclosedElements method (TypeElement), 395
getEnclosingXxx methods (Class), 161
getEngineXxx methods (ScriptEngineManager), 448
getEnumConstants method (Class), 227
gerroStream method (Process), 368
gerStream method (Process), 368
getFile method (Path), 300
gerFile method (RandomAccessFile), 297
getFirstDayOfWeek method (Calendar), 431
getGlobal method (Logger), 195
getHead method (Formatter), 203
getHeaderFields method (URLConnection), 307
getInputStream method
  of URL, 306
  of URLConnection, 307
getInstance method
  of Collator, 432
  of Currency, 428
getInterfaces method (Class), 161
getISOXXX methods (Locale), 425
getLength method (Array), 174
getLogger method (Logger), 196
getMax method (XXXSummaryStatistics), 272
getMethod(s) methods (Class), 162, 168, 171
getMethodCallSyntax method
  (ScriptEngineFactory), 451
getModifiers method
  of Class, 161
  of Constructor, 168
  of Executable, 172
  of Field, 168, 172
  of Method, 168
getMonthXXX methods
  of LocalDate, 61, 406
  of LocalDateTime, 409
  of ZonedDateTime, 412
getName method
  of Class, 159–161
  of Constructor, 168
  of Field, 168, 172
  of Method, 168
  of Parameter, 172
  of Path, 300
  of ParameterDescriptor, 173
getNumberInstance method (NumberFormat), 427
getObject method (ResourceBundle), 437
getOrDefault method (Map), 244–245
getOutputStream method (URLConnection), 306
getPackage, getPackageName methods (Class), 161
getParameters method (Executable), 169
getParent method (Path), 300
getAddress method (FileSystem), 305
getPercentInstance method (NumberFormat), 427
getProperties method (System), 248
getProperty method (System), 163
getPropertyDescriptors method (BeanInfo), 173
getPropertyType, getReadMethod methods
  (PropertyDescriptor), 173
getQualifiedName method (TypeElement), 395
getResource method (Class), 163, 435
getResourceAsStream method of Class, 161–162
  of Module, 481
getRoot method (Path), 300
getSimpleName method
  of Class, 161
  of Element, 395
getString method (ResourceBundle), 436
getSuperclass method (Class), 161, 227
getSuppressed method (IOException), 189
getSymbol method (Currency), 429
getSystemJavaCompiler method (ToolProvider), 444
getTail method (Formatter), 203
getTask method (JavaCompiler), 444–445
type method (Field), 168
typeName method (Class), 161
typeParameters method (Class), 228
getURLs method (URLClassLoader), 163
getValue method (LocalDate), 61
getWriteMethod method (PropertyDescriptor), 173
getXXX methods (Array), 174
getXXX methods (Field), 170, 172
getXXX methods (FileChannel), 297
getXXX methods (Preferences), 440
getXXXInstance methods (NumberFormat), 78
getXXXStream methods (Process), 367
getYear method
  of LocalDate, 406
  of LocalDateTime, 409
  of ZonedDateTime, 412
GlassFish administration tool, 463
Goetz, Brian, 329
Gregorian calendar reform, 406
GregorianCalendar class, 416–417

group method (Matcher, MatchResult), 315, 317

grouping, 274–275

classifier functions of, 274

reducing to numbers, 275

groupingBy method (Collectors), 274–277

groupingByConcurrent method (Collectors), 275, 282

GUI (graphical user interface)

callbacks in, 112–113

long-running tasks in, 340–341

missing fonts in, 438

H

H formatting symbol (date/time), 416

h, H conversion characters, 35

\h, \H, in regular expressions, 312

%h pattern variable, 202

handle method (CompletableFuture), 339

Handler class, 203

Hansen, Per Brinch, 360

hash method (Object), 151

hash codes, 150–151

computing in String class, 150

formatting for output, 35

hash functions, 150–151, 242

hash maps

concurrent, 350–352

weak, 251

hash tables, 242

hashCode method

of Arrays, 151

of Enum, 156

of Object, 146, 148, 150–151

HashMap class, 243

null values in, 244

HashSet class, 242

Hashtable class, 360

hasNext method (Iterator), 240

hasNext, hasNextXxx methods (Scanner), 33, 293

headMap method (SortedMap), 253

headSet method

of NavigableSet, 243

of SortedSet, 243, 253

heap pollution, 221, 254

Hello, World! program, 2

modular, 472–474

helper methods, 216

here documents, 463

hexadecimal numbers, 11, 13

formatting for output, 34

higher method (NavigableSet), 243

higher-order functions, 127–129

hn, hr elements (HTML), in documentation comments, 91

Hoare, Tony, 360

HTML documentation, generating, 398

HTTP connections, 306–309

HttpClient class, 306–309, 335

enabling logging for, 309

HttpHeaders class, 309

HttpResponse class, 308–309

HttpURLConnection class, 306–307

hyperlinks

in documentation comments, 93–94

regular expressions for, 310

I

[I prefix, 147, 160

IANA (Internet Assigned Numbers Authority), 410

IDE (integrated development environment), 3, 5

identity method

of Function, 121, 273

of UnaryOperator, 121

identity values, 278

if statement, 36

ifPresent, ifPresentOrElse methods (Optional), 268

IllegalArgumentException, 194

IllegalStateException, 273, 353

ImageIcon class, 163

images, locating, 162

img element (HTML), in documentation comments, 91

immutability, 346

immutable classes, 347

implements keyword, 101–102
import statement, 7, 83–84
  no annotations for, 382
static, 85
import static statement, 159
importPreferences method (Preferences), 441
InaccessibleObjectException, 170, 481
increment method (LongAdder), 356
increment operator, 19
incrementAndGet method (AtomicXxx), 355
incrementExact method (Math), 20
indexOf method
  of List, 238
  of String, 28
indexOfSubList method (Collections), 240
info method
  of Logger, 195
  of ProcessHandle, 370
inheritance, 136–154
  and default methods, 144–145
  classes win rule, 145, 151
@Inherited annotation, 387, 389
initCause method (Throwable), 191
initialization blocks, 72–73
  static, 76
inlining, 141
inner classes, 87–89
  anonymous, 130–131
  capturing this references in, 118
  invoking methods of outer classes, 88
  local, 126, 129–131
  syntax for, 89
input
  reading, 32–33, 293–294
  redirecting, 449
  setting locales for, 427
  splitting along delimiters, 317
input prompts, 33
input streams, 288
  copying, 290
  obtaining, 288
  reading from, 289
InputStream class, 289
  transferTo method, 290
InputStreamReader class, 293
INSTACE instance (enum types), 323
instance methods, 6, 66–67
instance variables, 65, 67–68
  annotating, 380
  comparing, 149
  default values of, 71–72
  final, 73
  in abstract classes, 142
  in JavaScript, 460
  initializing, 72–73, 139
  not accessible from static methods, 77
  of deserialized objects, 322–324
  protected, 143
  setting, 70
  transient, 321
  vs. local, 72
instanceof operator, 104, 140, 149
  annotating, 382
instances, 6
Instant class, 402
  and legacy classes, 417
  compareTo, equals methods, 403
  from method, 416
  immutability of, 347, 404
  minus, minusXXX, plus, plusXXX methods, 404
  now method, 403
instruction reordering, 343
int type, 10–12
  functional interfaces for, 122
  processing values of, 120
  random number generator for, 6, 38
  streams of, 279
  type conversions of, 20–22
  using class literals with, 160
IntBinaryOperator interface, 122
IntConsumer interface, 120, 122
Integer class, 46
  compare method, 110
  MIN_VALUE, MAX_VALUE constants, 11
  parseInt method, 27, 184
  toString method, 27
  unsigned division in, 12
  xxxUnsigned methods, 20
integer indicator, in string templates, 433
integer types, 10–12
  comparing, 110
  computing, 18, 20
integer types (cont.)
  formatting for output, 34
  in hexadecimal notation, 11
  reading/writing, 296–297
  type conversions of, 20–22
  values of:
    even/odd, 18
    signed, 12
@Api interface declaration, 384–385
@interface keyword, 101
interface methods, 106–108
interfaces, 100–105
  annotating, 380–381
  compatibility of, 107
  declarations of, 100–101
  defining variables in, 105
  documentation comments for, 90
  evolution of, 106
  extending, 104
  functional, 115–116
  implementing, 101–103
    in JavaScript, 459–460
      in scripting engines, 451
    multiple, 105
    methods of, 101–102
  nested, enumerating, 168–169
  no instance variables in, 105
  no redefining methods of the Object class in, 151
  views of, 252
Internet Engineering Task Force, 423
interrupted method (Thread), 364
interrupted status, 364
InterruptedException, 363–364
intersects method (BitSet), 249
IntFunction interface, 122, 220
IntPredicate interface, 122
intrinsic locks, 358–360
ints method (Random), 280
IntSequence interface, 129
IntStream class, 279–280
  parallel method, 280
IntSummaryStatistics class, 272, 280
IntSupplier, IntToXxxFunction,
  IntUnaryOperator interfaces, 122
InvalidClassException, 324
InvalidPathException, 298
Invocable interface, 450
InvocationHandler interface, 175
invoke method (Method), 171–172
invokeXxx methods (ExecutorService), 334
IOException, 183, 293
  addSuppressed, getSuppressed methods, 189
isAfter, isBefore methods
  of LocalDate, 406
  of LocalTime, 409
  of ZonedDateTime, 412
isAlive method
  of Process, 369
  of ProcessHandle, 370
isCancelled method (Future), 333
isDone method
  of CompletableFuture, 336
  of Future, 333
isEmpty method
  of BitSet, 249
  of Collection, 237
  of Map, 246
isEqual method (Predicate), 121–122
isFinite, isInfinite methods (Double), 13
isInterrupted method (Thread), 364
isLoggable method (Filter), 202
isNamePresent method (Parameter), 172
isNull method (Objects), 117
ISO 8601 format, 388
ISO 8859-1 encoding, 292, 295
isPresent method (Optional), 269–271
isXxx methods (Class), 161, 174
isXxx methods (Files), 300, 302
isXxx methods (Modifier), 162, 168
isZero, isNegative methods (Duration), 404
Iterable interface, 240–241, 300, 458
  iterator method, 240
iterate method (Stream), 262, 266, 279, 349
iterator method
  of Collection, 237
  of ServiceLoader, 167
  of Stream, 271
Iterator interface
  next, hasNext methods, 240
  remove, removeIf methods, 241
iterators, 240–241, 271
  converting to streams, 281
  for random numbers, 459
  invalid, 241
  traversing, 167
  weakly consistent, 350

J

JAR files, 80–81
  dependencies in, 491
  for split packages, 483
  manifest for, 484–485
  modular, 482–484
  processing order of, 82
  resources in, 163, 435
  scanning for deprecated elements, 387
jar program, 80–81
  -C option, 482–483
  -d option, 482–483
  --module-version option, 483
Java EE platform, 335, 486–487
Java Platform Module System, 469–493
Java Persistence Architecture, 377
Java Platform Module System, 469–493
  layers in, 483
  migration to, 484–486
  no support for versioning in, 471, 474, 483
  service loading in, 490–491
java program, 4
  --add-exports, --add-opens options, 486
  --add-module option, 484, 487
  -cp (--class-path, -classpath) option, 81–82
  -disableassertions (-da) option, 195
  --enableassertions (-ea) option, 194
  --enableSystemassertions (-esa) option, 195
  --illegal-access option, 486
  -jar option, 81
  -m, -p (--module, --module-path) options, 473, 483
  option files for, 487
  specifying locales in, 426
Java programming language
  compatibility with older versions of, 145, 216
  portability of, 19
  strongly typed, 14
  Unicode support in, 30–32
  uniformity of, 3, 108
java, javax, javafx global objects
  (JavaScript), 454
java.activation module, 486–487
java.awt package, 83, 471
java.base module, 475
java.class.path, java.home, java.io.tmpdir
  system properties, 248
java.corba module, 486–487
java.desktop module, 474
Java.extend function (JavaScript), 459
Java.from function (JavaScript), 457
java.lang, java.lang.annotation packages, 386
java.lang.reflect package, 168
java.logging module, 488
java.sql package, 417
Java.super function (JavaScript), 460
java.time package, 401–417
Java.to function (JavaScript), 457
java.transaction module, 486–487
Java.type function (JavaScript), 454–455
java.util package, 7, 350
java.util.concurrent package, 350, 353
java.util.concurrent.atomic package, 355
java.util.logging package, 196
java.util.logging system property, 248
java.xml.bind, java.xml.ws
  java.xml.ws.annotation modules, 486–487
JavaBeans, 172–173
javac program, 4
  -author option, 95
  -cp (--class-path, -classpath) option, 81
  -d option, 80, 94
  -encoding option, 438
  -link, -linksource options, 95
  -parameters option, 169
  -processor option, 394
  -version option, 95
  -Xlint option, 37
  -XPrintRounds option, 398
JavaCompiler.getTask method, 444–445
javap program, 90–95
  including annotations in, 389
JavaFileObject interface, 445
JavaFX platform, 112–113
  and threads, 341
  distributed over multiple modules, 489
javax.base, javax.controls modules, 487–488
javan.log files, 200
JavaScript programming language
  accessing classes of, from Java, 451
  anonymous functions in, 458
  anonymous subclasses in, 459
  arrays in, 457–458
  arrow function syntax, 459
  bracket notation in, 454, 457–458
  calling static methods in, 455
  catching Java exceptions in, 461
  constructing Java objects in, 454–455
  delimiters in, 450
  extending Java classes in, 459–460
  implementing Java interfaces in, 459–460
  inner classes in, 455
  instance variables in, 460
  lists and maps in, 458
  methods in, 453–454
  numbers in, 456
  objects in, 456
  REPL for, 452–453
  semicolons in, 450
  strings in, 456
  superclasses in, 460
JavaServer Faces framework, 246
javax.annotation package, 386
javax.swing package, 474
JAXB (Java Architecture for XML Binding), 479
jconsole program, 200
jdeps program, 387
jdeprscan program, 491
JDK (Java Development Kit), 3
  obsolete features in, 470
jdk.incubator.http package, 306
jjs program, 452–453
  command-line arguments in, 463
  executing commands in, 462
jlink program, 492
jmod program, 493
job scheduling, 251
join method
  of String, 25
  of Thread, 363
joining method (collectors), 272
JPA (Java Persistence API), 479
JShell, 7–10
  imported packages in, 9–10
  loading modules into, 484
JSP (JavaServer Pages), 464
JUnit, 377–378
K
  K formatting symbol (date/time), 416
  \k, in regular expressions, 313
key/value pairs
  adding new keys to, 243
  in annotations. See elements
  removed by garbage collector, 251
  values of, 243
keys method (Preferences), 440
keySet method
  of ConcurrentHashMap, 354
  of Map, 246, 252
keywords, 15
L
  L suffix, 11
  [L prefix, 160
labeled statements, 40–41
lambda expressions, 113–116
  and generic types, 213
  annotating targets for, 388
  capturing variables in, 124–127
  executing, 119
  for loggers, 196
  parameters of, 115
  processing, 119–123
  return type of, 115
  scope of, 124
  this reference in, 124
throwing exceptions in, 186
using with streams, 263

vs. anonymous functions (JavaScript), 458
with parallel streams, 348
language codes, 275, 424
language model API, 395
last method (SortedSet), 243
lastIndexOf method
  of List, 238
  of String, 28
lastIndexOfSubList method (Collections), 240
lastXxx methods (TemporalAdjusters), 408
lazy operations, 261, 266, 283, 317
leap seconds, 402
leap years, 405–406
legacy code, 416–417
length method
  of arrays, 43
  of RandomAccessFile, 297
  of String, 6, 31
.level suffix, 199
lib/modules file, 493
limit method (Stream), 264, 282
line feed
  character literal for, 14
  formatting for output, 35
  in regular expressions, 314
line.separator system property, 248
lines method
  of BufferedReader, 294
  of Files, 263, 293
@link tag (javadoc), 93–94
linked lists, 237, 241
LinkedList class, 237
LinkedList class, 237
LinkedHashMap class, 246
LinkedList interface, 237
list method (files), 302–303
List interface, 215, 237
  add, addAll, get, indexOf, lastIndexOf,
    listIterator, remove, replaceAll, set,
    sort methods, 238
  of method, 46, 48, 238, 252
  subList method, 238, 252
ListIterator interface, 241
ListResourceBundle class, 437
lists
  converting to streams, 281
  in Nashorn, 458
  mutable, 253
  printing elements of, 117
  removing null values from, 117
  sublists of, 253
  unmodifiable views of, 254
little-endian format, 291
load method (ServiceLoader), 167
load balancing, 319
loadClass method (ClassLoader), 164
local classes, 129–130
local date/time, 404–410
local variables, 41–43
  annotating, 380–381
  vs. instance, 72
Locale class, 273
  and legacy classes, 417
  datesUntil method, 406–407
  getXxx methods, 61, 406–407
  isXxx methods, 406
  minus, minusXxx methods, 406–407
  now method, 70, 77, 405–406
  of method, 61, 70, 405–406
  ofInstant method, 406
  parse method, 430
  plus, plusXxx methods, 61–62, 64,
    406–407
  toEpochSecond method, 406
  until method, 406
  withXxx methods, 406
LocaleDateTime class, 410
  and legacy classes, 417
  atZone method, 410
  parse method, 430
Locale class, 273
  forLanguageTag method, 426
  getAvailableLocales method, 425
  getCountry method, 274
  getCurrency method, 426, 436
  getDisplayName method, 426
  getISOXXX methods, 425
  setDefault method, 425
  predefined fields, 425
locales, 273–276, 422–427
date/time formatting for, 429–431
default, 413, 426, 429–430, 436
displaying names of, 426
first day of week in, 431
for template strings, 433–435
formatting styles for, 415, 430
sorting words for, 431–432
specifying, 423–425
weekdays and months in, 415–416

LocalTime class, 409–410
and legacy classes, 417
final, 141
getXxx, isXxx, minus, minusXxx, now, of,
ofInstant, plus, plusXxx, toXxx, withXxx
methods, 409
parse method, 430
lock method
of FileChannel, 298
of ReentrantLock, 358
locks, 346
error-prone, 347
intrinsic, 358–360
reentrant, 357–358
releasing, 189, 343

log handlers, 200–202
default, 197, 200
filtering/formatting, 202
installing custom, 200
levels of, 200
suppressing messages in, 197

Logger class, 488
addHandler method, 200
entering, exiting methods, 197
fine method, 197
getGlobal method, 195
getLogger method, 196
info method, 195
log method, 197–198
logp, logrb methods, 198
setFilter method, 203
setLevel method, 195, 197, 200
setUseParentHandlers method, 200
throwing method, 198
warning method, 197

loggers
defining, 196
filtering/formatting, 202
hierarchy of, 196

logging, 195–203
configuring, 197–200
enabling/disabling, 197
failures, 190
levels of, 197–200
localizing, 199
overriding methods for, 141
using for unexpected exceptions, 198

Long class, 46
MIN_VALUE, MAX_VALUE constants, 11
unsigned division in, 12
xxxUnsigned methods, 20

long indicator, in string templates, 433

long type, 10–12
atomic operations on, 356–357
functional interfaces for, 122

streams of, 279
type conversions of, 20–22
LongAccumulator class, 356
accumulate, get methods, 356
LongAdder class, 356–357
add, increment, sum methods, 356
LongConsumer, LongXXXOperator, LongPredicate,
LongSupplier, LongToXXXFunction
interfaces, 122
LongFunction interface, 122, 220
longs method (Random), 280
LongStream class, 279–280
LongSummaryStatistics class, 272, 280
long-term persistence, 324

Lookup class, 482
lookup method (MethodHandles), 482
loops, 38–39
exiting, 39–41
infinite, 39

M
m, M formatting symbols (date/time), 416
main method, 2, 6
decomposing, 52–54
string array parameter of, 49
ManagedExecutorService class, 335
Map interface, 238
  clear method, 246
  compute, computeIfXxx methods, 245
  containsXxx methods, 246
  entrySet method, 246
  forEach method, 246
  get, getOrDefault methods, 243, 245
  isEmpty method, 246
  keySet method, 246, 252
  merge method, 244–245
  merge method of, 244–245
  put method, 243, 245
  putAll, putIfAbsent methods, 245
  remove, replace methods, 245
  replaceAll method, 246
  size method, 246
  values method, 246, 252
map method
  of Optional, 268
  of Stream, 263
mapping method (Collectors), 276
maps, 243–246
  concurrent, 246, 274
  empty, 246
  in Nashorn, 458
  iterating over, 244
  of stream elements, 273–274, 282
  order of elements in, 246
  views of, 244
    unmodifiable, 254
mapToInt method (Stream), 278
mapToXxx methods (XxxStream), 280
marker interfaces, 153
Matcher class, 315–317
  quoteReplacement method, 318
  replaceAll method, 317–318
  replaceFirst method, 318
matcher, matches methods (Pattern), 314
MatchResult interface, 315–318
Math class
  E constant, 20
  floorMod method, 19
  max, min methods, 19
  PI constant, 20, 75
  pow method, 19, 77
round method, 21
sqrt method, 19
xxxExact methods, 20, 22
max method
  of Stream, 266
  of XxxStream, 280
MAX_VALUE constant (integer classes), 11
maxBy method
  of BinaryOperator, 121
  of Collectors, 276
medium indicator, in string templates, 433
memory
  allocating, 346
  caching, 342
  concurrent access to, 343
memory-mapped files, 297
merge method
  of ConcurrentHashMap, 351
  of Map, 244–245
Message class, 153–154
MessageFormat class, 433–435
meta-annotations, 384–390
META-INF/MANIFEST.MF file, 484–485
META-INF/services directory, 490
Method interface, 168–169
  getModifiers, getName methods, 168
  invoke method, 171–172
method calls, 6
  receiver of, 67
method expressions, 117, 145
method references, 117–118, 221
  annotating, 382
MethodHandles.lookup method, 482
methods, 2
  abstract, 115, 141–142
  accessor, 62
  annotating, 224, 380
  atomic, 351
  body of, 66
  chaining calls of, 62
  clashes of, 224–225
  compatible, 151
  declarations of, 65
  default, 106–108
  deprecated, 93
documentation comments for, 90, 92
methods (cont.)

- enumerating, 168–169
- factory, 70, 78
- final, 141, 347
- header of, 65
- inlining, 141
- instance, 66–67
- invoking, 171
- modifying functions, 128
- mutator, 62, 254, 347
- naming, 14–15
- native, 76
- overloading, 71, 118
- overriding, 106, 137–139, 141, 185–186, 387
- parameters of, 169
  - null checks for, 193
- passing arrays into, 53
- private, 109
- proxied, 176
- public, 101–102, 168
- restricted to subclasses, 142–143
- return value of, 2, 66
- returning functions, 127
- static, 53, 77–78, 85, 105–106
- storing in variables, 6
- symmetric, 150
- synchronized, 359–361
- utility, 83
- variable number of arguments of, 53

Microsoft Notepad, 291

Microsoft Windows
- batch files, 461
- path separator, 81, 248
- registry, 439

min method
- of Math, 19
- of Stream, 266
- of XxxStream, 280

MIN_VALUE constant (integer classes), 11

minBy method
- of BinaryOperator, 121
- of Collectors, 276

minus, minusXXX methods
- of Instant, Duration, 404
- of LocalDate, 406–407

of LocalDateTime, 409
- of ZonedDateTime, 411

Modifier interface
- isXXX methods, 162, 168
- toString method, 162

modifiers, checking, 168

module keyword, 473

module path, 473, 483–485

Module.getResourceAsStream method, 481

module-info.class file, 473, 482

module-info.java file, 473

modules, 469–493
- aggregator, 488
- and versioning, 471, 474, 483
- annotating, 474
- automatic, 484–485
- bundling up the minimal set of, 492
- declaration of, 472–473
- deprecated, 487
- documentation comments for, 91, 94
- illegal access to, 486
- inspecting files in, 493
- loading into JShell, 484
- naming, 472, 484
- open, 481
- reflective access for, 169–170
- required, 474–476, 487–489
- tools for, 491–493
- transitive, 487–489
- unnamed, 485

monads, 264

monitors (classes), 360

Month enumeration, 405–406, 412
- getDisplayName method, 415, 430

MonthDay class, 407

move method (Files), 301–302

multiplication, 18

multipliedBy method (Duration), 404

mutators, 62
  - and unmodifiable views, 254

N

- conversion character, 35
- formatting symbol (date/time), 416
for character literals, 14
in property files, 247–248
in regular expressions, 311–313, 318
name method (Enum), 156
NaN (not a number), 13
Nashorn engine, 448, 452–461
  anonymous subclasses in, 459
  arrays in, 457–458
  catching Java exceptions in, 461
  class objects in, 455
  extending Java classes in, 459–460
  getters/setters in, 454
  implementing Java interfaces in, 459–460
  instance variables in, 460
  lists and maps in, 458
  methods in, 453–454
  no standard input source in, 450
  numbers in, 456
  running from command line, 452
  shell scripting in, 461–464
  strings in, 455
  superclasses in, 460
native methods, 76
naturalOrder method (Comparator), 129
NavigableMap interface, 354
NavigableSet interface, 238, 242, 253
  methods of, 243
nCopies method (Collections), 237, 239
negate method (Predicate, BiPredicate), 121
negated method (Duration), 404
negateExact method (Math), 20
NEGATIVE_INFINITY value (Double), 13
negative values, 10
nested classes, 85–90
  annotating, 382
  enumerating, 168–169
  inner, 87–89
  public, 86
  static, 85–86
new operator, 6, 14, 17, 70
  as constructor reference, 118
  for anonymous classes, 130
  for arrays, 43–44, 50
  in JavaScript, 455–459
newBufferedReader method (Files), 294, 448
newBufferedWriter method (Files), 294, 302
newBuilder, newHttpClient methods
  (HttpClient), 308, 335
newFileSystem method (FileSystems), 305
newInputStream method (Files), 288, 302, 320
newInstance method
  of Array, 174
  of Class, 171, 227
  of Constructor, 171–172
newKeySet method (ConcurrentHashMap), 354
newline. See line feed
newOutputStream method (Files), 288, 302, 320
newProxyInstance method (Proxy), 175
newXxxThreadPool methods (Executors), 331
next method (Iterator), 240
next, nextOrSame methods
  (TemporalAdjusters), 408
next, nextXxx methods (Scanner), 32, 293
nextInt method (Random), 6, 38
nextXxxBit methods (BitSet), 249
nominal typing, 120
noneMatch method (Stream), 267
noneOf method (EnumSet), 250
noninterference, of stream operations, 283
@NonNull annotation, 381
normalize method (Path), 299
Normalizer class, 433
NoSuchElementException, 269, 353
notify, notifyAll methods (Object), 146, 361–362
NotSerializableException, 321
now method
  of Instant, 403
  of LocalDate, 70, 77, 405–406
  of LocalDateTime, 409
  of ZonedDateTime, 411
null value, 26, 64
  as default value, 71, 74
  checking parameters for, 193
  comparing against, 149
  converting to strings, 147
NullIOException, 26, 45, 64, 72, 184, 193, 244
vs. Optional, 266
nullsFirst, nullsLast methods (Comparator), 129
Number class, 428
number indicator, in string templates, 433
Number type (JavaScript), 456
NumberFormat class
getXXXInstance methods, 78, 427
not threadsafe, 365–366
parse method, 428
setCurrency method, 429
NumberFormatException, 184
numbers
big, 23–24
comparing, 110
converting to strings, 27
default value of, 71, 74
even or odd, 18
formatting, 34, 422, 427, 433
from grouped elements, 275
in regular expressions, 312
non-negative, 194, 248
printing, 34
random, 6, 38, 262, 264, 280
reading/writing, 293, 296–297
rounding, 13, 21
type conversions of, 20–22
unsigned, 12, 20
with fractional parts, 12–13

O
0 conversion character, 34
Object class, 145–154
call method, 143, 146, 151–154, 171
equals method, 146, 148–150
finalize method, 146
getClass method, 141, 146, 149, 159, 221, 227
hashCode method, 146, 148, 150–151
notify, notifyAll methods, 146, 361–362
toString method, 146–147
wait method, 146, 361–362
object references, 63–64
and serialization, 320
attempting to change, 69
comparing, 148
default value of, 71, 74
null, 64
passed by value, 69
ObjectInputStream class, 320–321
defaultReadObject method, 322
readFields method, 325
readObject method, 320–323, 325
object-oriented programming, 59–97
encapsulation, 469–470
ObjectOutputStream class, 320
defaultWriteObject method, 322
writeObject method, 320–322
object-relational mappers, 479
objects, 2, 60–64
calling methods on, 6
casting, 103–104
cloning, 151–154
comparing, 47, 148–150
constructing, 6, 69–74, 171–172
in JavaScript, 454–455
converting:
to strings, 146–147
to XML, 480
deep/shallow copies of, 152–154
deserialized, 322–324
externalizable, 322
immutable, 62
initializing variables with, 14
inspecting, 169–171
invoking static methods on, 77
mutable, 73
serializable, 319–321
sorting, 109–111
state of, 60
Objects class
equals method, 149
hash method, 151
isNull method, 117
requireNonNull method, 193
ObjXXXConsumer interfaces, 122
octal numbers, 11
formatting for output, 34
octonions, 31
odd numbers, 18
of method
  of EnumSet, 250  
  of IntStream, 279  
  of List, 46, 48, 238, 252  
  of LocalDate, 61, 70, 405–406  
  of LocalTime, 409  
  of Map, 246, 252  
  of Optional, 269  
  of ProcessHandle, 370  
  of Set, 252  
  of Stream, 261–262  
  of ZonedDateTime, 410–411  
  of DateAdjuster method (TemporalAdjusters), 408  
  of Days method (Period), 413  
  of Entries method (Map), 252  
  offer method (BlockingQueue), 353  
  offsetByCodePoints method (String), 31  
  OffsetDateTime class, 413  
  open keyword, 481  
  open method (FileChannel), 297  
  openConnection method (URL), 306  
  opens keyword, 481  
  qualified, 489  
  openStream method (URL), 288  
  Operation interface, 157  
  operations  
    associative, 278  
    atomic, 346, 351, 354–357, 360  
    bulk, 352  
    lazy, 261, 266, 283, 317  
    parallel, 348–350  
    performed optimistically, 355  
    stateless, 281  
    threadsafe, 350–354  
    operators, 17–24  
    precedence of, 17  
  option files, 487  
  Optional class, 266–270  
    creating values of, 269  
    empty method, 269  
    flatMap method, 269–271  
    for empty streams, 277–278  
    for processes, 370  
    get method, 269–271  
    ifPresent, ifPresentOrElse methods, 268  
    isPresent method, 269–271  
    map method, 268  
    of, ofNullable methods, 269  
    orElse method, 266–268  
    orElseXxx methods, 267–268  
    stream method, 270–271  
  OptionalXxx classes, 280  
  or method  
    of BitSet, 249  
    of Predicate, BiPredicate, 121  
  order method (ByteBuffer), 297  
  ordinal method (Enum), 156  
  org global object (JavaScript), 454  
  org.omg.corba package, 470  
  os.arch, os.name, os.version system properties, 248  
  OSGi (Open Service Gateway Initiative), 471  
  $OUT, in shell scripts, 462  
  output  
    formatted, 33–36  
    redirecting, 449  
    setting locales for, 427  
    writing, 294–295  
  output streams, 288  
    closing, 290  
    obtaining, 288  
    writing to, 290  
  OutputStream class, 320  
    write method, 290  
  OutputStreamWriter class, 294  
  @Override annotation, 138, 386–387
overriding, 137–139
  for logging/debugging, 141
overview.html file, 94

P
\p, \P, in regular expressions, 312
package statement, 79
package declarations, 79–80
Package object (JavaScript), 454
package-info.java file, 94, 380
packages, 3, 78–85
  accessing, 83, 143, 470, 477–478, 481, 484
  adding classes to, 83
  annotating, 380–381
default, 79
documentation comments for, 91, 94
exporting, 476–479, 481
naming, 79
not nesting, 79
split, 483
parallel method (XxxStream), 280
parallel streams, 348–349
parallelStream method (Collection), 237, 260–261, 280, 348
parallelXxx methods (Arrays), 49, 349
@param tag (javadoc), 92
Parameter class, 172
parameter variables, 68
  annotating, 380
  scope of, 42
ParameterizedType interface, 228
parentLogger method (Driver), 488
parse method
  of DateTimeFormatter, 415
  of LocalDate, ZonedDateTime, 430
  of NumberFormat, 428
Parse.quote method, 311
parseDouble method (Double), 27
ParseException, 428
parseInt method (Integer), 27, 184
partitioning, 347
partitioningBy method (Collectors), 275, 277
Pascal triangle, 51
passwords, 33
Path interface, 106, 298–300
  get method, 298–300
  getXXX methods, 300
  normalize, relativize methods, 299
  resolve, resolveSibling methods, 299
  subpath method, 300
toAbsolutePath, toFile methods, 299
path separators, 298
path.separator system property, 248
paths, 298
  absolute vs. relative, 298–299
  filtering, 303
  resolving, 299
  taking apart/combining, 300
Paths class, 106
Pattern class
  compile method, 315, 318
  flags, 318–319
  matcher, matches methods, 314
  split method, 317
  splitAsStream method, 263, 317
pattern variables, 202
PECS (producer extends, consumer super), 214
peek method
  of BlockingQueue, 353
  of Stream, 266
percent indicator, in string templates, 433
performance
  and atomic operations, 355
  and combined operators, 19
  and memory caching, 343
Period class, 405
  ofDays method, 413
@Persistent annotation, 389
PI constant (Math), 20, 75
placeholders, 433–435
platform class loader, 163
platform encoding, 292, 438
plugins, loading, 164
plus, plusXXX methods
  of Instant, Duration, 404
  of LocalDate, 61–62, 64, 406–407
  of LocalTime, 409
  of ZonedDateTime, 411–412
Point class, 146–147
Point2D class (JavaFX), 321
poll method (BlockingQueue), 353
pollXXX methods (NavigableSet), 243
pop method (ArrayDeque), 250
portability, 19
POSITIVE_INFINITY value (Double), 13
POST requests, 308
@PostConstruct annotation, 386, 388
pow method (Math), 19, 77
predefined character classes, 310, 312, 314
@PreDestroy annotation, 386, 388
predicate functions, 275
Predicate interface, 116, 121
and, or, negate methods, 121
isEqual method, 121–122
test method, 121, 213
Preferences class, 439–441
previous method (ListIterator), 241
previous, previousOrSame methods
(TemporalAdjusters), 408
previousXxxBit methods (BitSet), 249
preVisitDirectory, postVisitDirectory
methods (FileVisitor), 304
primitive types, 10–14
and type parameters, 220
attempting to update parameters of, 68
comparing, 149
converting to strings, 147
functions interfaces for, 122
passed by value, 69
streams of, 278–280
wrapper classes for, 46–47
printStackTrace method (Throwable), 192
PrintStream class, 6, 147, 295
print method, 6, 33, 195, 294–295
println method, 6, 32–33, 49, 117, 294–295
PrintWriter class, 294
close method, 187–188
printf method, 427
priority queues, 251
private modifier, 2, 83
for enum constructors, 157
Process class, 366–370
destroy, destroyForcibly methods, 369
exitValue method, 369
getErrorStream method, 368
getXXXStream methods, 367
isAlive method, 369
onExit method, 369
supportsNormalTermination method, 369
toHandle method, 370
waitFor method, 369
ProcessBuilder class, 366–370
directory method, 367
redirectXXX methods, 368
start method, 368
processes, 366–370
building, 367–368
getting info about, 370
killing, 369
running, 368–369
ProcessHandle interface, 370
processing pipeline, 337
Processor interface, 394
Programmer’s Day, 405
programming languages
dynamically typed, 456
functional, 99
object-oriented, 2
scripting, 448
programs
compiling, 3
configuration options for, 247
localizing, 421–441
packaging, 493
responsive, 340
running, 3
testing, 193
promises (in concurrent libraries), 336
properties, 172–173
loading from file, 247
naming, 173
read-only/write-only, 173
testing for, 213
Properties class, 247–248
.properties extension, 435
property files
   encoding, 247, 437
   generating, 398
   localizing, 435–437
protected modifier, 142–143
Provider.get, Provider.type methods, 167
provides keyword, 490
Proxy class, 175–176
   newProxyInstance method, 175
public modifier, 2, 83
   and method overriding, 139
   for interface methods, 101–102
push method (ArrayDeque), 250
put method
   of BlockingQueue, 353
   of FileChannel, 297
   of Map, 243, 245
   of Preferences, 440
putAll method (Map), 245
putAbsent method
   of ConcurrentHashMap, 351
   of Map, 245
putXxx methods (FileChannel), 297
putXxx methods (Preferences), 440

Q
\Q, in regular expressions, 311
qualified exports, 489
Queue interface, 238, 250
   synchronizing methods in, 360
   using ArrayDeque with, 250
quote method (Parse), 311
quoteReplacement method (Matcher), 318

R
\r (carriage return)
   for character literals, 14
   in property files, 248
\r, \R, in regular expressions, 311, 314
race conditions, 281, 344–346
Random class, 6
   ints, longs, doubles methods, 280
   nextInt method, 6, 38
random numbers, 6, 38, 459
   streams of, 262, 264, 280
RandomAccess interface, 237
RandomAccessFile class, 296–297
   getFilePointer method, 297
   length method, 297
   seek method, 296–297
RandomNumbers class, 77
range method (EnumSet), 250
range, rangeClosed methods (XxxStream), 279
ranges, 253
   converting to streams, 281
raw types, 217, 220–221
read method
   of Files, 289
   of InputStream, 289
   of InputStreamReader, 293
readAllXxx methods (Files), 289, 293
Reader class, 293
readers, 288
readExternal method (Externalizable), 322
readFields method (ObjectInputStream), 325
readLine function (shell scripts), 464
readLine method
   of BufferedReader, 294
   of Console, 33
readNBytes method (Files), 289
readObject method (ObjectInputStream), 320–323, 325
readPassword method (Console), 33
readResolve method (Serializable), 322–323
readXxx methods (DataInput), 296–297, 322
receiver parameters, 67, 383
redirection syntax, 33
redirectXxx methods (ProcessBuilder), 368
reduce method (Stream), 277–279
reduceXxx methods (ConcurrentHashMap), 352
reducing method (Collectors), 277
reductions, 266, 277–279
ReentrantLock class, 357–358
   lock, unlock methods, 358
reflection, 168–176
   and generic types, 222, 226–229
   and module system, 169–170, 479, 486
   processing annotations with, 391–393
ReflectiveOperationException, 160
regular expressions, 310–319
  finding matches of, 314–316
  flags for, 318–319
  groups in, 316–317
  replacing matches with, 317
  splitting input with, 317
relational operators, 22
relativize method (Path), 299
remainderUnsigned method (Integer, Long), 20
remove method
  of ArrayDeque, 250
  of ArrayList, 46
  of BlockingQueue, 353
  of Collection, 236
  of Iterator, 241
  of List, 238
  of Map, 245
  of Preferences, 440
removeIf method
  of ArrayList, 116
  of Iterator, 241
removeNode method (Preferences), 440
removeXXX methods (Collection), 236
@Repeatable annotation, 387, 389
REPL ("read-eval-print" loop), 452–453
replace method
  of Map, 245
  of String, 28
replaceAll method
  of Collections, 239
  of List, 238
  of Map, 246
  of Matcher, 317–318
  of String, 317
replaceFirst method (Matcher), 318
requireNonNull method (Objects), 193
requires keyword, 473, 476–479, 484, 487–489
resolve, resolveSibling methods (Path), 299
@Resource annotation, 386, 388
resource bundles, 435–438
resource injections, 388
ResourceBundle class, 199
  extending, 437
  getBundle method, 436–438
  getObject method, 437
  getString method, 436
resources, 159–168
  loading, 162, 481
  managing, 187
@Retention annotation, 387
resume method (Thread, deprecated), 363
retainAll method (Collection), 236
@Resource annotation, 386, 388
repl program, 453
reverse method (Collections), 49, 240
reverse domain name convention, 79, 472
reversed method (Comparator), 128
reverseOrder method (Comparator), 129
RFC 822, RFC 1123 formats, 414
rlwrap program, 453
rotate method (Collections), 240
round method (Math), 21
RoundEnvironment interface, 395
roundoff errors, 13
RowSetProvider class, 486
rt.jar file, 493
runAfterXXX methods (CompletableFuture), 339–340
Runnable interface, 112, 121, 331, 333
  executing on the UI thread, 341
  run method, 121, 330, 363–364
  using class literals with, 160
runtime
  raw types at, 220–221
  safety checks at, 218
Runtime class
  availableProcessors method, 331
  exec method, 367
runtime image file, 493
RuntimeException, 183
S
s formatting symbol (date/time), 416
s, 5 conversion characters, 34
\s, \S, in regular expressions, 312
safety checks, as runtime, 218
@SafeVarargs annotation, 224, 386, 388
Scala programming language
  REPL in, 453
  type parameters in, 215
Scanner class, 32
  findAll method, 316
  hasNext, hasNextXXX, next, nextXXX methods, 32, 293
  tokens method, 263, 294
  useLocale method, 427
scheduling applications
  and time zones, 405, 410
  computing dates for, 407–408
ScriptContext interface, 449
ScriptEngine interface
  createBindings method, 449
  eval method, 449–451
ScriptEngineFactory interface, 451
ScriptEngineManager class
  getEngineXXX methods, 448
  visibility of bindings in, 449
scripting engines, 448–449
  compiling code in, 452
  implementing Java interfaces in, 451
scripting languages, 448
  invoking functions in, 450
searchXXX methods (ConcurrentHashMap), 352
security, 83
SecurityException, 170
@see tag (javadoc), 93–94
seek method (RandomAccessFile), 296
sequences, producing, 262
serial numbers, 321
Serializable interface, 319–321
  readResolve, writeReplace methods, 322–323
serialization, 319–325
serialVersionUID instance variable, 324
server-side software, 319
ServiceLoader class, 166–168, 490
  iterator, load method, 167
ServiceLoader.Provider interface, 167
services
  configurable, 166
  loading, 166–168, 490–491
ServletException class, 191
Set interface, 238, 354
  of method, 252
  working with EnumSet, 250
set method
  of Array, 174
  of ArrayList, 46
  of BitSet, 249
  of Field, 172
  of List, 238
  of ListIterator, 241
setAccessible method (AccessibleObject), 170, 172
setContextClassLoader method (Thread), 165
setCurrency method (NumberFormat), 429
setDaemon method (Thread), 366
setDecomposition method (Collator), 432
setDefault method (Locale), 426
setDefaultUncaughtExceptionHandler method (Thread), 192
setDoOutput method (URLConnection), 306
setFilter methods (Handler, Logger), 203
setFormatter method (Handler), 203
setLevel method (Logger), 195, 197, 200
setOut method (System), 76
setReader method (ScriptContext), 449
setRequestProperty method (URLConnection), 306
sets, 242–243
  immutable, 347
  threadsafe, 354
  unmodifiable views of, 254
setIdStrength method (Collator), 432
setUncaughtExceptionHandler method (Thread), 363
setUseParentHandlers method (Logger), 200
setWriter method (ScriptContext), 449
setXXX methods (Array), 174
setXXX methods (Field), 170, 172
setXXXAssertionStatus methods
  (ClassLoader), 195
shallow copies, 152–154
shared variables, 343–347
  atomic mutations of, 354–357
  locking, 357–358
shebang, 464
shell scripts, 461–464
  command-line arguments in, 463
  environment variables in, 463
  executing, 462
  generating, 398
  string interpolation in, 462–463
shell, redirection syntax of, 33
shift operators, 23
Shift_JIS encoding, 292
short circuit evaluation, 22
short indicator, in string templates, 433
short type, 10–12
  streams of, 279
  type conversions of, 21
short-term persistence, 324
shuffle method (Collections), 49, 240
SimpleFileVisitor class, 304
SimpleJavaFileObject class, 446
@since tag (javadoc), 92
singletons, 323
size method
  of ArrayList, 46
  of Collection, 237
  of Map, 246
skip method (Stream), 264
sleep method (Thread), 363, 365
SLF4J (Simple Logging Facade for Java), 196, 472
SOAP protocol, 471
SocketHandler class, 200
sort method
  of Arrays, 49, 111–112, 116–117
  of Collections, 49, 215, 240
  of List, 238
sorted maps, 253–254
sorted method (Stream), 265
sorted sets, 238, 253
  traversing, 242
  unmodifiable views of, 254
sorted streams, 281
SortedMap interface, 253
SortedSet interface, 238, 242
  first, last methods, 243
  headSet, subSet, tailSet methods, 243, 253
sorting
  array lists, 49
  arrays, 49, 109–111
  chaining comparators for, 128
  changing order of, 127
  streams, 265
  strings, 26–27, 117, 431–432
source code, generating, 395–398
source files
  documentation comments for, 94
  encoding of, 438
  placing, in a file system, 80
  reading from memory, 445
  space flag (for output), 35
  spaces
    in regular expressions, 312
    removing, 28
split method
  of Pattern, 317
  of String, 25, 317
splitAsStream method (Pattern), 263, 317
spliterator method (Collection), 237
sqrt method (Math), 19
square root, computing, 270
Stack class, 250
stack trace, 192–193
StackWalker class, 192
standard output, 3
StandardCharsets class, 292
StandardJavaFileManager interface, 445–447
start method
  of Matcher, MatchResult, 315–316
  of ProcessBuilder, 368
  of Thread, 363
startsWith method (String), 28
stateless operations, 281
statements, combining, 43
static constants, 75–76
static imports, 85
static initialization, 164
static methods, 53, 77–78
  accessing static variables from, 77
  importing, 85
  in interfaces, 105–106
static modifier, 2, 16, 53, 74–78, 158
  for modules, 488
static nested classes, 85–86
static variables, 74–75
  accessing from static methods, 77
  importing, 85
  visibility of, 343
stop, suspend methods (Thread, deprecated), 363
Stream interface
  collect method, 271–272, 279
  concat method, 265
  count method, 261, 266
  distinct method, 265, 282
  dropWhile method, 265
  empty method, 262
  filter method, 261–263, 267
  findAny method, 267
  findFirst method, 168, 267
  flatMap method, 264
  forEach, forEachOrdered methods, 271
  generate method, 262, 279
  iterate method, 262, 266, 269, 349
  iterator method, 271
  limit method, 264, 282
  map method, 263
  mapToInt method, 278
  max, min methods, 266
  of method, 261–262
  ofNullable method, 271
  peek method, 266
  reduce method, 277–279
  skip method, 264
  sorted method, 265
  takeWhile method, 265
  toArray method, 118, 271
  unordered method, 282
  xxxMatch methods, 267
stream method
  of Arrays, 262, 279
  of BitSet, 249
  of Collection, 237, 260–261
  of Optional, 270–271
streams, 259–283
  collecting elements of, 271–274
  computing values from, 277–279
  converting to/from arrays, 262, 271, 281, 350
  creating, 261–263
  debugging, 266
  empty, 262, 266, 277–278
  filtering, 270
  finite, 262
  flattening, 264, 270
  infinite, 261–262, 264, 266
  intermediate operations for, 261
  locating services with, 167
  noninterference of, 283
  of primitive type values, 278–280
  of random numbers, 280
  ordered, 281
  processed lazily, 261, 266, 283
  reductions of, 266
  removing duplicates from, 265
  sorting, 265
  splitting/combing, 264–265
  summarizing, 272, 280
  terminal operation for, 261, 266
  transformations of, 263–264, 280
  vs. collections, 261
strictfp modifier, 19
StrictMath class, 20
String class, 6, 28
  charAt method, 31
  codePoints, codePointXxx methods, 31–32
  compareTo method, 26–27, 109, 431
  compareToIgnoreCase method, 117
  contains method, 28
  endsWith method, 28
  equals method, 25–26
  equalsIgnoreCase method, 26
  final, 141
  format method, 427
  hash codes, 150
  immutability of, 28, 347
  indexOf, lastIndexOf methods, 28
join method, 25
length method, 6, 31
offsetByCodePoints method, 31
replace method, 28
replaceAll method, 317
split method, 25, 317
startsWith method, 28
substring method, 25
toLowerCase method, 28, 263, 427
toUpperCase method, 28, 427
trim method, 28, 428
string interpolation, in shell scripts, 462–463
StringBuilder class, 25
strings, 6, 24–32
  comparing, 25–27
  concatenating, 24–25, 147
  converting:
    from byte arrays, 292
    from objects, 146–147
    to code points, 263
    to numbers, 27
empty, 26–27, 147
formatting for output, 34
internal representation of, 32
normalized, 432
sorting, 26–27, 117, 431–432
splitting, 25, 263
templates for, 433–435
transforming to lower/uppercase, 263, 427
StringSource class, 445
StringWriter class, 295
strong element (HTML), in documentation comments, 91
subclasses, 136–137
  anonymous, 143–144, 157
  calling toString method in, 147
  constructors for, 139
  initializing instance variables in, 139
  methods in, 137
  preventing, 141
  public, 139
  superclass assignments in, 139
subList method (List), 238, 253
subMap method (SortedMap), 253
subpath method (Path), 300
subSet method
  of NavigableSet, 243
  of SortedSet, 243, 253
substring method (String), 25
subtractExact method (Math), 20
subtraction, 18
  accurate, 24
  not associative, 278
subtypes, 103
  wildcards for, 212
sum method
  of LongAdder, 356
  of XxxStream, 280
summarizingXxx methods (Collectors), 272, 276
summaryStatistics method (XxxStream), 280
summingXxx methods (Collectors), 276
superclasses, 136–137
  annotating, 381
  calling equals method, 149
  default methods of, 144–145
  in JavaScript, 460
  methods of, 137–139
  public, 139
supertypes, 103–105
  wildcards for, 213–214
Supplier interface, 121, 336
supplyAsync method (CompletableFuture), 335–337
supportsNormalTermination method
  of Process, 369
  of ProcessHandle, 370
@SuppressWarnings annotation, 37, 220, 386, 388–389, 474
swap method (Collections), 240
Swing GUI toolkit, 113, 341
SwingConstants interface, 105
SwingWorker class (Swing), 341
switch statement, 37
  using enumerations in, 158
symbolic links, 302–303
synchronized keyword, 358–361
synchronized views, 254
synchronizedXxx methods (Collections), 240
System class
   getProperties method, 248
   getProperty method, 163
   setOut method, 76
system class loader, 163, 165
system classes, enabling/disabling
   assertions for, 195
system properties, 248
System.err constant, 192, 200, 366, 444
System.in constant, 32
System.out constant, 6, 16, 32–35, 49, 53,
   76, 117, 195, 294, 444
systemXxx methods (Preferences), 439
T
T, in dates, 414
t, T conversion characters, 35
\t
   in regular expressions, 311
   tab, for character literals, 14
%t pattern variable, 202
tab completion, 9
tagging interfaces, 153
tailMap method (SortedMap), 253
tailSet method
   of NavigableSet, 243
   of SortedSet, 243, 253
take method (BlockingQueue), 353
takeWhile method (Stream), 265
tar program, 80–81
@Target annotation, 384–385, 387
Task class (JavaFX), 341
tasks, 330–335
   cancelling, 333–334
   combining results from, 333–335
   computationally intensive, 331
   coordinating work between, 352–353
   defining, 112
   executing, 112, 331
   groups of, 366
   long-running, 340–341
   running, 330–332
   short-lived, 331
   submitting, 333
   vs. threads, 331
   working simultaneously, 336
Temporal interface, 408
TemporalAdjuster.ofDateAdjuster method, 408
TemporalAdjusters class, 408
terminal window, 4
test method
   of BiPredicate, 121
   of Predicate, 121, 213
   of XxxPredicate, 122
@Test annotation, 378–379, 384
TextStyle enumeration, 431
thenAccept method (CompletableFuture), 335, 339
thenAcceptBoth, thenCombine methods
   (CompletableFuture), 339–340
thenApply, thenApplyAsync methods
   (CompletableFuture), 337–339
thenComparing method (Comparator), 128–129
thenCompose method (CompletableFuture), 338–339
thenRun method (CompletableFuture), 339
third-party libraries, 484–485
this reference, 67–68
   annotating, 383
   capturing, 118
   in constructors, 71, 348
   in lambda expressions, 124
Thread class
   get/setContextClassLoader methods, 165
   interrupted, isInterrupted methods, 364
   join method, 363
   properties, 366
   resume, stop, suspend methods
      (deprecated), 363
   setDaemon method, 366
   setDefaultUncaughtExceptionHandler
      method, 192
   setUncaughtExceptionHandler method, 363
   sleep method, 363, 365
   start method, 363
ThreadLocal class, 365–366
   get, withInitial methods, 365
   threads, 331, 362–366
   and visibility, 342–344, 360
   atomic mutations in, 354–357
creating, 112
daemon, 366
groups of, 366
interrupting, 333, 364–365
local variables in, 365–366
locking, 357–358
names of, 366
priorities of, 366
race conditions in, 281, 344–346
running tasks in, 112
starting, 363
states of, 366
temporarily inactive, 364
terminating, 331–332
uncaught exception handlers of, 366
vs. tasks, 331
waiting on conditions, 360
worker, 340–341
throw statement, 183
Throwable class, 183
getStackTrace, printStackTrace methods, 192
in assertions, 194
initCause method, 191
no generic subtypes for, 225
throwing method (Logger), 198
throws keyword, 185

type variables in, 225–226
@throws tag (javadoc), 92, 186
time

current, 402
formatting, 413–416, 429–431
measuring, 403
parsing, 415
Time class, 416–417
time indicator, in string templates, 433
time zones, 410–413
TimeException, 333
Timestamp class, 150, 416–417
timestamps, 413
using instants as, 403
TimeZone class, 417
™ (trademark symbol), 432–433
toAbsolutePath method (Path), 299
toArray method

of Collection, 237
of Stream, 118, 271
of XxxStream, 280
toByteArray method

of BitSet, 249
of ByteArrayOutputStream, 288–289
toCollection method (Collectors), 272
toConcurrentMap method (Collectors), 274
toEpochSecond method

of LocalDate, 406
of LocalTime, 409
toFile method (Path), 300
toFormat method (DateTimeFormatter), 415
toGenericString method (Class), 161
toHandle method (Process), 370
toInstant method

of Date, 416
of ZonedDateTime, 410, 412
toIntExact method (Math), 22
tokens method (Scanner), 263, 294
toList method (Collectors), 272
toLocalXxx methods (ZonedDateTime), 412
toLongArray method (BitSet), 249
toLowerCase method (String), 28, 263, 427
toMap method (Collectors), 273–274
ToolProvider.getSystemJavaCompiler method, 444
tools.jar file, 493
toPath method (File), 300
toSet method (Collectors), 272, 275
toString method
calling from subclasses, 147
of Arrays, 49, 147
of BitSet, 249
of Class, 161
of Double, Integer, 27
of Enum, 156
of Modifier, 162
of Object, Point, 146–147
toUnsignedInt method (Byte), 12
toUpperCase method (String), 28, 427
toXxx methods (Duration), 403
ToXxxFunction interfaces, 122, 220
toXxxOfDay methods (LocalTime), 409
toZonedDateTime method (GregorianCalendar), 416–417
transferTo method (InputStream), 290
transient modifier, 321
transitive keyword, 487–489
TreeMap class, 243, 274
TreeSet class, 242
trim method (String), 28, 428
tru
true value (boolean), 14
try statement, 186–190
   for visiting directories, 302
tryLock method (FileChannel), 298
trySetAccessible method (AccessibleObject), 170
try-with-resources statement, 187–189
   closing output streams with, 290
   for file locking, 298
type bounds, 210–211, 229
   annotating, 382
type erasure, 216–219, 224
   clashes after, 224–225
Type interface, 228
type method (ServiceLoader.Provider), 167
type parameters, 109, 208–209
   and primitive types, 209, 220
   annotating, 380
type variables
   and exceptions, 225–226
   in static context, 224
   no instantiating of, 221–223
   wildcards with, 214–215
TypeElement interface, 395
TypeVariable interface, 228
Unix operating system
   bash scripts, 461
   path separator, 81, 248
   specifying locales in, 426
   wildcard in classpath in, 82
   unlock method (ReentrantLock), 358
unmodifiableXxx methods (Collections), 240
unordered method (Stream), 282
until method (LocalDate), 405–406
updateAndGet method (AtomicXxx), 355
URI class, 308
URL class, 308
   final, 141
   getInputStream method, 306
   openConnection method, 306
   openStream method, 288
URLClassLoader class, 163
URLConnection class, 306–307
   connect method, 306
   getHeaderFields method, 307
   getInputStream method, 307
   getOutputStream method, 306
   setDoOutput method, 306
   setRequestProperty method, 306
URLs, reading from, 288, 306
useLocale method (Scanner), 427
user directory, 299
user interface. See GUI
user preferences, 439–441
user.dir, user.home, user.name system properties, 248
userXxx methods (Preferences), 439
uses keyword, 491
UTC (coordinated universal time), 411
UTF-8 encoding, 290–291
   for source files, 438
   modified, 296
UTF-16 encoding, 13, 31, 279, 291
   in regular expressions, 311
Util class, 165
V
   V formatting symbol (date/time), 416
   \v, \V, in regular expressions, 312
valueOf method
   of BitSet, 249
variables, 6, 14–16
  atomic mutations of, 354–357
capturing, in lambda expressions, 124–127
declaring, 14–15
defined in interfaces, 105
deprecated, 93
documentation comments for, 91–92
effectively final, 126
final, 343, 347
holding object references, 63–64
initializing, 14–16
local, 41–43
naming, 14–15
parameter, 68
private, 65, 83
public static final, 105
redefining, 42
scope of, 41, 83
shared, 343–347, 357–358
static final. See constants
static, 74–75, 77, 85, 343
thread-local, 365–366
using an abstract class as type of, 142
visibility of, 342–344, 360
volatile, 343–344
@version tag (javadoc), 91, 95
versioning, 324
views, 252–254
virtual machine, 4
  instruction reordering in, 343
visibility, 342–344
guaranteed with locks, 360
visitFile, visitFileFailed methods (FileVisitor), 304
void keyword, 2, 53
  using class literals with, 160
volatile modifier, 343–344
W
\w, \W, in regular expressions, 312
wait method (Object), 146, 361–362
waitFor method (Process), 369
waiting on a condition, 361
walk method (Files), 302–305
walkFileTree method (Files), 302, 304
warning method (Logger), 197
warnings
  for switch statements, 159
  suppressing, 220, 224, 388
weak references, 251
weaker access privilege, 139
WeakHashMap class, 251
weakly consistent iterators, 350
WeakReference class, 252
web pages
  extracting links from, 337
  reading, 338, 340
whenComplete method (CompletableFuture), 336, 338–339
while statement, 38–39
  breaking/continuing, 40
  continuing, 40
  declaring variables for, 42
white space
  in regular expressions, 312
  removing, 28
wildcards
  annotating, 382
  capturing, 216
  for annotation processors, 394
  for types, 212–214
  in class path, 81
  unbounded, 215
  with imported classes, 83–84
  with type variables, 214–215
WildcardType interface, 228
Window class, 83
WindowAdapter class, 106
WindowListener interface, 106
with method (Temporal), 408
withInitial method (ThreadLocal), 365
withLocale method (DateTimeFormatter), 413, 430
withXxx methods
  of LocalDate, 406
  of LocalTime, 409
  of ZonedDateTime, 411
words
  in regular expressions, 312
  reading from a file, 293
  sorting alphabetically, 431–432
working directory, 367
wrapper classes, 46–47
write method
  of files, 295, 302
  of OutputStream, 290
writeExternal method (Externalizable), 322
writeObject method (ObjectOutputStream), 320–322
Writer class, 294–295
  write method, 294
writeReplace method (Serializable), 322–323
writers, 288
writeXxx methods (DataOutput), 296–297, 322
X
  x formatting symbol (date/time), 416
  x, X conversion characters, 34
  \x, in regular expressions, 311
XML descriptors, generating, 398
XmlElement annotation, 480
XmlRootElement annotation, 480
xor method (BitSet), 249
Y
  y formatting symbol (date/time), 416
Year, YearMonth classes, 407
Z
  z, Z formatting symbols (date/time), 414, 416
  \z, \Z, in regular expressions, 314
ZIP file systems, 305
ZipInputStream, ZipOutputStream classes, 305
zoned time, 404–407, 410–413
ZonedDateTime class, 410–413
  and legacy classes, 417
  getDayOfXxx methods, 411
  getMonthXxx, getYear, getXxx, isXxx
    methods, 412
  minus, minusXxx, now, ofInstant methods, 411
  of method, 410–411
  parse method, 430
  plus, plusXxx methods, 411–412
toInstant method, 410, 412
toLocalXxx methods, 412
  withXxx methods, 411
ZoneId class, 410