CORE 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

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

ISBN-13: 978-0-13-516631-4
ISBN-10: 0-13-516631-4

ScoutAutomatedPrintCode

Contents

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 charm 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 Philion, 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 Stelting, 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

CHAPTER 1

Streams

In this chapter

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`

```
package streams;

/**
 * @version 1.01 2018-05-01
 * @author Cay Horstmann
 */

import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;

public class CountLongWords
{
   public static void main(String[] args) throws IOException
   {
      var contents = Files.readStr(
         Paths.get("../gutenberg/alice30.txt"));
      List<String> words = List.of(contents.split("\\PL+"));

      long count = 0;
      for (String w : words)
      {
         if (w.length() > 12) count++;
      }
      System.out.println(count);

      count = words.stream().filter(w -> w.length() > 12).count();
      System.out.println(count);

      count = words.parallelStream().filter(w -> w.length() > 12).count();
      System.out.println(count);
   }
}
```

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`

```
package streams;

/**
 * @version 1.01 2018-05-01
 * @author Cay Horstmann
 */

import java.io.IOException;
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;
import java.util.regex.Pattern;
import java.util.stream.*;

public class CreatingStreams
{
   public static <T> void show(String title, Stream<T> stream)
   {
```

(Continues)

Listing 1.2 *(Continued)*

```java
      final int SIZE = 10;
      List<T> firstElements = stream
         .limit(SIZE + 1)
         .collect(Collectors.toList());
      System.out.print(title + ": ");
      for (int i = 0; i < firstElements.size(); i++)
      {
         if (i > 0) System.out.print(", ");
         if (i < SIZE) System.out.print(firstElements.get(i));
         else System.out.print("...");
      }
      System.out.println();
   }

   public static void main(String[] args) throws IOException
   {
      Path path = Paths.get("../gutenberg/alice30.txt");
      var contents = Files.readStr(path);

      Stream<String> words = Stream.of(contents.split("\\PL+"));
      show("words", words);
      Stream<String> song = Stream.of("gently", "down", "the", "stream");
      show("song", song);
      Stream<String> silence = Stream.empty();
      show("silence", silence);

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

      Stream<Double> randoms = Stream.generate(Math::random);
      show("randoms", randoms);

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

      Stream<String> wordsAnotherWay = Pattern.compile("\\PL+").splitAsStream(contents);
      show("wordsAnotherWay", wordsAnotherWay);

      try (Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8))
      {
         show("lines", lines);
      }

      Iterable<Path> iterable = FileSystems.getDefault().getRootDirectories();
      Stream<Path> rootDirectories = StreamSupport.stream(iterable.spliterator(), false);
      show("rootDirectories", rootDirectories);
```

```
      Iterator<Path> iterator = Paths.get("/usr/share/dict/words").iterator();
      Stream<Path> pathComponents = StreamSupport.stream(Spliterators.spliteratorUnknownSize(
         iterator, Spliterator.ORDERED), false);
      show("pathComponents", pathComponents);
   }
}
```

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.