

C O R E
JAVA

Volume II—Advanced Features

ELEVENTH EDITION



CAY S. HORSTMANN

Core Java



Volume II—Advanced Features

Eleventh Edition

This page intentionally left blank

Core Java

Volume II—Advanced Features

Eleventh Edition

Cay S. Horstmann

◆◆ Addison-Wesley

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

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

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

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

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

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

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

Visit us on the Web: informit.com

Library of Congress Preassigned Control Number: 2018963595

Copyright © 2019 Pearson Education Inc.

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

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

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

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

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

ISBN-13: 978-0-13-516631-4

ISBN-10: 0-13-516631-4

ScoutAutomatedPrintCode

Contents

<i>Preface</i>	<i>xv</i>
<i>Acknowledgments</i>	<i>xxi</i>
Chapter 1: Streams	1
1.1 From Iterating to Stream Operations	2
1.2 Stream Creation	5
1.3 The filter, map, and flatMap Methods	11
1.4 Extracting Substreams and Combining Streams	12
1.5 Other Stream Transformations	14
1.6 Simple Reductions	15
1.7 The Optional Type	16
1.7.1 Getting an Optional Value	17
1.7.2 Consuming an Optional Value	17
1.7.3 Pipelining Optional Values	18
1.7.4 How Not to Work with Optional Values	19
1.7.5 Creating Optional Values	20
1.7.6 Composing Optional Value Functions with flatMap	21
1.7.7 Turning an Optional into a Stream	22
1.8 Collecting Results	25
1.9 Collecting into Maps	30
1.10 Grouping and Partitioning	34
1.11 Downstream Collectors	36
1.12 Reduction Operations	41
1.13 Primitive Type Streams	43
1.14 Parallel Streams	48
Chapter 2: Input and Output	55
2.1 Input/Output Streams	56
2.1.1 Reading and Writing Bytes	56
2.1.2 The Complete Stream Zoo	59

2.1.3	Combining Input/Output Stream Filters	63
2.1.4	Text Input and Output	68
2.1.5	How to Write Text Output	68
2.1.6	How to Read Text Input	70
2.1.7	Saving Objects in Text Format	72
2.1.8	Character Encodings	75
2.2	Reading and Writing Binary Data	78
2.2.1	The <code>DataInput</code> and <code>DataOutput</code> interfaces	78
2.2.2	Random-Access Files	80
2.2.3	ZIP Archives	85
2.3	Object Input/Output Streams and Serialization	88
2.3.1	Saving and Loading Serializable Objects	88
2.3.2	Understanding the Object Serialization File Format	93
2.3.3	Modifying the Default Serialization Mechanism	100
2.3.4	Serializing Singletons and Typesafe Enumerations	102
2.3.5	Versioning	103
2.3.6	Using Serialization for Cloning	106
2.4	Working with Files	109
2.4.1	Paths	109
2.4.2	Reading and Writing Files	112
2.4.3	Creating Files and Directories	113
2.4.4	Copying, Moving, and Deleting Files	114
2.4.5	Getting File Information	116
2.4.6	Visiting Directory Entries	118
2.4.7	Using Directory Streams	120
2.4.8	ZIP File Systems	123
2.5	Memory-Mapped Files	124
2.5.1	Memory-Mapped File Performance	125
2.5.2	The Buffer Data Structure	132
2.6	File Locking	134
2.7	Regular Expressions	137
2.7.1	The Regular Expression Syntax	137
2.7.2	Matching a String	142
2.7.3	Finding Multiple Matches	145

2.7.4	Splitting along Delimiters	147
2.7.5	Replacing Matches	148
Chapter 3: XML	153
3.1	Introducing XML	154
3.2	The Structure of an XML Document	156
3.3	Parsing an XML Document	159
3.4	Validating XML Documents	169
3.4.1	Document Type Definitions	171
3.4.2	XML Schema	179
3.4.3	A Practical Example	182
3.5	Locating Information with XPath	188
3.6	Using Namespaces	193
3.7	Streaming Parsers	196
3.7.1	Using the SAX Parser	197
3.7.2	Using the StAX Parser	202
3.8	Generating XML Documents	206
3.8.1	Documents without Namespaces	206
3.8.2	Documents with Namespaces	207
3.8.3	Writing Documents	208
3.8.4	Writing an XML Document with StAX	210
3.8.5	An Example: Generating an SVG File	215
3.9	XSL Transformations	216
Chapter 4: Networking	227
4.1	Connecting to a Server	227
4.1.1	Using Telnet	227
4.1.2	Connecting to a Server with Java	230
4.1.3	Socket Timeouts	232
4.1.4	Internet Addresses	234
4.2	Implementing Servers	236
4.2.1	Server Sockets	236
4.2.2	Serving Multiple Clients	239
4.2.3	Half-Close	243
4.2.4	Interruptible Sockets	244
4.3	Getting Web Data	251

4.3.1	URLs and URIs	251
4.3.2	Using a URLConnection to Retrieve Information	254
4.3.3	Posting Form Data	261
4.4	The HTTP Client	271
4.5	Sending E-Mail	278
Chapter 5: Database Programming		283
5.1	The Design of JDBC	284
5.1.1	JDBC Driver Types	285
5.1.2	Typical Uses of JDBC	286
5.2	The Structured Query Language	287
5.3	JDBC Configuration	293
5.3.1	Database URLs	294
5.3.2	Driver JAR Files	294
5.3.3	Starting the Database	294
5.3.4	Registering the Driver Class	295
5.3.5	Connecting to the Database	296
5.4	Working with JDBC Statements	299
5.4.1	Executing SQL Statements	299
5.4.2	Managing Connections, Statements, and Result Sets	303
5.4.3	Analyzing SQL Exceptions	304
5.4.4	Populating a Database	306
5.5	Query Execution	310
5.5.1	Prepared Statements	311
5.5.2	Reading and Writing LOBs	317
5.5.3	SQL Escapes	319
5.5.4	Multiple Results	321
5.5.5	Retrieving Autogenerated Keys	322
5.6	Scrollable and Updatable Result Sets	322
5.6.1	Scrollable Result Sets	323
5.6.2	Updatable Result Sets	325
5.7	Row Sets	329
5.7.1	Constructing Row Sets	330
5.7.2	Cached Row Sets	330
5.8	Metadata	334
5.9	Transactions	344

5.9.1	Programming Transactions with JDBC	344
5.9.2	Save Points	345
5.9.3	Batch Updates	345
5.9.4	Advanced SQL Types	348
5.10	Connection Management in Web and Enterprise Applications ..	349
Chapter 6: The Date and Time API		353
6.1	The Time Line	354
6.2	Local Dates	358
6.3	Date Adjusters	364
6.4	Local Time	365
6.5	Zoned Time	367
6.6	Formatting and Parsing	371
6.7	Interoperating with Legacy Code	376
Chapter 7: Internationalization		379
7.1	Locales	380
7.1.1	Why Locales?	380
7.1.2	Specifying Locales	381
7.1.3	The Default Locale	384
7.1.4	Display Names	384
7.2	Number Formats	387
7.2.1	Formatting Numeric Values	387
7.2.2	Currencies	393
7.3	Date and Time	394
7.4	Collation and Normalization	402
7.5	Message Formatting	409
7.5.1	Formatting Numbers and Dates	409
7.5.2	Choice Formats	411
7.6	Text Input and Output	413
7.6.1	Text Files	414
7.6.2	Line Endings	414
7.6.3	The Console	414
7.6.4	Log Files	415
7.6.5	The UTF-8 Byte Order Mark	415
7.6.6	Character Encoding of Source Files	416
7.7	Resource Bundles	417

7.7.1	Locating Resource Bundles	417
7.7.2	Property Files	418
7.7.3	Bundle Classes	419
7.8	A Complete Example	421
Chapter 8: Scripting, Compiling, and Annotation Processing		439
8.1	Scripting for the Java Platform	440
8.1.1	Getting a Scripting Engine	440
8.1.2	Script Evaluation and Bindings	441
8.1.3	Redirecting Input and Output	444
8.1.4	Calling Scripting Functions and Methods	444
8.1.5	Compiling a Script	446
8.1.6	An Example: Scripting GUI Events	447
8.2	The Compiler API	452
8.2.1	Invoking the Compiler	453
8.2.2	Launching a Compilation Task	453
8.2.3	Capturing Diagnostics	454
8.2.4	Reading Source Files from Memory	454
8.2.5	Writing Byte Codes to Memory	455
8.2.6	An Example: Dynamic Java Code Generation	457
8.3	Using Annotations	463
8.3.1	An Introduction into Annotations	464
8.3.2	An Example: Annotating Event Handlers	465
8.4	Annotation Syntax	471
8.4.1	Annotation Interfaces	471
8.4.2	Annotations	473
8.4.3	Annotating Declarations	475
8.4.4	Annotating Type Uses	476
8.4.5	Annotating this	477
8.5	Standard Annotations	478
8.5.1	Annotations for Compilation	480
8.5.2	Annotations for Managing Resources	480
8.5.3	Meta-Annotations	481
8.6	Source-Level Annotation Processing	484
8.6.1	Annotation Processors	484
8.6.2	The Language Model API	485

8.6.3	Using Annotations to Generate Source Code	486
8.7	Bytecode Engineering	489
8.7.1	Modifying Class Files	490
8.7.2	Modifying Bytecodes at Load Time	495
Chapter 9: The Java Platform Module System		499
9.1	The Module Concept	500
9.2	Naming Modules	501
9.3	The Modular “Hello, World!” Program	502
9.4	Requiring Modules	504
9.5	Exporting Packages	506
9.6	Modular JARs	510
9.7	Modules and Reflective Access	511
9.8	Automatic Modules	515
9.9	The Unnamed Module	517
9.10	Command-Line Flags for Migration	518
9.11	Transitive and Static Requirements	519
9.12	Qualified Exporting and Opening	521
9.13	Service Loading	522
9.14	Tools for Working with Modules	524
Chapter 10: Security		529
10.1	Class Loaders	530
10.1.1	The Class-Loading Process	530
10.1.2	The Class Loader Hierarchy	532
10.1.3	Using Class Loaders as Namespaces	534
10.1.4	Writing Your Own Class Loader	534
10.1.5	Bytecode Verification	541
10.2	Security Managers and Permissions	546
10.2.1	Permission Checking	546
10.2.2	Java Platform Security	547
10.2.3	Security Policy Files	551
10.2.4	Custom Permissions	559
10.2.5	Implementation of a Permission Class	560
10.3	User Authentication	566
10.3.1	The JAAS Framework	566
10.3.2	JAAS Login Modules	573

10.4	Digital Signatures	582
10.4.1	Message Digests	583
10.4.2	Message Signing	587
10.4.3	Verifying a Signature	589
10.4.4	The Authentication Problem	592
10.4.5	Certificate Signing	594
10.4.6	Certificate Requests	596
10.4.7	Code Signing	597
10.5	Encryption	599
10.5.1	Symmetric Ciphers	600
10.5.2	Key Generation	602
10.5.3	Cipher Streams	607
10.5.4	Public Key Ciphers	608
Chapter 11: Advanced Swing and Graphics		613
11.1	Tables	613
11.1.1	A Simple Table	614
11.1.2	Table Models	618
11.1.3	Working with Rows and Columns	622
11.1.3.1	Column Classes	622
11.1.3.2	Accessing Table Columns	623
11.1.3.3	Resizing Columns	624
11.1.3.4	Resizing Rows	625
11.1.3.5	Selecting Rows, Columns, and Cells	626
11.1.3.6	Sorting Rows	627
11.1.3.7	Filtering Rows	628
11.1.3.8	Hiding and Displaying Columns	630
11.1.4	Cell Rendering and Editing	639
11.1.4.1	Rendering Cells	639
11.1.4.2	Rendering the Header	641
11.1.4.3	Editing Cells	641
11.1.4.4	Custom Editors	642
11.2	Trees	652
11.2.1	Simple Trees	654
11.2.1.1	Editing Trees and Tree Paths	663
11.2.2	Node Enumeration	672

11.2.3	Rendering Nodes	674
11.2.4	Listening to Tree Events	677
11.2.5	Custom Tree Models	684
11.3	Advanced AWT	693
11.3.1	The Rendering Pipeline	694
11.3.2	Shapes	696
11.3.2.1	The Shape Class Hierarchy	697
11.3.2.2	Using the Shape Classes	698
11.3.3	Areas	714
11.3.4	Strokes	715
11.3.5	Paint	724
11.3.6	Coordinate Transformations	727
11.3.7	Clipping	733
11.3.8	Transparency and Composition	735
11.4	Raster Images	744
11.4.1	Readers and Writers for Images	745
11.4.1.1	Obtaining Readers and Writers for Image File Types	745
11.4.1.2	Reading and Writing Files with Multiple Images	747
11.4.2	Image Manipulation	756
11.4.2.1	Constructing Raster Images	756
11.4.2.2	Filtering Images	763
11.5	Printing	772
11.5.1	Graphics Printing	772
11.5.2	Multiple-Page Printing	782
11.5.3	Print Services	792
11.5.4	Stream Print Services	796
11.5.5	Printing Attributes	799
Chapter 12: Native Methods	809	
12.1	Calling a C Function from a Java Program	810
12.2	Numeric Parameters and Return Values	817
12.3	String Parameters	819
12.4	Accessing Fields	825
12.4.1	Accessing Instance Fields	825

12.4.2	Accessing Static Fields	829
12.5	Encoding Signatures	831
12.6	Calling Java Methods	832
12.6.1	Instance Methods	833
12.6.2	Static Methods	834
12.6.3	Constructors	835
12.6.4	Alternative Method Invocations	835
12.7	Accessing Array Elements	840
12.8	Handling Errors	844
12.9	Using the Invocation API	849
12.10	A Complete Example: Accessing the Windows Registry	855
12.10.1	Overview of the Windows Registry	855
12.10.2	A Java Platform Interface for Accessing the Registry	856
12.10.3	Implementation of Registry Access Functions as Native Methods	857
<i>Index</i>	873

Preface



To the Reader

The book you have in your hands is the second volume of the eleventh edition of *Core Java*, fully updated for Java SE 11. The first volume covers the essential features of the language; this volume deals with the advanced topics that a programmer needs to know for professional software development. Thus, as with the first volume and the previous editions of this book, we are still targeting programmers who want to put Java technology to work in real projects.

As is the case with any book, errors and inaccuracies are inevitable. Should you find any in this book, we would very much like to hear about them. Of course, we would prefer to hear about them only once. For this reason, we have put up a web site at <http://horstmann.com/corejava> with a FAQ, bug fixes, and workarounds. Strategically placed at the end of the bug report web page (to encourage you to read the previous reports) is a form that you can use to report bugs or problems and to send suggestions for improvements for future editions.

About This Book

The chapters in this book are, for the most part, independent of each other. You should be able to delve into whatever topic interests you the most and read the chapters in any order.

In **Chapter 1**, you will learn all about the Java stream library that brings a modern flavor to processing data, by specifying what you want without describing in detail how the result should be obtained. This allows the stream library to focus on an optimal evaluation strategy, which is particularly advantageous for optimizing concurrent computations.

The topic of **Chapter 2** is input and output handling (I/O). In Java, all input and output is handled through input/output streams. These streams (not to be confused with those in Chapter 1) let you deal, in a uniform manner, with communications among various sources of data, such as files, network connections, or memory blocks. We include detailed coverage of the reader and

writer classes that make it easy to deal with Unicode. We show you what goes on under the hood when you use the object serialization mechanism, which makes saving and loading objects easy and convenient. We then move on to regular expressions and working with files and paths. Throughout this chapter, you will find welcome enhancements in recent Java versions.

Chapter 3 covers XML. We show you how to parse XML files, how to generate XML, and how to use XSL transformations. As a useful example, we show you how to specify the layout of a Swing form in XML. We also discuss the XPath API, which makes finding needles in XML haystacks much easier.

Chapter 4 covers the networking API. Java makes it phenomenally easy to do complex network programming. We show you how to make network connections to servers, how to implement your own servers, and how to make HTTP connections. This chapter includes coverage of the new HTTP client.

Chapter 5 covers database programming. The main focus is on JDBC, the Java database connectivity API that lets Java programs connect to relational databases. We show you how to write useful programs to handle realistic database chores, using a core subset of the JDBC API. (A complete treatment of the JDBC API would require a book almost as big as this one.) We finish the chapter with a brief introduction into hierarchical databases and discuss JNDI (the Java Naming and Directory Interface) and LDAP (the Lightweight Directory Access Protocol).

Java had two prior attempts at libraries for handling date and time. The third one was the `java.util.Date` class in Java 8. In **Chapter 6**, you will learn how to deal with the complexities of calendars and time zones, using the new date and time library.

Chapter 7 discusses a feature that we believe can only grow in importance: internationalization. The Java programming language is one of the few languages designed from the start to handle Unicode, but the internationalization support on the Java platform goes much further. As a result, you can internationalize Java applications so that they cross not only platforms but country boundaries as well. For example, we show you how to write a retirement calculator that uses either English, German, or Chinese languages.

Chapter 8 discusses three techniques for processing code. The scripting and compiler APIs allow your program to call code in scripting languages such as JavaScript or Groovy, and to compile Java code. Annotations allow you to add arbitrary information (sometimes called metadata) to a Java program. We

show you how annotation processors can harvest these annotations at the source or class file level, and how annotations can be used to influence the behavior of classes at runtime. Annotations are only useful with tools, and we hope that our discussion will help you select useful annotation processing tools for your needs.

In **Chapter 9**, you will learn about the Java Platform Module System that was introduced in Java 9 to facilitate an orderly evolution of the Java platform and core libraries. This module system provides encapsulation for packages and a mechanism for describing module requirements. You will learn the properties of modules so that you can decide whether to use them in your own applications. Even if you decide not to, you need to know the new rules so that you can interact with the Java platform and other modularized libraries.

Chapter 10 takes up the Java security model. The Java platform was designed from the ground up to be secure, and this chapter takes you under the hood to see how this design is implemented. We show you how to write your own class loaders and security managers for special-purpose applications. Then, we take up the security API that allows for such important features as message and code signing, authorization and authentication, and encryption. We conclude with examples that use the AES and RSA encryption algorithms.

Chapter 11 contains all the Swing material that didn't make it into Volume I, especially the important but complex tree and table components. We also cover the Java 2D API, which you can use to create realistic drawings and special effects. Of course, not many programmers need to program Swing user interfaces these days, so we pay particular attention to features that are useful for images that can be generated on a server.

Chapter 12 takes up native methods, which let you call methods written for a specific machine such as the Microsoft Windows API. Obviously, this feature is controversial: Use native methods, and the cross-platform nature of Java vanishes. Nonetheless, every serious programmer writing Java applications for specific platforms needs to know these techniques. At times, you need to turn to the operating system's API for your target platform when you interact with a device or service that is not supported by Java. We illustrate this by showing you how to access the registry API in Windows from a Java program.

As always, all chapters have been completely revised for the latest version of Java. Outdated material has been removed, and the new APIs of Java 9, 10, and 11 are covered in detail.

Conventions

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



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



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



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



C++ NOTE: There are a number of C++ notes that explain the difference between the Java programming language and C++. You can skip them if you aren't interested in C++.

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

Application Programming Interface 1.2

Programs whose source code is included in the companion code for this book are listed as examples, for instance

Listing 1.1 ScriptTest.java

You can download the companion code from <http://horstmann.com/corejava>.

Register your copy of *Core Java, Volume II—Advanced Features, Eleventh Edition*, on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780135166314) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

This page intentionally left blank

Acknowledgments



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

A large number of individuals at Pearson provided valuable assistance, but they managed to stay behind the scenes. I'd like them all to know how much I appreciate their efforts. As always, my warm thanks go to my editor, Greg Doench, for steering the book through the writing and production process, and for allowing me to be blissfully unaware of the existence of all those folks behind the scenes. I am very grateful to Julie Nahil for production support, and to Dmitry Kirsanov and Alina Kirsanova for copyediting and typesetting the manuscript.

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

Reviewers of this and earlier editions include Chuck Allison (Contributing Editor, *C/C++ Users Journal*), Lance Anderson (Oracle), Alec Beaton (PointBase, Inc.), Cliff Berg (iSavvix Corporation), Joshua Bloch, David Brown, Corky Cartwright, Frank Cohen (PushToTest), Chris Crane (devXsolution), Dr. Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), Robert Evans (Senior Staff, The Johns Hopkins University Applied Physics Lab), David Geary (Sabreware), Jim Gish (Oracle), Brian Goetz (Oracle), Angela Gordon, Dan Gordon, Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Steve Haines, Marty Hall (The Johns Hopkins University Applied Physics Lab), Vincent Hardy, Dan Harkey (San Jose State University), William Higgins (IBM), Vladimir Ivanovic (PointBase), Jerry Jackson (ChannelPoint Software), Tim Kimmet (Preview Systems), Chris Laffra, Charlie Lai, Angelika Langer, Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University), Hao Pham, Paul Phillion, Blake Ragsdell, Ylber Ramadani (Ryerson University),

Stuart Reges (University of Arizona), Simon Ritter, Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nice, France), Dr. Paul Sanghera (San Jose State University and Brooks College), Paul Sevinc (Teamup AG), Yoshiki Shabata, Devang Shah, Richard Slywczak (NASA/Glenn Research Center), Bradley A. Smith, Steven Stelling, Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (author of *Core JFC, Second Edition*), Janet Traub, Paul Tyma (consultant), Christian Ullenboom, Peter van der Linden, Burt Walsh, Joe Wang (Oracle), and Dan Xu (Oracle).

Cay Horstmann
San Francisco, California
December 2018

Streams

In this chapter

- 1.1 From Iterating to Stream Operations, page 2
- 1.2 Stream Creation, page 5
- 1.3 The `filter`, `map`, and `flatMap` Methods, page 11
- 1.4 Extracting Substreams and Combining Streams, page 12
- 1.5 Other Stream Transformations, page 14
- 1.6 Simple Reductions, page 15
- 1.7 The Optional Type, page 16
- 1.8 Collecting Results, page 25
- 1.9 Collecting into Maps, page 30
- 1.10 Grouping and Partitioning, page 34
- 1.11 Downstream Collectors, page 36
- 1.12 Reduction Operations, page 41
- 1.13 Primitive Type Streams, page 43
- 1.14 Parallel Streams, page 48

Compared to collections, streams provide a view of data that lets you specify computations at a higher conceptual level. With a stream, you specify what you want to have done, not how to do it. You leave the scheduling of operations to the implementation. For example, suppose you want to compute the

average of a certain property. You specify the source of data and the property, and the stream library can then optimize the computation, for example by using multiple threads for computing sums and counts and combining the results.

In this chapter, you will learn how to use the Java stream library, which was introduced in Java 8, to process collections in a “what, not how” style.

1.1 From Iterating to Stream Operations

When you process a collection, you usually iterate over its elements and do some work with each of them. For example, suppose we want to count all long words in a book. First, let’s put them into a list:

```
var contents = Files.readStr(
    Paths.get("alice.txt")); // Read file into string
List<String> words = List.of(contents.split("\\PL+"));
// Split into words; nonletters are delimiters
```

Now we are ready to iterate:

```
int count = 0;
for (String w : words) {
    if (w.length() > 12) count++;
}
```

With streams, the same operation looks like this:

```
long count = words.stream()
    .filter(w -> w.length() > 12)
    .count();
```

Now you don’t have to scan the loop for evidence of filtering and counting. The method names tell you right away what the code intends to do. Moreover, where the loop prescribes the order of operations in complete detail, a stream is able to schedule the operations any way it wants, as long as the result is correct.

Simply changing `stream` to `parallelStream` allows the stream library to do the filtering and counting in parallel.

```
long count = words.parallelStream()
    .filter(w -> w.length() > 12)
    .count();
```

Streams follow the “what, not how” principle. In our stream example, we describe what needs to be done: get the long words and count them. We don’t specify in which order, or in which thread, this should happen. In contrast, the loop at the beginning of this section specifies exactly how the computation should work, and thereby forgoes any chances of optimization.

A stream seems superficially similar to a collection, allowing you to transform and retrieve data. But there are significant differences:

1. A stream does not store its elements. They may be stored in an underlying collection or generated on demand.
2. Stream operations don’t mutate their source. For example, the `filter` method does not remove elements from a stream but yields a new stream in which they are not present.
3. Stream operations are *lazy* when possible. This means they are not executed until their result is needed. For example, if you only ask for the first five long words instead of all, the `filter` method will stop filtering after the fifth match. As a consequence, you can even have infinite streams!

Let us have another look at the example. The `stream` and `parallelStream` methods yield a *stream* for the `words` list. The `filter` method returns another stream that contains only the words of length greater than twelve. The `count` method reduces that stream to a result.

This workflow is typical when you work with streams. You set up a pipeline of operations in three stages:

1. Create a stream.
2. Specify *intermediate operations* for transforming the initial stream into others, possibly in multiple steps.
3. Apply a *terminal operation* to produce a result. This operation forces the execution of the lazy operations that precede it. Afterwards, the stream can no longer be used.

In the example in Listing 1.1, the stream is created with the `stream` or `parallelStream` method. The `filter` method transforms it, and `count` is the terminal operation.

In the next section, you will see how to create a stream. The subsequent three sections deal with stream transformations. They are followed by five sections on terminal operations.

Listing 1.1 streams/CountLongWords.java

```
1 package streams;
2
3 /**
4  * @version 1.01 2018-05-01
5  * @author Cay Horstmann
6  */
7
8 import java.io.*;
9 import java.nio.charset.*;
10 import java.nio.file.*;
11 import java.util.*;
12
13 public class CountLongWords
14 {
15     public static void main(String[] args) throws IOException
16     {
17         var contents = Files.readString(
18             Paths.get("../gutenberg/alice30.txt"));
19         List<String> words = List.of(contents.split("\\PL+"));
20
21         long count = 0;
22         for (String w : words)
23         {
24             if (w.length() > 12) count++;
25         }
26         System.out.println(count);
27
28         count = words.stream().filter(w -> w.length() > 12).count();
29         System.out.println(count);
30
31         count = words.parallelStream().filter(w -> w.length() > 12).count();
32         System.out.println(count);
33     }
34 }
```

***java.util.stream.Stream<T>* 8**

- `Stream<T> filter(Predicate<? super T> p)`
yields a stream containing all elements of this stream fulfilling `p`.
- `long count()`
yields the number of elements of this stream. This is a terminal operation.

```
java.util.Collection<E> 1.2
```

- default Stream<E> stream()
- default Stream<E> parallelStream()
yields a sequential or parallel stream of the elements in this collection.

1.2 Stream Creation

You have already seen that you can turn any collection into a stream with the stream method of the Collection interface. If you have an array, use the static Stream.of method instead.

```
Stream<String> words = Stream.of(contents.split("\\PL+"));  
// split returns a String[] array
```

The of method has a varargs parameter, so you can construct a stream from any number of arguments:

```
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

Use Arrays.stream(array, from, to) to make a stream from a part of an array.

To make a stream with no elements, use the static Stream.empty method:

```
Stream<String> silence = Stream.empty();  
// Generic type <String> is inferred; same as Stream.<String>empty()
```

The Stream interface has two static methods for making infinite streams. The generate method takes a function with no arguments (or, technically, an object of the Supplier<T> interface). Whenever a stream value is needed, that function is called to produce a value. You can get a stream of constant values as

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

or a stream of random numbers as

```
Stream<Double> randoms = Stream.generate(Math::random);
```

To produce sequences such as 0 1 2 3 . . ., use the iterate method instead. It takes a “seed” value and a function (technically, a UnaryOperator<T>) and repeatedly applies the function to the previous result. For example,

```
Stream<BigInteger> integers  
= Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

The first element in the sequence is the seed `BigInteger.ZERO`. The second element is `f(seed)` which yields 1 (as a big integer). The next element is `f(f(seed))` which yields 2, and so on.

To produce a finite stream instead, add a predicate that specifies when the iteration should finish:

```
var limit = new BigInteger("10000000");
Stream<BigInteger> integers
    = Stream.iterate(BigInteger.ZERO,
        n -> n.compareTo(limit) < 0,
        n -> n.add(BigInteger.ONE));
```

As soon as the predicate rejects an iteratively generated value, the stream ends.

Finally, the `Stream.ofNullable` method makes a really short stream from an object. The stream has length 0 if the object is `null` or length 1 otherwise, containing just the object. This is mostly useful in conjunction with `flatMap`—see Section 1.7.7, “Turning an Optional into a Stream,” on p. 22 for an example.



NOTE: A number of methods in the Java API yield streams. For example, the `Pattern` class has a method `splitAsStream` that splits a `CharSequence` by a regular expression. You can use the following statement to split a string into words:

```
Stream<String> words = Pattern.compile("\\\\PL+").splitAsStream(contents);
```

The `Scanner.tokens` method yields a stream of tokens of a scanner. Another way to get a stream of words from a string is

```
Stream<String> words = new Scanner(contents).tokens();
```

The static `Files.lines` method returns a `Stream` of all lines in a file:

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



NOTE: If you have an `Iterable` that is not a collection, you can turn it into a stream by calling

```
StreamSupport.stream(iterable.spliterator(), false);
```

If you have an `Iterator` and want a stream of its results, use

```
StreamSupport.stream(Spliterators.spliteratorUnknownSize(
    iterator, Spliterator.ORDERED), false);
```



CAUTION: It is very important that you don't modify the collection backing a stream while carrying out a stream operation. Remember that streams don't collect their data—the data is always in a separate collection. If you modify that collection, the outcome of the stream operations becomes undefined. The JDK documentation refers to this requirement as *noninterference*.

To be exact, since intermediate stream operations are lazy, it is possible to mutate the collection up to the point where the terminal operation executes. For example, the following, while certainly not recommended, will work:

```
List<String> wordList = . . . ;
Stream<String> words = wordList.stream();
wordList.add("END");
long n = words.distinct().count();
```

But this code is wrong:

```
Stream<String> words = wordList.stream();
words.forEach(s -> if (s.length() < 12) wordList.remove(s));
// ERROR--interference
```

The example program in Listing 1.2 shows the various ways of creating a stream.

Listing 1.2 streams/CreatingStreams.java

```
1 package streams;
2
3 /**
4  * @version 1.01 2018-05-01
5  * @author Cay Horstmann
6  */
7
8 import java.io.IOException;
9 import java.math.BigInteger;
10 import java.nio.charset.StandardCharsets;
11 import java.nio.file.*;
12 import java.util.*;
13 import java.util.regex.Pattern;
14 import java.util.stream.*;
15
16 public class CreatingStreams
17 {
18     public static <T> void show(String title, Stream<T> stream)
19     {
```

(Continues)

Listing 1.2 *(Continued)*

```
20     final int SIZE = 10;
21     List<T> firstElements = stream
22         .limit(SIZE + 1)
23         .collect(Collectors.toList());
24     System.out.print(title + ": ");
25     for (int i = 0; i < firstElements.size(); i++)
26     {
27         if (i > 0) System.out.print(", ");
28         if (i < SIZE) System.out.print(firstElements.get(i));
29         else System.out.print("...");
30     }
31     System.out.println();
32 }
33
34 public static void main(String[] args) throws IOException
35 {
36     Path path = Paths.get("../gutenberg/alice30.txt");
37     var contents = Files.readStr(path);
38
39     Stream<String> words = Stream.of(contents.split("\\PL+"));
40     show("words", words);
41     Stream<String> song = Stream.of("gently", "down", "the", "stream");
42     show("song", song);
43     Stream<String> silence = Stream.empty();
44     show("silence", silence);
45
46     Stream<String> echos = Stream.generate(() -> "Echo");
47     show("echos", echos);
48
49     Stream<Double> randoms = Stream.generate(Math::random);
50     show("randoms", randoms);
51
52     Stream<BigInteger> integers = Stream.iterate(BigInteger.ONE,
53         n -> n.add(BigInteger.ONE));
54     show("integers", integers);
55
56     Stream<String> wordsAnotherWay = Pattern.compile("\\PL+").splitAsStream(contents);
57     show("wordsAnotherWay", wordsAnotherWay);
58
59     try (Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8))
60     {
61         show("lines", lines);
62     }
63
64     Iterable<Path> iterable = FileSystems.getDefault().getRootDirectories();
65     Stream<Path> rootDirectories = StreamSupport.stream(iterable.spliterator(), false);
66     show("rootDirectories", rootDirectories);
```

```
67
68     Iterator<Path> iterator = Paths.get("/usr/share/dict/words").iterator();
69     Stream<Path> pathComponents = StreamSupport.stream(Spliterators.splitIteratorUnknownSize(
70         iterator, Spliterator.ORDERED), false);
71     show("pathComponents", pathComponents);
72 }
73 }
```

java.util.stream.Stream 8

- static <T> Stream<T> of(T... values)
yields a stream whose elements are the given values.
- static <T> Stream<T> empty()
yields a stream with no elements.
- static <T> Stream<T> generate(Supplier<T> s)
yields an infinite stream whose elements are constructed by repeatedly invoking the function s.
- static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
- static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> f)
yields a stream whose elements are seed, f invoked on seed, f invoked on the preceding element, and so on. The first method yields an infinite stream. The stream of the second method comes to an end before the first element that doesn't fulfill the hasNext predicate.
- static <T> Stream<T> ofNullable(T t) 9
returns an empty stream if t is null or a stream containing t otherwise.

java.util.Spliterators 8

- static <T> Spliterator<T> spliteratorUnknownSize(Iterator<? extends T> iterator, int characteristics)
turns an iterator into a splittable iterator of unknown size with the given characteristics (a bit pattern containing constants such as Spliterator.ORDERED).

java.util.Arrays 1.2

- static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive) 8
yields a stream whose elements are the specified range of the array.