contents

preface xix
acknowledgments xxi
about this book xxiii
about the authors xxviii
about the cover illustration xxx

PART 1 FUNDAMENTALS 1

1 *Java 8, 9, 10, and 11: what's happening? 3*

1.1 So, what's the big story? 3
1.2 Why is Java still changing? 6
Java's place in the programming language ecosystem 6
Stream processing 8 ▪ Passing code to methods with behavior parameterization 9 ▪ Parallelism and shared mutable data 10
Java needs to evolve 11
1.3 Functions in Java 12
Methods and lambdas as first-class citizens 12 ▪ Passing code: an example 14 ▪ From passing methods to lambdas 16
1.4 Streams 17
Multithreading is difficult 19

1.5 Default methods and Java modules 21
1.6 Other good ideas from functional programming 23

2 *Passing code with behavior parameterization* 26

2.1 Coping with changing requirements 27

First attempt: filtering green apples 28 ▪ Second attempt: parameterizing the color 28 ▪ Third attempt: filtering with every attribute you can think of 29

2.2 Behavior parameterization 30

Fourth attempt: filtering by abstract criteria 31

2.3 Tackling verbosity 35

Anonymous classes 36 ▪ Fifth attempt: using an anonymous class 36 ▪ Sixth attempt: using a lambda expression 37 Seventh attempt: abstracting over List type 38

2.4 Real-world examples 39

Sorting with a Comparator 39 ▪ Executing a block of code with Runnable 40 ▪ Returning a result using Callable 40 GUI event handling 41

3 *Lambda expressions* 42

3.1 Lambdas in a nutshell 43
3.2 Where and how to use lambdas 46

Functional interface 46 ▪ Function descriptor 48

3.3 Putting lambdas into practice: the execute-around pattern 50

Step 1: Remember behavior parameterization 51 ▪ Step 2: Use a functional interface to pass behaviors 51 ▪ Step 3: Execute a behavior! 52 ▪ Step 4: Pass lambdas 52

3.4 Using functional interfaces 53

Predicate 54 ▪ Consumer 54 ▪ Function 55

3.5 Type checking, type inference, and restrictions 59

Type checking 59 ▪ Same lambda, different functional interfaces 61 ▪ Type inference 63 ▪ Using local variables 63

3.6 Method references 64

In a nutshell 65 ▪ Constructor references 68

3.7 Putting lambdas and method references into practice 70

Step 1: Pass code 71 ▪ Step 2: Use an anonymous class 71 Step 3: Use lambda expressions 71 ▪ Step 4: Use method references 72

3.8 Useful methods to compose lambda expressions 72

Composing Comparators 73 ▪ *Composing Predicates 73*
Composing Functions 74

3.9 Similar ideas from mathematics 76

Integration 76 ▪ *Connecting to Java 8 lambdas 77*

PART 2 FUNCTIONAL-STYLE DATA PROCESSING WITH STREAMS 79

4 ***Introducing streams 81***

4.1 What are streams? 82

4.2 Getting started with streams 86

4.3 Streams vs. collections 88

Traversable only once 90 ▪ *External vs. internal iteration 91*

4.4 Stream operations 93

Intermediate operations 94 ▪ *Terminal operations 95*
Working with streams 95

4.5 Road map 96

5 ***Working with streams 98***

5.1 Filtering 99

Filtering with a predicate 99 ▪ *Filtering unique elements 100*

5.2 Slicing a stream 100

Slicing using a predicate 101 ▪ *Truncating a stream 102*
Skipping elements 103

5.3 Mapping 104

Applying a function to each element of a stream 104
Flattening streams 105

5.4 Finding and matching 108

Checking to see if a predicate matches at least one element 108
Checking to see if a predicate matches all elements 109
Finding an element 109 ▪ *Finding the first element 110*

5.5 Reducing 111

Summing the elements 111 ▪ *Maximum and minimum 113*

5.6 Putting it all into practice 117

The domain: Traders and Transactions 117 ▪ *Solutions 118*

5.7 Numeric streams 121

Primitive stream specializations 121 ▪ Numeric ranges 123
Putting numerical streams into practice: Pythagorean triples 123

5.8 Building streams 126

Streams from values 126 ▪ Stream from nullable 126
Streams from arrays 127 ▪ Streams from files 127
Streams from functions: creating infinite streams! 128

5.9 Overview 132

6 *Collecting data with streams 134*

6.1 Collectors in a nutshell 136

Collectors as advanced reductions 136 ▪ Predefined collectors 137

6.2 Reducing and summarizing 138

Finding maximum and minimum in a stream of values 138
Summarization 139 ▪ Joining Strings 140 ▪ Generalized summarization with reduction 141

6.3 Grouping 146

Manipulating grouped elements 147 ▪ Multilevel grouping 149
Collecting data in subgroups 150

6.4 Partitioning 154

Advantages of partitioning 155 ▪ Partitioning numbers into prime and nonprime 156

6.5 The Collector interface 159

Making sense of the methods declared by Collector interface 160
Putting them all together 163

6.6 Developing your own collector for better performance 165

Divide only by prime numbers 166 ▪ Comparing collectors' performances 170

7 *Parallel data processing and performance 172*

7.1 Parallel streams 173

Turning a sequential stream into a parallel one 174
Measuring stream performance 176 ▪ Using parallel streams correctly 180 ▪ Using parallel streams effectively 182

7.2 The fork/join framework 184

Working with RecursiveTask 184 ▪ Best practices for using the fork/join framework 188 ▪ Work stealing 189

7.3 Spliterator 190

The splitting process 191 ▪ Implementing your own Spliterator 192

PART 3 EFFECTIVE PROGRAMMING WITH STREAMS AND LAMBDAS 199

8 ***Collection API enhancements 201***

8.1 Collection factories 202

List factory 203 ▪ Set factory 204 ▪ Map factories 204

8.2 Working with List and Set 205

removeIf 205 ▪ replaceAll 206

8.3 Working with Map 207

forEach 207 ▪ Sorting 208 ▪ getOrDefault 208
Compute patterns 209 ▪ Remove patterns 210
Replacement patterns 211 ▪ Merge 211

8.4 Improved ConcurrentHashMap 213

Reduce and Search 213 ▪ Counting 214 ▪ Set views 214

9 ***Refactoring, testing, and debugging 216***

9.1 Refactoring for improved readability and flexibility 217

Improving code readability 217 ▪ From anonymous classes to lambda expressions 217 ▪ From lambda expressions to method references 219 ▪ From imperative data processing to Streams 220
Improving code flexibility 221

9.2 Refactoring object-oriented design patterns with lambdas 223

Strategy 224 ▪ Template method 225 ▪ Observer 226
Chain of responsibility 229 ▪ Factory 230

9.3 Testing lambdas 232

Testing the behavior of a visible lambda 232 ▪ Focusing on the behavior of the method using a lambda 233 ▪ Pulling complex lambdas into separate methods 234 ▪ Testing high-order functions 234

9.4 Debugging 234

Examining the stack trace 235 ▪ Logging information 236

10 ***Domain-specific languages using lambdas 239***

10.1 A specific language for your domain 241

Pros and cons of DSLs 242 ▪ Different DSL solutions available on the JVM 244

10.2 Small DSLs in modern Java APIs 248

The Stream API seen as a DSL to manipulate collections 249 Collectors as a DSL to aggregate data 250

10.3 Patterns and techniques to create DSLs in Java 252

Method chaining 255 ▪ Using nested functions 257 Function sequencing with lambda expressions 259 Putting it all together 261 ▪ Using method references in a DSL 263

10.4 Real World Java 8 DSL 266

jOOQ 266 ▪ Cucumber 267 ▪ Spring Integration 269

PART 4 EVERYDAY JAVA 273

11 ***Using Optional as a better alternative to null 275***

11.1 How do you model the absence of a value? 276

Reducing NullPointerExceptions with defensive checking 277 Problems with null 278 ▪ What are the alternatives to null in other languages? 279

11.2 Introducing the Optional class 280

11.3 Patterns for adopting Optionals 281

Creating Optional objects 281 ▪ Extracting and transforming values from Optionals with map 282 ▪ Chaining Optional objects with flatMap 283 ▪ Manipulating a stream of optionals 287 Default actions and unwrapping an Optional 288 ▪ Combining two Optionals 289 ▪ Rejecting certain values with filter 290

11.4 Practical examples of using Optional 292

Wrapping a potentially null value in an Optional 292 Exceptions vs. Optional 293 ▪ Primitive optionals and why you shouldn't use them 294 ▪ Putting it all together 294

12 *New Date and Time API* 297

12.1 LocalDate, LocalTime, LocalDateTime, Instant, Duration, and Period 298

Working with LocalDate and LocalTime 299 ▪ *Combining a date and a time* 300 ▪ *Instant: a date and time for machines* 301 *Defining a Duration or a Period* 301

12.2 Manipulating, parsing, and formatting dates 303

Working with TemporalAdjusters 305 ▪ *Printing and parsing date-time objects* 308

12.3 Working with different time zones and calendars 310

Using time zones 310 ▪ *Fixed offset from UTC/Greenwich* 311 *Using alternative calendar systems* 311

13 *Default methods* 314

13.1 Evolving APIs 317

API version 1 317 ▪ *API version 2* 318

13.2 Default methods in a nutshell 320

13.3 Usage patterns for default methods 322

Optional methods 322 ▪ *Multiple inheritance of behavior* 323

13.4 Resolution rules 326

Three resolution rules to know 327 ▪ *Most specific default-providing interface wins* 327 ▪ *Conflicts and explicit disambiguation* 329 ▪ *Diamond problem* 330

14 *The Java Module System* 333

14.1 The driving force: reasoning about software 334

Separation of concerns 334 ▪ *Information hiding* 334 *Java software* 335

14.2 Why the Java Module System was designed 336

Modularity limitations 336 ▪ *Monolithic JDK* 337 *Comparison with OSGi* 338

14.3 Java modules: the big picture 339

14.4 Developing an application with the Java Module System 340

Setting up an application 340 ▪ *Fine-grained and coarse-grained modularization* 342 ▪ *Java Module System basics* 342

14.5 Working with several modules 343

The exports clause 344 ▪ The requires clause 344 Naming 345

14.6 Compiling and packaging 345

14.7 Automatic modules 349

14.8 Module declaration and clauses 350

requires 350 ▪ exports 350 ▪ requires transitive 351 exports to 351 ▪ open and opens 351 ▪ uses and provides 352

14.9 A bigger example and where to learn more 352

PART 5 ENHANCED JAVA CONCURRENCY 355

15 ***Concepts behind CompletableFuture and reactive programming 357***

15.1 Evolving Java support for expressing concurrency 360

Threads and higher-level abstractions 361 ▪ Executors and thread pools 362 ▪ Other abstractions of threads: non-nested with method calls 364 ▪ What do you want from threads? 366

15.2 Synchronous and asynchronous APIs 366

Future-style API 368 ▪ Reactive-style API 369 ▪ Sleeping (and other blocking operations) considered harmful 370 Reality check 372 ▪ How do exceptions work with asynchronous APIs? 372

15.3 The box-and-channel model 373

15.4 CompletableFuture and combinators for concurrency 375

15.5 Publish-subscribe and reactive programming 378

Example use for summing two flows 380 ▪ Backpressure 384 A simple form of real backpressure 384

15.6 Reactive systems vs. reactive programming 385

15.7 Road map 386

16 ***CompletableFuture: composable asynchronous programming 387***

16.1 Simple use of Futures 388

Understanding Futures and their limitations 389 ▪ Using CompletableFutures to build an asynchronous application 390

16.2 Implementing an asynchronous API 391

Converting a synchronous method into an asynchronous one 392
Dealing with errors 394

16.3 Making your code nonblocking 396

Parallelizing requests using a parallel Stream 397
Making asynchronous requests with CompletableFutures 397
Looking for the solution that scales better 399 ▪ Using a custom Executor 400

16.4 Pipelining asynchronous tasks 402

Implementing a discount service 403 ▪ Using the Discount service 404 ▪ Composing synchronous and asynchronous operations 405 ▪ Combining two CompletableFutures: dependent and independent 408 ▪ Reflecting on Future vs. CompletableFuture 409 ▪ Using timeouts effectively 410

16.5 Reacting to a CompletableFuture completion 411

Refactoring the best-price-finder application 412
Putting it all together 414

16.6 Road map 414

17 ***Reactive programming 416***

17.1 The Reactive Manifesto 417

Reactive at application level 418 ▪ Reactive at system level 420

17.2 Reactive streams and the Flow API 421

Introducing the Flow class 421 ▪ Creating your first reactive application 424 ▪ Transforming data with a Processor 429
Why doesn't Java provide an implementation of the Flow API? 431

17.3 Using the reactive library RxJava 431

Creating and using an Observable 433 ▪ Transforming and combining Observables 437

PART 6 FUNCTIONAL PROGRAMMING AND FUTURE JAVA EVOLUTION 443

18 ***Thinking functionally 445***

18.1 Implementing and maintaining systems 446

Shared mutable data 446 ▪ Declarative programming 447
Why functional programming? 448

18.2 What's functional programming? 449

Functional-style Java 450 ▪ Referential transparency 452 Object-oriented vs. functional-style programming 452 Functional style in practice 453

18.3 Recursion vs. iteration 455

19 **Functional programming techniques 460**

19.1 Functions everywhere 461

Higher-order functions 461 ▪ Currying 463

19.2 Persistent data structures 464

Destructive updates vs. functional 464 ▪ Another example with Trees 467 ▪ Using a functional approach 468

19.3 Lazy evaluation with streams 469

Self-defining stream 470 ▪ Your own lazy list 472

19.4 Pattern matching 476

Visitor design pattern 477 ▪ Pattern matching to the rescue 478

19.5 Miscellany 481

Caching or memoization 481 ▪ What does "Return the same object" mean? 482 ▪ Combinators 483

20 **Blending OOP and FP: Comparing Java and Scala 485**

20.1 Introduction to Scala 486

Hello beer 486 ▪ Basic data structures: List, Set, Map, Tuple, Stream, Option 488

20.2 Functions 493

First-class functions in Scala 493 ▪ Anonymous functions and closures 494 ▪ Currying 496

20.3 Classes and traits 497

Less verbosity with Scala classes 497 ▪ Scala traits vs. Java interfaces 498

21 **Conclusions and where next for Java 500**

21.1 Review of Java 8 features 501

Behavior parameterization (lambdas and method references) 501 Streams 502 ▪ CompletableFuture 502 ▪ Optional 503 Flow API 503 ▪ Default methods 504

21.2 The Java 9 module system 504

21.3 Java 10 local variable type inference 505

21.4 What's ahead for Java? 507

Declaration-site variance 507 ▪ Pattern matching 507 Richer forms of generics 508 ▪ Deeper support for immutability 510 ▪ Value types 511

21.5 Moving Java forward faster 514

21.6 The final word 515

appendix A Miscellaneous language updates 517

appendix B Miscellaneous library updates 521

appendix C Performing multiple operations in parallel on a stream 529

appendix D Lambdas and JVM bytecode 538

index 543