ORACLE PRESS

Java™

# CORE JAVA

## Volume II: Advanced Features

FOURTEENTH EDITION

ORACLE

Cay S. Horstmann

This page intentionally left blank

# Core Java

## Volume II: Advanced Features

### Fourteenth Edition

Cay S. Horstmann

Pearson

Cover image: surangaw/stock.adobe.com
Figures 3.3, 4.8: Mozilla Foundation
Figures 4.1-4.5, 5.3, 5.4: Microsoft Corporation
Figure 4.6: USPS
Figures 5.6, 7.1-7.6, 9.7, 10.1, 10.3, 10.8, 10.10, 10.11, 10.14-10.16, 11.4, 11.5, 11.8-11.27, 11.29-11.34, 12.1-12.5, 12.7-12.10, 12.14-12.26, 12.28-12.30, 12.39, 12.44-12.46, 12.54-12.56, 12.60, 12.62: Oracle Corporation
Figure 9.3: HHD Software Ltd
Figure 12.51: Shao-Chun Wang/123RF
Figures 12.57-12.59: Pearson Education



Visit us on the Web: informit.com

Library of Congress Control Number: 2025945104



ISBN-13: 978-0-13-555869-0
ISBN-10: 0-13-555869-7

# Contents

# Preface

## To the Reader

The book you have in your hands is the second volume of the thirteenth edition of *Core Java*, fully updated for Java 25. 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.

With the explosive growth of the Java class library, a one-volume treatment of all the features of Java that serious programmers need to know is simply not possible. Hence, the book is broken up into two volumes.

This first volume concentrates on the fundamental concepts of the Java language:

- Object-oriented programming
- Reflection and proxies
- Interfaces and inner classes
- Exception handling
- Generic programming
- The collections framework
- Concurrency
- Annotations
- The Java platform module system

This second volume goes further into the most important libraries. For twelve editions, user interface programming was considered fundamental, but the time has come to recognize that it is no more, and to move it into the second volume. The volume includes detailed discussions of these topics:

- The Stream API
- File processing and regular expressions
- Databases
- XML processing
- Scripting and Compiling APIs
- Internationalization
- Network programming
- Graphical user interface design
- Graphics programming
- Foreign functions and memory

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 https://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.

## A Tour of 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 executing computations in parallel with multiple cores.

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 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.)

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 the scripting and compiler APIs that allow your program to call code in scripting languages such as JavaScript or Groovy, and to compile Java code.

**Chapter 9** takes up the Java security model, user authentication, and the cryptographic functions in the Java security library. You will learn about important features such as message and code signing, authorization and authentication, and encryption. We conclude with examples that use the AES and RSA encryption algorithms.

**Chapter 10** provides an introduction into GUI programming. I show how you can make windows, how to paint on them, how to draw with geometric shapes, how to format text in multiple fonts, and how to display images. Next, you'll see how to write code that responds to events, such as mouse clicks or key presses.

**Chapter 11** discusses the Swing GUI toolkit in great detail. The Swing toolkit allows you to build cross-platform graphical user interfaces. You'll learn all about the various kinds of buttons, text components, borders, sliders, list boxes, menus, and dialog boxes.

**Chapter 12** contains advanced Swing material, 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 13** takes up foreign functions, which let you call code written in other programming languages. 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. Or you may want to call a battle-tested library for machine learning or data analysis.

As always, all chapters have been completely revised for the latest version of Java. Outdated material has been removed, and the new APIs up to Java 25 are covered in detail.

## Conventions

As is common in many computer books, I 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, I warn you with a "caution" icon.

---

**Preview:** Preview features that are slated to become a part of the language or API in the future are labeled with this icon.

---

Java comes with a large programming library, or Application Programming Interface (API). When using an API call for the first time, I add a short summary description at the end of the section. These descriptions are a bit more informal but, hopefully, 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, as shown in the following example:

**`java.lang.IO`** **25**

- `println(Object obj)`
  Converts the object to a string and prints it on the console, followed by a line separator.

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

**Listing 1.1: `NotHelloWorld.java`**

```
void main() {
    System.out.println("We will not use 'Hello, World!'");
}
```

You can download the companion code from https://horstmann.com/corejava.

# 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.

My thanks go to my editor, Harry Misthos, and to Julie Nahil from Pearson for steering the book through the production process. I wrote the book using HTML and CSS, and Prince (`https://princexml.com`) turned it into PDF—a workflow that I highly recommend.

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 (Utah Valley University), Lance Andersen (Oracle), Gail Anderson (Anderson Software Group), Paul Anderson (Anderson Software Group), Alan Bateman (Oracle), Alec Beaton (IBM), Cliff Berg, Andrew Binstock (Oracle), Joshua Bloch, David Brown, Brian Burkhalter (Oracle), Corky Cartwright, Hillmer Chona, Frank Cohen (PushToTest), Chris Crane (devXsolution), Joe Darcy (Oracle), Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), Ahmad R. Elkomey, Hanno Embregts (Info Support), Robert Evans (Senior Staff, The Johns Hopkins University Applied Physics Lab), David Geary (Clarity Training), Jim Gish (Oracle), Brian Goetz (Oracle), Angela Gordon, Dan Gordon (Electric Cloud), Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Andrzej Grzesik, Marty Hall, Majid Hameed, Vincent Hardy (Adobe Systems), Dan Harkey (San Jose State University), Steve Haines, William Higgins (IBM), Marc Hoffmann (mtrail), Vladimir Ivanovic (PointBase), Jerry Jackson (CA Technologies), Josh Juneau, Heinz Kabutz (The Java Specialists' Newsletter, `https://javaspecialists.eu`), Stepan V. Kalinin (I-Teco/Servionica LTD), Tim Kimmet (Walmart), John Kostaras, Jerzy Krolak, Chris Laffra, Charlie Lai (Apple), Angelika Langer, Jeff Langr (Langr Software Solutions), Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Jason Lee (IBM), Gregory Longshore, Bob Lynch (Lynch Associates), Michael McMahon (Oracle), Rustam Mehmandarov, Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Maurice Naftalin, Mahesh Neelakanta (Florida Atlantic University), José Paumard (Oracle), Hao Pham, Paul Philion, Blake Ragsdell, Ylber Ramadani (Ryerson University), Stuart Reges (University of Arizona), Simon Ritter (Azul Systems), Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nice, France), Dr. Paul Sanghera (San Jose State University and Brooks College), Naoto Sato (Oracle), Paul Sevinc (Teamup AG), Devang Shah (Sun Microsystems), Yoshiki Shibata, Richard Slywczak (NASA/Glenn Research Center), Bradley A. Smith, Steven Stelting (Oracle), Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (StreamingEdge), Janet Traub, Henri Tremblay, Paul Tyma (consultant), Christian Ullenboom, Peter van der Linden, Joe Wang (Oracle), Sven Woltmann, Burt Walsh, Dan Xu (Oracle), and John Zavgren (Oracle).

Finally, a warm thank you to my coauthor of earlier editions, Gary Cornell, and to Greg Doench who was my editor for almost thirty years.

*Cay Horstmann*
*Düsseldorf, Germany*
*September 2025*

Chapter 1

# Streams

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 allows you to process sequences of values 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:

```
String contents =  Files.readString(Path.of("alice.txt")); // Read file into string
String[] words = contents.split("\\PL+"); // Split into words
```

The `split` method splits a string into parts, given a regular expression for the delimiters. (See Chapter 2 for more information about regular expressions. For now, just take it on faith that the `\PL+` expression works.)

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 = Stream.of(words)
    .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 12. 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` methods. The `filter` method transforms it, and `count` is the terminal operation.

In the next section, you will see how to create a stream. Three subsequent sections deal with intermediate operations. Then we turn to terminal operations.

**Listing 1.1:** `v2ch01/streams/CountLongWords.java`

```
package v2ch01.streams;

import module java.base;

class CountLongWords {
    void main() throws Exception {
        String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));
        String[] words = contents.split("\\PL+");

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

        count = Stream.of(words).filter(w -> w.length() > 12).count();
        IO.println(count);

        count = Stream.of(words).parallel().filter(w -> w.length() > 12).count();
        IO.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()`
  yield 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+"));
```

In the preceding example, we make a stream from the array that the `split` method returns. 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 parameters (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 for an example.

A number of methods in the Java API yield streams. For example, the String class has a lines method that yields a stream of the lines contained in the string:

```
Stream<String> greetings = "Hello\nGuten Tag\nBonjour".lines();
```

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 that the `try`-with-resources block is necessary to close the file.

It is becoming more common to offer data as streams in the Java API. For example, Java 21 added a static `availableLocales` method that yields a stream of `Locale` objects.

---

**Note:** To produce a finite stream, you can always add the elements to a list which you then turn into a stream. The `Stream.Builder` interface is slightly more efficient for that purpose:

```
Stream<Integer> digits(int n) {
    Stream.Builder<Integer> builder = Stream.builder();
    while (n != 0) {
        builder.add(n % 10);
        n = n / 10;
    }
    return builder.build();
}
```

---

To view the contents of one of the streams introduced in this section, use the `toList` method, which collects the stream's elements in a list. Like `count`, `toList` is a terminal operation. If the stream is infinite, first truncate it with the `limit` method:

```
IO.println(Stream.generate(Math::random).limit(10).toList());
```

---

**Note:** In order to turn an `Iterator`, or a general `Iterable` that is not a collection, into a stream, you need to use helper methods for *spliterators*. A spliterator is similar to an iterator, but it can be split so that each resulting spliterator yields a part of the elements. This splitting is used for processing elements in parallel—see Section 1.13.

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

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

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

```
StreamSupport.stream(iterable.spliterator(), 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:** `v2ch01/streams/CreatingStreams.java`

```
package v2ch01.streams;

import module java.base;

class CreatingStreams {
    void main() throws Exception {
        Path path = Path.of("gutenberg", "alice30.txt");
        String contents = Files.readString(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> greetings = "Hello\nGuten Tag\nBonjour".lines();
        show("greetings", greetings);

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

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

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

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

    <T> void show(String title, Stream<T> stream) {
        final int SIZE = 10;
        List<T> firstElements = stream.limit(SIZE + 1).toList();
        IO.print(title + ": ");
        for (int i = 0; i < firstElements.size(); i++) {
            if (i > 0) IO.print(", ");
            if (i < SIZE)
                IO.print(firstElements.get(i));
            else
                IO.print("...");
        }
        IO.println();
    }

}
```

***`java.util.stream.Stream<T>`*** **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)`
  yield 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.

**java.lang.String**

- String[] split(String regex) **1.4**
  finds all matches of the regular expression and splits the string by removing them. Returns an array with the fragments, but not a trailing empty string.
- static Stream<String> lines() **11**
  yields a stream whose elements are the lines of this string.

**java.util.regex.Pattern 1.4**

- Stream<String> splitAsStream(CharSequence input) **8**
  yields a stream whose elements are the parts of the input that are delimited by this pattern.

**java.nio.file.Files 7**

- static Stream<String> lines(Path path) **8**
- static Stream<String> lines(Path path, Charset cs) **8**
  yield a stream whose elements are the lines of the specified file, with the UTF-8 charset or the given charset.

**java.util.stream.StreamSupport 8**

- static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel)
  yields a stream containing the values produced by the given splittable iterator.

***java.lang.Iterable* 5.0**

- Spliterator<T> spliterator() **8**
  yields a splittable iterator for this Iterable. The default implementation does not split and does not report a size.

**java.util.Scanner** 5.0

- public Stream<String> tokens() 9
  yields a stream of strings returned by calling the next method of this scanner.

***java.util.function.Supplier<T>*** 8

- T get()
  supplies a value.

## 1.3. The filter, map, and flatMap Methods

A stream transformation produces a stream whose elements are derived from those of another stream. You have already seen the filter transformation that yields a new stream with those elements that match a certain condition. Here, we transform a stream of strings into another stream containing only long words:

```
List<String> wordList = . . .;
Stream<String> longWords = wordList.stream().filter(w -> w.length() > 12);
```

The parameter type of filter is Predicate<T>—that is, a function from T to boolean.

Often, you want to transform the values in a stream in some way. Use the map method and pass the function that carries out the transformation. For example, you can transform all words to lowercase like this:

```
Stream<String> lowercaseWords = words.stream().map(String::toLowerCase);
```

Here, we used map with a method reference. Often, you will use a lambda expression instead:

```
Stream<Character> firstCodeUnits = words.stream().map(s -> s.charAt(0));
```

The resulting stream contains the first code unit of each word.

When you use map, a function is applied to each element, yielding a new stream of the returned values. Now consider the situation where the returned values are themselves streams. The following method yields a stream of all grapheme clusters of a string.

```
public static Stream<String> graphemeClusters(String s) {
    return Stream.of(s.split("\\b{g}"));
}
```

For example, graphemeClusters("Ahoy 🏴‍☠️") is a stream of strings "A", "h", "o", "y", " ", and "🏴‍☠️". (Note that the flag consists of multiple char values.)

Now let's map the graphemeClusters method on a stream of strings:

```
List<String> wordList = List.of(. . ., "your", "boat", . . .);
Stream<Stream<String>> result = wordList.stream().map(w -> graphemeClusters(w));
```

You will get a stream of streams, like `[. . . ["y", "o", "u", "r"], ["b", "o", "a", "t"], . . .]`. To flatten it out to a single stream `[. . . "y", "o", "u", "r", "b", "o", "a", "t", . . .]`, use the `flatMap` method instead of `map`:

```
Stream<String> flatResult = words.stream().flatMap(w -> graphemeClusters(w));
    // Calls graphemeClusters on each word and flattens the results
```

Sometimes, it is inefficient to produce a stream for each result sequence. The `mapMulti` method offers an alternative. Instead of producing a stream of results, you generate the results and pass them to a collector—an object of a class implementing the functional interface `Consumer`. For each result, invoke the collector's `accept` method.

Let's do this with an example. The following loop iterates over the grapheme clusters of a string `s`:

```
BreakIterator iter = BreakIterator.getCharacterInstance();
. . .
iter.setText(s);
int start = iter.first();
int end = iter.next();
while (end != BreakIterator.DONE) {
    String gc = s.substring(start, end);
    start = end;
    end = iter.next();
    // Do something with gc
}
```

When calling `mapMulti`, you provide a function that is invoked with the stream element and the collector. In your function, pass your results to the collector.

```
Stream<String> result = words.stream().mapMulti((s, collector) -> {
    iter.setText(s);
    int start = iter.first();
    int end = iter.next();
    while (end != BreakIterator.DONE) {
        String gc = s.substring(start, end);
        start = end;
        end = iter.next();
        collector.accept(gc);
    }
});
```

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

- `Stream<T> filter(Predicate<? super T> predicate)`
  yields a stream containing the elements of this stream that fulfill the predicate.
- `<R> Stream<R> map(Function<? super T,? extends R> mapper)`
  yields a stream containing the results of applying `mapper` to the elements of this stream.

- <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
  yields a stream obtained by concatenating the results of applying mapper to the elements of this stream. (Note that each mapper result is a stream.)
- <R> Stream<R> mapMulti(BiConsumer<? super T,? super Consumer<R>> mapper) **16**
  For each stream element, the mapper is invoked, and all elements passed to the Consumer during invocation are added to the result stream.

## 1.4. Extracting Substreams and Combining Streams

The call *stream*.limit(n) returns a new stream that ends after n elements (or when the original stream ends if it is shorter). This method is particularly useful for cutting infinite streams down to size. For example,

```
Stream<Double> randoms = Stream.generate(Math::random).limit(100);
```

yields a stream with 100 random numbers.

The call *stream*.skip(n) does the exact opposite. It discards the first n elements. This is handy in our book reading example where, due to the way the split method works, the first element is an unwanted empty string. We can make it go away by calling skip:

```
Stream<String> words = Stream.of(contents.split("\\PL+")).skip(1);
```

The *stream*.takeWhile(*predicate*) call takes all elements from the stream while the predicate is true, and then stops.

For example, suppose we use the graphemeClusters method of the preceding section to split a string into characters, and we want to collect all initial digits. The takeWhile method can do this:

```
Stream<String> initialDigits = graphemeClusters(str).takeWhile(
    s -> "0123456789".contains(s));
```

The dropWhile method does the opposite, dropping elements while a condition is true and yielding a stream of all elements starting with the first one for which the condition was false. For example,

```
Stream<String> withoutInitialWhiteSpace = graphemeClusters(str).dropWhile(
    s -> s.strip().length() == 0);
```

You can concatenate two streams with the static concat method of the Stream class:

```
Stream<String> combined = Stream.concat(
    graphemeClusters("Hello"), graphemeClusters("World"));
    // Yields the stream ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]
```

Of course, the first stream should not be infinite—otherwise the second one wouldn't ever get a chance.

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

- `Stream<T> limit(long maxSize)`
  yields a stream with up to `maxSize` of the initial elements from this stream.
- `Stream<T> skip(long n)`
  yields a stream whose elements are all but the initial `n` elements of this stream.
- `Stream<T> takeWhile(Predicate<? super T> predicate)` **9**
  yields a stream whose elements are the initial elements of this stream that fulfill the predicate.
- `Stream<T> dropWhile(Predicate<? super T> predicate)` **9**
  yields a stream whose elements are the elements of this stream except for the initial ones that fulfill the predicate.
- `static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)`
  yields a stream whose elements are the elements of `a` followed by the elements of `b`.

## 1.5. Other Stream Transformations

The `distinct` method returns a stream that yields elements from the original stream, in the same order, except that duplicates are suppressed. The duplicates need not be adjacent.

```
Stream<String> uniqueWords
    = Stream.of("merrily", "merrily", "merrily", "gently").distinct();
    // Only one "merrily" is retained
```

For sorting a stream, there are several variations of the `sorted` method. One works for streams of `Comparable` elements, and another accepts a `Comparator`. Here, we sort strings so that the longest ones come first:

```
Stream<String> longestFirst
    = words.stream().sorted(Comparator.comparing(String::length).reversed());
```

As with all stream transformations, the `sorted` method yields a new stream whose elements are the elements of the original stream in sorted order.

Of course, you can sort a collection without using streams. The `sorted` method is useful when the sorting process is part of a stream pipeline.

Finally, the `peek` method yields another stream with the same elements as the original, but a function is invoked every time an element is retrieved. That is handy for debugging:

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
    .peek(e -> IO.println("Fetching " + e))
    .limit(20)
    .toArray();
```

When an element is actually accessed, a message is printed. This way you can verify that the infinite stream returned by `iterate` is processed lazily.

**Tip:** When you use a debugger to debug a stream computation, you can set a breakpoint in a method that is called from one of the transformations. With most IDEs, you can also set breakpoints in lambda expressions. If you just want to know what happens at a particular point in the stream pipeline, add

```
.peek(x -> {
    return;
})
```

and set a breakpoint on the second line.

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

- `Stream<T> distinct()`
  yields a stream of the distinct elements of this stream.
- `Stream<T> sorted()`
- `Stream<T> sorted(Comparator<? super T> comparator)`
  yield a stream whose elements are the elements of this stream in sorted order. The first method requires that the elements are instances of a class implementing `Comparable`.
- `Stream<T> peek(Consumer<? super T> action)`
  yields a stream with the same elements as this stream, passing each element to `action` as it is consumed.

## 1.6. Simple Reductions

Now that you have seen how to create and transform streams, we will finally get to the most important point—getting answers from the stream data. The methods covered in this section are called *reductions*. Reductions are *terminal operations*. They reduce the stream to a nonstream value that can be used in your program.

You have already seen a simple reduction: the `count` method that returns the number of elements of a stream.

Other simple reductions are `max` and `min` that return the largest or smallest value. There is a twist—these methods return an `Optional<T>` value that either wraps the answer or indicates that there is none (because the stream happened to be empty). In the olden days, it was common to return `null` in such a situation. But that can lead to null pointer exceptions when it happens in an incompletely tested program. The `Optional` type is a better way of indicating a missing return value. We discuss the `Optional` type in detail in the next section. Here is how you can get the maximum of a stream:

```
Optional<String> largest = words.max(String::compareToIgnoreCase);
IO.println("largest: " + largest.orElse(""));
```

The `findFirst` returns the first value in a nonempty collection. It is often useful when combined with `filter`. For example, here we find the first word that starts with the letter Q, if it exists:

```
Optional<String> startsWithQ
    = words.filter(s -> s.startsWith("Q")).findFirst();
```

If you are OK with any match, not just the first one, use the `findAny` method. This is effective when you parallelize the stream, since the stream can report any match that it finds instead of being constrained to the first one.

```
Optional<String> startsWithQ
    = words.parallel().filter(s -> s.startsWith("Q")).findAny();
```

If you just want to know if there is a match, use the terminal `anyMatch` operation with a predicate argument:

```
boolean aWordStartsWithQ
    = words.parallel().anyMatch(s -> s.startsWith("Q"));
```

There are methods `allMatch` and `noneMatch` that return `true` if all or no elements match a predicate. These methods also benefit from being run in parallel.

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

- `Optional<T> max(Comparator<? super T> comparator)`
- `Optional<T> min(Comparator<? super T> comparator)`
  yield a maximum or minimum element of this stream, using the ordering defined by the given comparator, or an empty `Optional` if this stream is empty. These are terminal operations.
- `Optional<T> findFirst()`
- `Optional<T> findAny()`
  yield the first, or any, element of this stream, or an empty `Optional` if this stream is empty. These are terminal operations.
- `boolean anyMatch(Predicate<? super T> predicate)`
- `boolean allMatch(Predicate<? super T> predicate)`
- `boolean noneMatch(Predicate<? super T> predicate)`
  return `true` if any, all, or none of the elements of this stream match the given predicate. These are terminal operations.

## 1.7. The Optional Type

An `Optional<T>` object is a wrapper for either an object of type `T` or no object. In the former case, we say that the value is *present*. The `Optional<T>` type is intended as a safer alternative for a reference of type `T` that either refers to an object or is `null`. But it is only safer if you use it right. The next three sections show you how.

### 1.7.1. Getting an Optional Value

The key to using `Optional` effectively is to use a method that either *produces an alternative* if the value is not present, or *consumes the value* only if it is present.

In this section, we look at the first strategy. Often, there is a default that you want to use when there is no match, perhaps the empty string:

```
String result = optionalString.orElse("");
    // The wrapped string, or "" if none
```

You can also invoke code to compute the default:

```
String result = optionalString.orElseGet(() -> System.getProperty("myapp.default"));
    // The function is only called when needed
```

Or you can throw an exception if there is no value:

```
String result = optionalString.orElseThrow(IllegalStateException::new);
    // Supply a method that yields an exception object
```

**java.util.Optional 8**

- `T orElse(T other)`
  yields the value of this `Optional`, or `other` if this `Optional` is empty.
- `T orElseGet(Supplier<? extends T> other)`
  yields the value of this `Optional`, or the result of invoking `other` if this `Optional` is empty.
- `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`
  yields the value of this `Optional`, or throws the result of invoking `exceptionSupplier` if this `Optional` is empty.

## 1.7.2. Consuming an Optional Value

In the preceding section, you saw how to produce an alternative if no value is present. The other strategy for working with optional values is to consume the value only if it is present.

The `ifPresent` method accepts a function. If the optional value exists, it is passed to that function. Otherwise, nothing happens.

```
optionalValue.ifPresent(v -> Process v);
```

For example, if you want to add the value to a set if it is present, call

```
optionalValue.ifPresent(v -> results.add(v));
```

or simply

```
optionalValue.ifPresent(results::add);
```

If you want to take one action if the `Optional` has a value and another action if it doesn't, use `ifPresentOrElse`:

```
optionalValue.ifPresentOrElse(
    v -> IO.println("Found " + v),
    () -> logger.warning("No match"));
```

**java.util.Optional** 8

- `void ifPresent(Consumer<? super T> action)`
  If this `Optional` is nonempty, passes its value to `action`.
- `void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)` 9
  If this `Optional` is nonempty, passes its value to `action`, else invokes `emptyAction`.

### 1.7.3. Pipelining Optional Values

In the preceding sections, you saw how to get a value out of an `Optional` object. Another useful strategy is to keep the `Optional` intact. You can transform the value inside an `Optional` by using the `map` method:

```
Optional<Path> transformed = optionalString.map(Path::of);
```

If `optionalString` is empty, then `transformed` is also empty.

**Note:** This `map` method is the analog of the `map` method of the `Stream` interface that you have seen in Section 1.3. Simply imagine an optional value as a stream of size zero or one. The result again has size zero or one, and in the latter case, the function has been applied.

Similarly, you can use the `filter` method to only consider `Optional` values that fulfill a certain property before or after transforming it. If the property is not fulfilled, the pipeline yields an empty result:

```
Optional<Path> transformed = optionalString
    .filter(s -> s.endsWith(".txt"))
    .map(Path::of);
```

You can substitute an alternative `Optional` for an empty `Optional` with the `or` method. The alternative is computed lazily.

```
Optional<String> result = optionalString.or(() -> // Supply an Optional
    alternatives.stream().findFirst());
```

If `optionalString` has a value, then `result` is `optionalString`. If not, the lambda expression is evaluated, and its result is used.

**java.util.Optional** 8

- `<U> Optional<U> map(Function<? super T,? extends U> mapper)`
  yields an `Optional` whose value is obtained by applying the given function to the value of this `Optional` if present, or an empty `Optional` otherwise.
- `Optional<T> filter(Predicate<? super T> predicate)`
  yields an `Optional` with the value of this `Optional` if it fulfills the given predicate, or an empty `Optional` otherwise.

- `Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)` **9**
  yields this `Optional` if it is nonempty, or the one produced by the supplier otherwise.

### 1.7.4. How Not to Work with Optional Values

If you don't use `Optional` values correctly, you have no benefit over the “something or `null`” approach of the past.

The `get` method gets the wrapped element of an `Optional` value if it exists, or throws a `NoSuchElementException` if it doesn’t. Therefore,

```
Optional<T> optionalValue = . . .;
optionalValue.get().someMethod()
```

is no safer than

```
T value = . . .;
value.someMethod();
```

The `isPresent` and `isEmpty` methods report whether or not an `Optional<T>` object has a value. But

```
if (optionalValue.isPresent()) optionalValue.get().someMethod();
```

is no easier than

```
if (value != null) value.someMethod();
```

**Tip:** If you find yourself using the `get`, `isPresent`, or `isEmpty` methods, try to rethink your approach and use one of the strategies of the preceding two sections.

**Note:** The `orElseThrow` method is a scarier-sounding synonym for the `get` method. When you call `optionalValue.orElseThrow().someMethod()`, it becomes explicit that an exception will occur if the `optionalValue` is empty. The hope is that programmers will only use `orElseThrow` when it is absolutely clear that this cannot happen.

Here are a few more tips for the proper use of the `Optional` type:

- A variable of type `Optional` should *never* be `null`.
- Don't use fields of type `Optional`. The cost is an additional object. Inside a class, use `null` for an absent field. To discourage `Optional` fields, the class is not serializable.
- Method parameters of type `Optional` are questionable. They make the call unpleasant in the common case where the requested value is present. Instead, consider two overloaded versions of the method, with and without the parameter. (On the other hand, returning an `Optional` is fine. It is the proper way to indicate that a function may not have a result.)
- Don't put `Optional` objects in a set, and don't use them as keys for a map. Collect the values instead.

**java.util.Optional 8**

- `T get()`
- `T orElseThrow()` **10**
  yield the value of this `Optional`, or throw a `NoSuchElementException` if it is empty.
- `boolean isEmpty()` **11**
- `boolean isPresent()`
  return `true` if this `Optional` is empty or not empty.

### 1.7.5. Creating Optional Values

So far, we have discussed how to consume an `Optional` object someone else created. If you want to write a method that creates an `Optional` object, there are several static methods for that purpose, including `Optional.of(result)` and `Optional.empty()`. For example,

```
public static Optional<Double> inverse(Double x) {
    return x == 0 ? Optional.empty() : Optional.of(1 / x);
}
```

The `ofNullable` method is intended as a bridge from possibly `null` values to optional values. `Optional.ofNullable(obj)` returns `Optional.of(obj)` if `obj` is not `null` and `Optional.empty()` otherwise.

**java.util.Optional 8**

- `static <T> Optional<T> of(T value)`
- `static <T> Optional<T> ofNullable(T value)`
  yield an `Optional` with the given value. If `value` is `null`, the first method throws a `NullPointerException` and the second method yields an empty `Optional`.
- `static <T> Optional<T> empty()`
  yields an empty `Optional`.

### 1.7.6. Composing Optional Value Functions with `flatMap`

Suppose you have a method `f` yielding an `Optional<T>`, and the target type `T` has a method `g` yielding an `Optional<U>`. If they were normal methods, you could compose them by calling `s.f().g()`. But that composition doesn't work since `s.f()` has type `Optional<T>`, not `T`. Instead, call

```
Optional<U> result = s.f().flatMap(T::g);
```

If `s.f()` is present, then `g` is applied to it. Otherwise, an empty `Optional<U>` is returned.

Clearly, you can repeat that process if you have more methods or lambdas that yield `Optional` values. You can then build a pipeline of steps, simply by chaining calls to `flatMap`, that will succeed only when all parts do.

For example, consider the safe `inverse` method of the preceding section. Suppose we also have a safe square root:

```
public static Optional<Double> squareRoot(Double x) {
    return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
}
```

Then you can compute the square root of the `inverse` as

```
Optional<Double> result = inverse(arg).flatMap(x -> squareRoot(x));
```

We can make such a chain look a little more regular by starting out with an `Optional`:

```
Optional<Double> result
    = Optional.of(arg).flatMap(x -> inverse(x)).flatMap(x -> squareRoot(x));
```

It looks even neater with method expressions:

```
Optional<Double> result
    = Optional.of(arg).flatMap(this::inverse).flatMap(this::squareRoot);
```

If either the `inverse` method or the `squareRoot` returns `Optional.empty()`, the result is empty.

**Note:** In this example, taken from Listing 1.3, the `inverse` and `squareRoot` methods are instance methods. Hence the method expressions start with `this::` . Had these been static methods, they would be prefixed by *ClassName*`::`.

**Note:** You have already seen a `flatMap` method in the `Stream` interface (see Section 1.3). That method maps a stream-yielding method to all elements of a stream, and then flattens out the resulting stream of streams. The `Optional.flatMap` method works in the same way. By mapping an optional-yielding method to an optional, you get an optional of optional, which is then flattened out.

**java.util.Optional** 8

- `<U> Optional<U> flatMap(Function<? super T,? extends Optional<? extends U>> mapper)`
  yields the result of applying `mapper` to the value in this `Optional` if present, or an empty optional otherwise.

### 1.7.7. Turning an Optional into a Stream

The `stream` method turns an `Optional<T>` into a `Stream<T>` with zero or one element. Sure, why not, but why would you ever want that?

This becomes useful with methods that return an `Optional` result. Suppose you have a stream of user IDs and a method

```
Optional<User> lookup(String id)
```

How do you get a stream of users, skipping those IDs that are invalid?

Of course, you can filter out the invalid IDs and then apply `get` to the remaining ones:

```
Stream<String> ids = . . .;
Stream<User> users = ids.map(Users::lookup)
    .filter(Optional::isPresent)
    .map(Optional::get);
```

But that uses the `isPresent` and `get` methods that we warned about. It is more elegant to call

```
Stream<User> users = ids.map(Users::lookup)
    .flatMap(Optional::stream);
```

Each call to `stream` returns a stream with zero or one element. The `flatMap` method combines them all. That means the nonexistent users are simply dropped.

---

**Note:** In this section, we consider the happy scenario in which we have a method that returns an `Optional` value. These days, many methods return `null` when there is no valid result. Suppose `Users.classicLookup(id)` returns a `User` object or `null`, not an `Optional<User>`. Then you can of course filter out the `null` values:

```
Stream<User> users = ids.map(Users::classicLookup)
    .filter(Objects::nonNull);
```

But if you prefer the `flatMap` approach, you can use

```
Stream<User> users = ids.flatMap(
    id -> Stream.ofNullable(Users.classicLookup(id)));
```

or

```
Stream<User> users = ids.map(Users::classicLookup)
    .flatMap(Stream::ofNullable);
```

The call `Stream.ofNullable(obj)` yields an empty stream if `obj` is `null` or a stream just containing `obj` otherwise.

---

The example program in Listing 1.3 demonstrates the `Optional` API.

**Listing 1.3:** `v2ch01/optional/OptionalDemo.java`

```
package v2ch01.optional;

import module java.base;

class OptionalDemo {
    void main() throws Exception {
        String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));
        List<String> wordList = List.of(contents.split("\\PL+"));

        Optional<String> optionalValue = wordList
```

```
                .stream()
                .filter(s -> s.contains("fred"))
                .findFirst();
        IO.println(optionalValue.orElse("No word") + " contains fred");

        Optional<String> optionalString = Optional.empty();
        String result = optionalString.orElse("N/A");
        IO.println("result: " + result);
        result = optionalString.orElseGet(() -> Locale.getDefault().getDisplayName());
        IO.println("result: " + result);
        try {
            result = optionalString.orElseThrow(IllegalStateException::new);
            IO.println("result: " + result);
        }
        catch (Throwable t) {
            t.printStackTrace(System.out);
        }

        optionalValue = wordList.stream().filter(s -> s.contains("red")).findFirst();
        optionalValue.ifPresent(s -> IO.println(s + " contains red"));

        var results = new HashSet<String>();
        optionalValue.ifPresent(results::add);
        Optional<Boolean> added = optionalValue.map(results::add);
        IO.println(added);

        IO.println(inverse(4.0).flatMap(this::squareRoot));
        IO.println(inverse(-1.0).flatMap(this::squareRoot));
        IO.println(inverse(0.0).flatMap(this::squareRoot));
        Optional<Double> result2 = Optional.of(-4.0)
                .flatMap(this::inverse)
                .flatMap(this::squareRoot);
        IO.println(result2);
    }

    Optional<Double> inverse(Double x) {
        return x == 0 ? Optional.empty() : Optional.of(1 / x);
    }

    Optional<Double> squareRoot(Double x) {
        return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
    }
}
```

**`java.util.Optional`** 8

- `<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)` 9
  yields the result of applying `mapper` to the value of this `Optional`, or an empty `Optional` if this `Optional` is empty.

## 1.8. Collecting Results

When you are done with a stream, you will often want to look at the results. You can call the `iterator` method, which yields an old-fashioned iterator that you can use to visit the elements.

Alternatively, you can call the `forEach` method to apply a function to each element:

```
stream.forEach(IO::println);
```

On a parallel stream, the `forEach` method traverses elements in arbitrary order. If you want to process them in stream order, call `forEachOrdered` instead. Of course, you might then give up some or all of the benefits of parallelism.

But more often than not, you will want to collect the result in a data structure. You have already seen the `toList` method that yields a list of the stream elements.

Call `toArray` to get an array of the stream elements.

Since it is not possible to create a generic array at runtime, the expression `stream.toArray()` returns an `Object[]` array. If you want an array of the correct type, pass in the array constructor:

```
String[] result = stream.toArray(String[]::new);
    // stream.toArray() has type Object[]
```

To collect results into other data structures, the `Stream` interface has a generic `collect` method. You can concatenate the results in a string, store them in any collection, and insert them into maps. By implementing your own collector, you can consume the stream data in arbitrary ways. You will see the details in the following sections.

***java.util.stream.BaseStream*** **8**

- `Iterator<T> iterator()`
  yields an iterator for obtaining the elements of this stream. This is a terminal operation.

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

- `List<T> toList()` **16**
  yields a list of the elements in this stream.
- `void forEach(Consumer<? super T> action)`
  invokes `action` on each element of the stream.
- `Object[] toArray()`
- `<A> A[] toArray(IntFunction<A[]> generator)`
  yield an array of objects, or of type `A` when passed a constructor reference `A[]::new`. These are terminal operations.
- `<R,A> R collect(Collector<? super T,A,R> collector)`
  collects the elements in this stream, using the given collector. The `Collectors` class has factory methods for many collectors.

## 1.9. Collectors

For collecting stream elements to another target, there is a convenient `collect` method that takes an instance of the `Collector` interface. A *collector* is an object that accumulates elements and produces a result. The `Collectors` class provides a large number of factory methods for common collectors.

Suppose you want to collect all strings in a stream by concatenating them. You can call

```
String result = stream.collect(Collectors.joining());
```

If you want a delimiter between elements, pass it to the `joining` method:

```
String result = stream.collect(Collectors.joining(", "));
```

If your stream contains objects other than strings, you need to first convert them to strings, like this:

```
String result = stream.map(Object::toString).collect(Collectors.joining(", "));
```

If you want to reduce the stream results to a sum, count, average, maximum, or minimum, use one of the `summarizing(Int|Long|Double)` methods. These methods take a function that maps the stream objects to numbers and yield a result of type `(Int|Long|Double)SummaryStatistics`, simultaneously computing the sum, count, average, maximum, and minimum.

```
IntSummaryStatistics summary = stream.collect(
    Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

Note that you are not calling the `joining` and `summarizingInt` methods on a stream. These are static methods of the `Collectors` class. They yield instances of `Collector` which you pass to the `collect` method of the `Stream` interface.

The `joining` collector yields a string, and the summarizing collectors yield four numbers. However, most collectors collect values in data structures, as you will see in the following sections.

The example program in Listing 1.5 shows how to collect elements from a stream.

**Listing 1.4: `v2ch01/collecting/CollectingResults.java`**

```
package v2ch01.collecting;

import module java.base;

class CollectingResults {
    void main() throws Exception {
        // Turning a stream into an iterator
        Iterator<Integer> iter = Stream.iterate(0, n -> n + 1).limit(10).iterator();
        while (iter.hasNext())
            IO.println(iter.next());

        Object[] numbers = Stream.iterate(0, n -> n + 1).limit(10).toArray();
        IO.println("Object array: " + numbers + " " + Arrays.toString(numbers));
        // Note it's an Object[] array

        try {
            var number = (Integer) numbers[0]; // OK
            IO.println("number: " + number);
            IO.println("The following statement throws an exception:");
```

```
            var _ = (Integer[]) numbers; // Throws exception
        }
        catch (ClassCastException e) {
            IO.println(e);
        }

        Integer[] numbers3 = Stream.iterate(0, n -> n + 1).limit(10).toArray(Integer[]::new);
        IO.println("Integer array: " + numbers3 + " " + Arrays.toString(numbers3));
        // Note it's an Integer[] array

        String result = noVowels().limit(10).collect(Collectors.joining());
        IO.println("Joining: " + result);
        result = noVowels().limit(10).collect(Collectors.joining(", "));
        IO.println("Joining with commas: " + result);

        IntSummaryStatistics summary = noVowels()
                .collect(Collectors.summarizingInt(String::length));
        double averageWordLength = summary.getAverage();
        double maxWordLength = summary.getMax();
        IO.println("Average word length: " + averageWordLength);
        IO.println("Max word length: " + maxWordLength);
        IO.println("forEach:");
        noVowels().limit(10).forEach(IO::println);
    }

    Stream<String> noVowels() throws IOException {
        String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));
        Stream<String> words = Stream.of(contents.split("\\PL+"));
        return words.map(s -> s.replaceAll("[aeiouAEIOU]", ""));
    }

    <T> void show(String label, Set<T> set) {
        IO.print(label + ": " + set.getClass().getName());
        IO.println("["
            + set.stream().limit(10).map(Object::toString).collect(Collectors.joining(", "))
            + "]");
    }
}
```

**java.util.stream.Collectors 8**

- `static Collector<CharSequence,?,String> joining()`
- `static Collector<CharSequence,?,String> joining(CharSequence delimiter)`
- `static Collector<CharSequence,?,String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`
  yield a collector that joins strings. The delimiter is placed between strings, and the prefix and suffix before the first and after the last string. When not specified, these are empty.
- `static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)`
- `static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)`
- `static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper)`
  yield collectors that produce an `(Int|Long|Double)SummaryStatistics` object, from which you can obtain the count, sum, average, maximum, and minimum of the results of applying `mapper` to each element.

**java.util.IntSummaryStatistics** 8

**java.util.LongSummaryStatistics** 8

**java.util.DoubleSummaryStatistics** 8

- `long getCount()`
  yields the count of the summarized elements.
- *xxx* `getSum()`
- `double getAverage()`
  yield the sum or average of the summarized elements, or zero if there are no elements.
- *xxx* `getMax()`
- *xxx* `getMin()`
  yield the maximum or minimum of the summarized elements, or `(Integer|Long|Double).(MAX|MIN)_VALUE` if there are no elements.

### 1.9.1. Collecting into Collections

Before the `toList` method was added in Java 16, you had to use the collector produced by `Collectors.toList()`:

```
List<String> result = stream.collect(Collectors.toList());
```

Similarly, here is how you can collect stream elements into a set:

```
Set<String> result = stream.collect(Collectors.toSet());
```

These calls give you a list or set, but you cannot make any further assumptions. The collection might not be mutable, serializable, or thread-safe. If you want to control which kind of collection you get, use the following call instead:

```
TreeSet<String> result = stream.collect(Collectors.toCollection(TreeSet::new));
```

The example program in Listing 1.5 shows how to place stream elements into a collection.

**Listing 1.5:** `v2ch01/collecting/CollectingResults.java`

```
package v2ch01.collecting;

import module java.base;

class CollectingResults {
    void main() throws Exception {
        // Turning a stream into an iterator
        Iterator<Integer> iter = Stream.iterate(0, n -> n + 1).limit(10).iterator();
        while (iter.hasNext())
            IO.println(iter.next());

        Object[] numbers = Stream.iterate(0, n -> n + 1).limit(10).toArray();
        IO.println("Object array: " + numbers + " " + Arrays.toString(numbers));
```

```
        // Note it's an Object[] array

        try {
            var number = (Integer) numbers[0]; // OK
            IO.println("number: " + number);
            IO.println("The following statement throws an exception:");
            var _ = (Integer[]) numbers; // Throws exception
        }
        catch (ClassCastException e) {
            IO.println(e);
        }

        Integer[] numbers3 = Stream.iterate(0, n -> n + 1).limit(10).toArray(Integer[]::new);
        IO.println("Integer array: " + numbers3 + " " + Arrays.toString(numbers3));
        // Note it's an Integer[] array

        String result = noVowels().limit(10).collect(Collectors.joining());
        IO.println("Joining: " + result);
        result = noVowels().limit(10).collect(Collectors.joining(", "));
        IO.println("Joining with commas: " + result);

        IntSummaryStatistics summary = noVowels()
                .collect(Collectors.summarizingInt(String::length));
        double averageWordLength = summary.getAverage();
        double maxWordLength = summary.getMax();
        IO.println("Average word length: " + averageWordLength);
        IO.println("Max word length: " + maxWordLength);
        IO.println("forEach:");
        noVowels().limit(10).forEach(IO::println);
    }

    Stream<String> noVowels() throws IOException {
        String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));
        Stream<String> words = Stream.of(contents.split("\\PL+"));
        return words.map(s -> s.replaceAll("[aeiouAEIOU]", ""));
    }

    <T> void show(String label, Set<T> set) {
        IO.print(label + ": " + set.getClass().getName());
        IO.println("["
            + set.stream().limit(10).map(Object::toString).collect(Collectors.joining(", "))
            + "]");
    }
}
```

**java.util.stream.Collectors 8**

- `static <T> Collector<T,?,List<T>> toList()`
- `static <T> Collector<T,?,List<T>> toUnmodifiableList()` **10**
- `static <T> Collector<T,?,Set<T>> toSet()`
- `static <T> Collector<T,?,Set<T>> toUnmodifiableSet()` **10**

  yield collectors that collect elements in a list or set.

- `static <T,C extends Collection<T>> Collector<T,?,C> toCollection(Supplier<C> collectionFactory)`
  yields a collector that collects elements into an arbitrary collection. Pass a constructor reference such as `TreeSet::new`.

### 1.9.2. Collecting into Maps

Suppose you have a `Stream<Person>` and want to collect the elements into a map so that later you can look up people by their ID. Call `Collectors.toMap` with two functions that produce the map's keys and values. For example,

```
public record Person(int id, String name) {}
. . .
Map<Integer, String> idToName = people.collect(
    Collectors.toMap(Person::id, Person::name));
```

In the common case when the values should be the actual elements, use `Function.identity()` for the second function.

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(Person::id, Function.identity()));
```

If there is more than one element with the same key, there is a conflict, and the collector will throw an `IllegalStateException`. You can override that behavior by supplying a third function that resolves the conflict and determines the value for the key, given the existing and the new value. Your function could return the existing value, the new value, or a combination of them.

Here, we construct a map that contains, for each language in the available locales, as key its name in your default locale (such as `"German"`), and as value its localized name (such as `"Deutsch"`).

```
Map<String, String> languageNames = Locale.availableLocales().collect(
    Collectors.toMap(
        Locale::getDisplayLanguage,
        loc -> loc.getDisplayLanguage(loc),
        (existingValue, newValue) -> existingValue));
```

We don't care that the same language might occur twice (for example, German in Germany and in Switzerland), so we just keep the first entry.

---

**Note:** In this chapter, I use the `Locale` class as a source of an interesting data set. See Chapter 7 for more information on working with locales.

---

Now suppose we want to know all languages in a given country. Then we need a `Map<String, Set<String>>`. For example, the value for `"Switzerland"` is the set `[French, German, Italian]`. At first, we store a singleton set for each language. Whenever a new language is found for a given country, we form the union of the existing and the new set.

```
Map<String, Set<String>> countryLanguageSets = Locale.availableLocales().collect(
    Collectors.toMap(
        Locale::getDisplayCountry,
        l -> Collections.singleton(l.getDisplayLanguage()),
        (a, b) -> { // Union of a and b
            var union = new HashSet<String>(a);
            union.addAll(b);
            return union;
        }));
```

You will see a simpler way of obtaining this map in the next section.

If you want a TreeMap, supply the constructor as the fourth argument. You must provide a merge function. Here is one of the examples from the beginning of the section, now yielding a TreeMap:

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(
        Person::id,
        Function.identity(),
        (existingValue, newValue) -> { throw new IllegalStateException(); },
        TreeMap::new));
```

**Note:** For each of the toMap methods, there is an equivalent toConcurrentMap method that yields a concurrent map. A single concurrent map is used in the parallel collection process. When used with a parallel stream, a shared map is more efficient than merging maps. Note that elements are no longer collected in stream order, but that doesn't usually make a difference.

The program in Listing 1.6 gives examples of collecting stream results into maps.

**Listing 1.6:** **v2ch01/collecting/CollectingIntoMaps.java**

```
package v2ch01.collecting;

import module java.base;

record Person(int id, String name) {}

class CollectingIntoMaps {
    void main() throws Exception {
        Map<Integer, String> idToName = people()
                .collect(Collectors.toMap(Person::id, Person::name));
        IO.println("idToName: " + idToName);

        Map<Integer, Person> idToPerson = people()
                .collect(Collectors.toMap(Person::id, Function.identity()));
        IO.println("idToPerson: " + idToPerson.getClass().getName() + idToPerson);

        idToPerson = people().collect(
                Collectors.toMap(Person::id, Function.identity(), (_, _) -> {
                    throw new IllegalStateException();
```

```
                }, TreeMap::new));
        IO.println("idToPerson: " + idToPerson.getClass().getName() + idToPerson);

        Map<String, String> languageNames = Locale.availableLocales()
                .collect(Collectors.toMap(Locale::getDisplayLanguage,
                        l -> l.getDisplayLanguage(l),
                        (existingValue, _) -> existingValue));
        IO.println("languageNames: " + languageNames);

        Map<String, Set<String>> countryLanguageSets = Locale.availableLocales()
                .collect(Collectors.toMap(Locale::getDisplayCountry,
                        l -> Set.of(l.getDisplayLanguage()), (a, b) -> { // union of a and b
                            Set<String> union = new HashSet<>(a);
                            union.addAll(b);
                            return union;
                        }));
        IO.println("countryLanguageSets: " + countryLanguageSets);
    }

    static Stream<Person> people() {
        return Stream.of(new Person(1001, "Peter"), new Person(1002, "Paul"),
                new Person(1003, "Mary"));
    }
}
```

**`java.util.stream.Collectors`** 8

- `static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)`
- `static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)`
- `static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)`
- `static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)` **10**
- `static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)` **10**
- `static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)`
- `static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)`
- `static <T,K,U,M extends ConcurrentMap<K,U>> Collector<T,?,M> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)`

  yield a collector that produces a map, unmodifiable map, or concurrent map. The `keyMapper` and `valueMapper` functions are applied to each collected element, yielding a key/value entry of the resulting map. By default, an `IllegalStateException` is thrown when two elements give rise to the same key. You can instead supply a `mergeFunction` that merges values with the same key. By default, the result is a `HashMap` or `ConcurrentHashMap`. You can instead supply a `mapSupplier` that yields the desired map instance.

### 1.9.3. Grouping and Partitioning

In the preceding section, you saw how to collect all languages in a given country. But the process was a bit tedious. You had to generate a singleton set for each map value and then specify how to merge the existing and new values. Forming groups of values with the same characteristic is very common, so the `groupingBy` method supports it directly.

Let’s look at the problem of grouping locales by country. First, form this map:

```
Map<String, List<Locale>> countryToLocales = Locale.availableLocales().collect(
    Collectors.groupingBy(Locale::getCountry));
```

The function `Locale::getCountry` is the *classifier function* of the grouping. You can now look up all locales for a given country code, for example

```
List<Locale> swissLocales = countryToLocales.get("CH");
    // Yields locales de_CH, fr_CH, it_CH, and maybe more
```

---

**Note:** A quick refresher on locales: Each locale has a language code (such as `en` for English) and a country code (such as `US` for the United States). The locale `en_US` describes English in the United States, and `en_IE` is English in Ireland. Some countries have multiple locales. For example, `ga_IE` is Gaelic in Ireland, and, as the preceding example shows, the JDK knows at least three locales in Switzerland.

---

When the classifier function is a predicate function (that is, a function returning a `boolean` value), the stream elements are partitioned into two lists: those where the function returns `true` and the complement. In this case, it is more efficient to use `partitioningBy` instead of `groupingBy`. For example, here we split all locales into those that use English and all others:

```
Map<Boolean, List<Locale>> englishAndOtherLocales = Locale.availableLocales().collect(
    Collectors.partitioningBy(l -> l.getLanguage().equals("en")));
List<Locale> englishLocales = englishAndOtherLocales.get(true);
```

---

**Note:** If you call the `groupingByConcurrent` method, you get a concurrent map that, when used with a parallel stream, is concurrently populated. This is entirely analogous to the `toConcurrentMap` method.

---

**`java.util.stream.Collectors`** 8

- `static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)`
- `static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier)`

  yield a collector that produces a map or concurrent map whose keys are the results of applying `classifier` to all collected elements, and whose values are lists of elements with the same key.

- `static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)` yields a collector that produces a map whose keys are `true`/`false`, and whose values are lists of the elements that fulfill/do not fulfill the predicate.

### 1.9.4. Downstream Collectors

The `groupingBy` method yields a map whose values are lists. If you want to process those lists in some way, supply a *downstream collector*. For example, if you want sets instead of lists, you can use the `Collectors.toSet` collector that you saw in the preceding section:

```
Map<String, Set<Locale>> countryToLocaleSet = Locale.availableLocales().collect(
    groupingBy(Locale::getCountry, toSet()));
```

**Note:** In this example, as well as the remaining examples of this section, I assume a static import of `java.util.stream.Collectors.*` to make the expressions easier to read.

You can also apply `groupingBy` twice:

```
Map<String, Map<String, List<Locale>>> countryAndLanguageToLocale =
    Locale.availableLocales().collect(
        groupingBy(Locale::getCountry,
            groupingBy(Locale::getLanguage)));
```

Then `countryAndLanguageToLocale.get("IN").get("hi")` is a list of the Hindi locales in India. (There are several variants.)

Several collectors are provided for reducing collected elements to numbers:

- `counting` produces a count of the collected elements. For example,

  ```
  Map<String, Long> countryToLocaleCounts = Locale.availableLocales().collect(
      groupingBy(Locale::getCountry, counting()));
  ```

  counts how many locales there are for each country.
- `summing(Int|Long|Double)` and `averaging(Int|Long|Double)` apply a provided function to the downstream elements and produce the sum or average of the function's results. For example,

  ```
  public record City(String name, String state, int population) {}
  . . .
  Map<String, Integer> stateToCityPopulation = cities.collect(
      groupingBy(City::state, averagingInt(City::population)));
  ```

  computes the average of populations per state in a stream of cities.
- `maxBy` and `minBy` take a comparator and produce maximum and minimum of the downstream elements. For example,

```
Map<String, Optional<City>> stateToLargestCity = cities.collect(
    groupingBy(City::state,
        maxBy(Comparator.comparing(City::population))));
```

produces the largest city per state.

The collectingAndThen collector adds a final processing step behind a collector. For example, if you want to know how many distinct results there are, collect them into a set and then compute the size:

```
Map<Character, Integer> stringCountsByStartingLetter = strings.collect(
    groupingBy(s -> s.charAt(0),
        collectingAndThen(toSet(), Set::size)));
```

The mapping collector does the opposite. It applies a function to each collected element and passes the results to a downstream collector.

```
Map<Character, Set<Integer>> stringLengthsByStartingLetter = strings.collect(
    groupingBy(s -> s.charAt(0),
        mapping(String::length, toSet())));
```

Here, we group strings by their first character. Within each group, we produce the lengths and collect them in a set.

The mapping method also yields a nicer solution to a problem from the preceding section—gathering a set of all languages in a country.

```
Map<String, Set<String>> countryToLanguages = Locale.availableLocales().collect(
    groupingBy(Locale::getDisplayCountry,
        mapping(Locale::getDisplayLanguage,
            toSet())));
```

There is a flatMapping method as well, for use with functions that return streams.

If the grouping or mapping function has return type int, long, or double, you can collect elements into a summary statistics object, as discussed in Section 1.8. For example,

```
Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities.collect(
    groupingBy(City::state,
        summarizingInt(City::population)));
```

Then you can get the sum, count, average, minimum, and maximum of the function values from the summary statistics objects of each group.

The filtering collector applies a filter to each group, for example:

```
Map<String, Set<City>> largeCitiesByState
    = cities.collect(
        groupingBy(City::state,
            filtering(c -> c.population() > 500000,
                toSet()))); // States without large cities have empty sets
```

Finally, you can use the `teeing` collector to branch into two downstream collections. This is useful whenever you need to compute more than one result from a stream. Suppose you want to collect city names and also compute their average population. You can't read a stream twice, but `teeing` lets you carry out two computations. Specify two downstream collectors and a function that combines the results.

```
record Pair<S, T>(S first, T second) {}
Pair<List<String>, Double> result = cities.filter(c -> c.state().equals("NV"))
    .collect(teeing(
        mapping(City::name, toList()), // First downstream collector
        averagingDouble(City::population), // Second downstream collector
        (list, avg) -> new Pair(list,  avg))); // Combining function
```

**Note:** There are also three versions of a `reducing` method that apply general reductions, as described in the next section.

Composing collectors is powerful, but it can lead to very convoluted expressions. The best use is with `groupingBy` or `partitioningBy` to process the “downstream” map values. Otherwise, simply apply methods such as `map`, `reduce`, `count`, `max`, or `min` directly on streams.

The example program in Listing 1.7 demonstrates downstream collectors.

**Listing 1.7:** `v2ch01/collecting/DownstreamCollectors.java`

```
package v2ch01.collecting;

import static java.util.stream.Collectors.*;

import module java.base;

record City(String name, String state, int population) {}

class DownstreamCollectors {

    void main() throws Exception {
        Map<String, Set<Locale>> countryToLocaleSet = Locale.availableLocales()
                .collect(groupingBy(Locale::getCountry, toSet()));
        IO.println("countryToLocaleSet: " + countryToLocaleSet);

        Map<String, Map<String, List<Locale>>> countryAndLanguageToLocale = Locale
                .availableLocales()
                .collect(groupingBy(Locale::getCountry, groupingBy(Locale::getLanguage)));
        IO.println(
            "Hindi locales in India: " + countryAndLanguageToLocale.get("IN").get("hi"));

        Map<String, Long> countryToLocaleCounts = Locale.availableLocales()
                .collect(groupingBy(Locale::getCountry, counting()));
        IO.println("countryToLocaleCounts: " + countryToLocaleCounts);

        Stream<City> cities = readCities("v2ch01/cities.txt");
        Map<String, Integer> stateToCityPopulation = cities
```

```
                .collect(groupingBy(City::state, summingInt(City::population)));
        IO.println("stateToCityPopulation: " + stateToCityPopulation);

        cities = readCities("v2ch01/cities.txt");
        Map<String, Optional<String>> stateToLongestCityName = cities.collect(groupingBy(
            City::state, mapping(City::name, maxBy(Comparator.comparing(String::length)))));
        IO.println("stateToLongestCityName: " + stateToLongestCityName);

        Map<String, Set<String>> countryToLanguages = Locale.availableLocales()
                .collect(groupingBy(Locale::getDisplayCountry,
                    mapping(Locale::getDisplayLanguage, toSet())));
        IO.println("countryToLanguages: " + countryToLanguages);

        cities = readCities("v2ch01/cities.txt");
        Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities
                .collect(groupingBy(City::state, summarizingInt(City::population)));
        IO.println(stateToCityPopulationSummary.get("NY"));

        cities = readCities("v2ch01/cities.txt");
        Map<String, String> stateToCityNames = cities.collect(groupingBy(City::state,
            reducing("", City::name, (s, t) -> s.length() == 0 ? t : s + ", " + t)));

        cities = readCities("v2ch01/cities.txt");
        stateToCityNames = cities
                .collect(groupingBy(City::state, mapping(City::name, joining(", "))));
        IO.println("stateToCityNames: " + stateToCityNames);

        cities = readCities("v2ch01/cities.txt");
        record Pair<S, T>(S first, T second) {
        }
        Pair<List<String>, Double> result = cities.filter(c -> c.state().equals("NV")).collect(
            teeing(mapping(City::name, toList()), averagingDouble(City::population),
                (names, avg) -> new Pair<>(names, avg)));
        IO.println(result);
    }

    Stream<City> readCities(String filename) throws IOException {
        return Files.lines(Path.of(filename)).map(l -> l.split(", "))
                .map(a -> new City(a[0], a[1], Integer.parseInt(a[2])));
    }
}
```

**`java.util.stream.Collectors`** 8

- `public static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`
  yields a collector that produces a map. The keys are the results of applying `classifier` to all collected elements. The values are the results of collecting elements with the same key, using the downstream collector.
- `static <T> Collector<T,?,Long> counting()`
  yields a collector that counts the collected elements.

- static <T> Collector<T,?,Integer> summingInt(ToIntFunction<? super T> mapper)
- static <T> Collector<T,?,Long> summingLong(ToLongFunction<? super T> mapper)
- static <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper)
  yield a collector that computes the sum of the results of applying mapper to the collected elements.
- static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator)
- static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator)
  yield a collector that computes the maximum or minimum of the collected elements, using the ordering specified by comparator.
- static <T,A,R,RR> Collector<T,A,RR> collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)
  yields a collector that sends elements to the downstream collector and then applies the finisher function to its result.
- static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)
  yields a collector that calls mapper on each element and passes the results to the downstream collector.
- static <T,U,A,R> Collector<T,?,R> flatMapping(Function<? super T,? extends Stream<? extends U>> mapper, Collector<? super U,A,R> downstream)
  yields a collector that calls mapper on each element and passes the elements of the results to the downstream collector.
- static <T,A,R> Collector<T,?,R> filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream)
  yields a collector that passes the elements fulfilling the predicate to the downstream collector.

### 1.9.5. Implementing Collectors

In the collection process, a collector accumulates incoming stream elements in an internal data structure called a "result container." Each collector can choose a suitable result container type. If the stream is parallel, multiple result containers are filled concurrently and then merged. After accumulation and merging, the collector can optionally transform the final result container into another object that is the collection result.

Specifically, a Collector must implement four methods, each of which yields a function object:

- Supplier<A> supplier() supplies a result container of type A.
- BiConsumer<A,T> accumulator() accumulates a stream element of type T into a result container.
- BinaryOperator<A> combiner() combines two result containers into one.
- Function<A, R> finisher() transforms the final result container into the result of type R.

Let's look at the function objects of the Collectors.toList() collector:

- The supplier is a function yielding an empty ArrayList<T> or simply ArrayList<T>::new.
- The accumulator is (rc, t) -> rc.add(t) or List::add. (That method expression is why the result container is the first parameter.)
- The combiner concatenates two lists: (rc1, rc2) -> { rc1.addAll(rc2); return rc1; }.
- The finisher does nothing (which is the most common case).

The `characteristics` method of a collector returns a set of `Collector.Characteristics` flags that can help the stream with optimizing the collection process. There are three of them:

- `IDENTITY_FINISH`: the finisher need not be called.
- `UNORDERED`: it is ok not to call the accumulator in encounter order.
- `CONCURRENT`: it is safe to call the accumulator from multiple threads on the same result container.

`Collectors.joining` returns a collector with none of these characteristics. The result container is a `StringBuilder`. The finisher must convert it into a string. Accumulation order matters. And the result container is not threadsafe.

In contrast, the `toConcurrentMap` collector has all three characteristics. The result container is a threadsafe `ConcurrentHashMap`. Insertion order does not matter, and no finishing is needed.

---

**Caution:** Collectors are meant to work efficiently with parallel streams, which makes them unsuitable for certain terminal operations. Consider collecting every *n*th element from a stream. That can't be parallelized, since the accumulator can't track the indexes in each segment. But a collector can’t tell the stream that it wants the elements sequentially.

In this situation, you are better off using the gatherer API—see Section 1.11.

---

Let’s build a collector that statistically samples elements, accepting them with a given probability. For now, we’ll collect the accepted elements in a list.

Following the `Collectors` API, let’s make a method that yields the collector:

```
<T> Collector<T, List<T>, List<T>> randomSampling(double p) {
   return Collector.of(ArrayList<T>::new,
           (resultContainer, e) -> {
               if (Math.random() < p)
                   resultContainer.add(e);
               },
           (rc1, rc2) -> { rc1.addAll(rc2); return rc1; },
           Function.identity(),
           Collector.Characteristics.IDENTITY_FINISH);
   }
```

The `Collector.of` method has five parameters: the four functions and a varargs parameter for the characteristics.

This collector is almost the same as the `toList` collector, except that the accumulator only stores some of the elements.

Here is how to use it:

```
Stream<Integer> numbers = . . .;
List<Integer> sampled = numbers.collect(randomSampling(0.05)); // About every 20th element
```

What if we don't want the accepted elements in a list? Our API should support downstream collectors. Here is how:

```
<T, A, R> Collector<T, A, R> randomSampling(double p, Collector<T, A, R> downstream) {
        BiConsumer<A, T> downstreamAccumulator = downstream.accumulator();
        return Collector.of(downstream.supplier(),
                (resultContainer, e) -> {
                    if (Math.random() < p)
                        downstreamAccumulator.accept(resultContainer, e);
                    },
                downstream.combiner(),
                downstream.finisher(),
                downstream.characteristics().toArray(Collector.Characteristics[]::new));
    }
```

Here, `downstream` is a collector that accepts elements of type `T`, with a result container of type `A`, and result type `R`.

We can reuse the downstream's supplier, combiner, and finisher, but we must adapt its accumulator, so that only a fraction of the incoming elements are inserted.

Unfortunately, the API leaves something to be desired for this particular use case. I have to turn the downstream's set of characteristics into an array for the varargs parameter.

The program in Listing 1.8 has the complete example.

**Listing 1.8:** `v2ch01/collecting/ImplementingCollectors.java`

```
package v2ch01.collecting;

import module java.base;

class ImplementingCollectors {
    void main() {
        IO.println(IntStream.range(1, 1000)
            .boxed()
            .parallel()
            .collect(randomSampling(0.05)));
        IO.println(IntStream.range(1, 1000)
            .boxed()
            .parallel()
            .collect(randomSampling(0.05, Collectors.toSet())));
    }

    <T> Collector<T, List<T>, List<T>> randomSampling(double p) {
        return Collector.of(ArrayList<T>::new,
                (resultContainer, e) -> {
                    if (Math.random() < p)
                        resultContainer.add(e);
                    },
                (rc1, rc2) -> { rc1.addAll(rc2); return rc1; },
                Function.identity(),
                Collector.Characteristics.IDENTITY_FINISH);
```

```java
    }

    <T, A, R> Collector<T, A, R> randomSampling(double p, Collector<T, A, R> downstream) {
        BiConsumer<A, T> downstreamAccumulator = downstream.accumulator();
        return Collector.of(downstream.supplier(),
                (resultContainer, e) -> {
                    if (Math.random() < p)
                        downstreamAccumulator.accept(resultContainer, e);
                    },
                downstream.combiner(),
                downstream.finisher(),
                downstream.characteristics().toArray(Collector.Characteristics[]::new));
    }
}
```

**`java.util.stream.Collector<T, A, R>`** 8

- `Supplier<A> supplier()`
  The function to supply a new result container.
- `BiConsumer<A,T> accumulator()`
  The function to incorporate an element from the stream into a result container.
- `BinaryOperator<A> combiner()`
  The function to combine two result containers.
- `Function<A,R> finisher()`
  The function that turns the final result container into the result of the collector
- `Set<Collector.Characteristics> characteristics()`
  The characteristics of this collector.
- `static <T,A,R> Collector<T,A,R> of(Supplier<A> supplier, BiConsumer<A,T> accumulator, BinaryOperator<A> combiner, Function<A,R> finisher, Collector.Characteristics... characteristics)`
  yields a collector with the given supplier, accumulator, combiner, finisher, and characteristics.

## 1.10. Reduction Operations

The `reduce` method is another general mechanism for computing a value from a stream. In particular with mathematical operations, it is much easier to specify a reduction than it would be to implement the same operation with a collector. The simplest form takes a binary function and keeps applying it, starting with the first two elements. It's easy to explain this if the function is the sum:

```java
List<Integer> values = . . .;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

In this case, the `reduce` method computes $v_0 + v_1 + v_2 + \ldots$, where $v_i$ are the stream elements. The method returns an `Optional` because there is no valid result if the stream is empty.

**Note:** In this case, you can write `reduce(Integer::sum)` instead of `reduce((x, y) -> x + y)`.

More generally, you can use any operation that combines a partial result `x` with the next value `y` to yield a new partial result.

Here is another way of looking at reductions. Given a reduction operation *op*, the reduction yields $v_0$ *op* $v_1$ *op* $v_2$ *op* ..., where $v_i$ *op* $v_{i+1}$ denotes the function call *op*($v_i$, $v_{i+1}$). There are many operations that might be useful in practice—such as sum, product, string concatenation, maximum and minimum, set union or intersection.

If you want to use reduction with parallel streams, the operation must be *associative*: It shouldn't matter in which order you combine the elements. In math notation, ($x$ *op* $y$) *op* $z$ must be equal to $x$ *op* ($y$ *op* $z$). An example of an operation that is not associative is subtraction. For example, $(6 - 3) - 2 \neq 6 - (3 - 2)$.

Often, there is an *identity* $e$ such that $e$ *op* $x = x$, and that element can be used as the start of the computation. For example, 0 is the identity for addition, and you can use the second form of `reduce`:

```
List<Integer> values = . . .;
Integer sum = values.stream().reduce(0, (x, y) -> x + y);
    // Computes 0 + v0 + v1 + v2 + . . .
```

The identity value is returned if the stream is empty, and you no longer need to deal with the `Optional` class.

Now suppose you have a stream of objects and want to form the sum of some property, such as lengths in a stream of strings. You can't use the simple form of `reduce`. It requires a function `(T, T) -> T`, with the same types for the parameters and the result, but in this situation you have two types: The stream elements have type `String`, and the accumulated result is an integer. There is a form of `reduce` that can deal with this situation.

First, you supply an “accumulator” function `(total, word) -> total + word.length()`. That function is called repeatedly, forming the cumulative total. But when the computation is parallelized, there will be multiple computations of this kind, and you need to combine their results. You supply a second function for that purpose. The complete call is

```
int result = words.reduce(0,
    (total, word) -> total + word.length(),
    (total1, total2) -> total1 + total2);
```

**Note:** In practice, you probably won't use the `reduce` method a lot. It is usually easier to map to a stream of numbers and use one of its methods to compute sum, maximum, or minimum. (We discuss streams of numbers in Section 1.12.) In this particular example, you could have called `words.mapToInt(String::length).sum()`, which is both simpler and more efficient since it doesn’t involve boxing.

**Note:** There are times when `reduce` is not general enough. For example, suppose you want to collect the results in a `BitSet`. If the collection is parallelized, you can’t put the elements directly into a single `BitSet` because a `BitSet` object is not thread-safe. For that reason, you can't use reduce. Each segment needs to start out with its own empty set, and reduce only lets you supply one identity value. Instead, use the three-parameter form of collect. You provide three functions:

1. A *supplier* to construct new instances of the target object
2. An *accumulator* that adds an element to the target
3. A *combiner* that merges two target objects into one

Here is how the collect method works for a bit set:

```
BitSet result = stream.collect(BitSet::new, BitSet::set, BitSet::or);
```

---

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

- Optional<T> reduce(BinaryOperator<T> accumulator)
- T reduce(T identity, BinaryOperator<T> accumulator)
- <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
  form a cumulative total of the stream elements with the given accumulator function. If identity is provided, then it is the first value to be accumulated. If combiner is provided, it can be used to combine totals of segments that are accumulated separately.
- <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
  collects elements in a result of type R. On each segment, supplier is called to provide an initial result, accumulator is called to mutably add elements to it, and combiner is called to combine two results.

## 1.11. Gatherers

The stream API has an extension point for implementing arbitrary *terminal* operations: the collect method, whose argument is a Collector. Many collector implementations are provided in the Collectors class. You can also implement your own, as you have seen in Section 1.9.5.

Java 24 adds an extension point for intermediate operations. The gather method, whose argument is a Gatherer, turns a stream into another stream.

### 1.11.1. Predefined Gatherers

The Gatherers class provides a few Gatherer implementations. The windowFixed and windowSliding gatherers group adjacent elements together, yielding a stream of lists. It's easier to show them in action than to explain them in words:

```
IntStream.range(0, 10).boxed().gather(Gatherers.windowFixed(4)).toList()
    // [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9]]
IntStream.range(0, 10).boxed().gather(Gatherers.windowSliding(4)).toList()
    // [[0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6],
    // [4, 5, 6, 7], [5, 6, 7, 8], [6, 7, 8, 9]]
```

The windowSliding gatherer is useful for comparing adjacent elements of a stream. For example, here is how to drop adjacent duplicates:

```
Stream.concat(words, Stream.of((String) null))
    .gather(Gatherers.windowSliding(2))
    .filter(w -> !w.get(0).equalsIgnoreCase(w.get(1)))
    .map(w -> w.get(0))
```

By appending a `null` to the stream, the last element can be compared with something.

The `mapConcurrent` gatherer is similar to `map`, but the computations run in *virtual* threads. This is very useful for blocking operations. For example, if you have a list of URLs, you can read them concurrently.

```
String read(String url) {
    try {
        return new String(URI.create(url).toURL().openStream().readAllBytes());
    }
    catch (IOException e) {
        throw new UncheckedIOException(e);
    }
}

int max = 100; // Maximumum number of concurrent threads
List<String> contents = urls.stream()
    .gather(Gatherers.mapConcurrent(max, this::read))
    .toList();
```

If one of the computations throws an exception, all remaining tasks are cancelled, and the exception is rethrown.

---

**Note:** As you will see in Section 1.13, parallel streams are used to parallelize computationally intensive operations over multiple processor cores. In contrast, the `mapConcurrent` gatherer is intended for concurrent operations that mostly block.

---

Finally, there are `fold` and `scan` gatherers. A *fold* is a general mechanism for repeatedly applying an operation *op*. It starts with an initial value *iv* and computes $(((iv\ op\ v_0)\ op\ v_1)\ op\ v_2\ op)$ ..., where $v_i$ are the stream elements.

This looks similar to a reduction, which you saw in Section 1.10. However, reductions are intended for parallelizable operations. A fold is inherently sequential, and the operation need not be associative.

Some functional programmers love folds because any loop can be translated into a fold. Consider this computation of a number from its decimal digits:

```
int n = 0;
for (int d : digits)
    n = 10 * n + d;
```

Here is the same computation as a fold:

```
int n = digits.stream()
    .gather(Gatherers.fold(() -> 0, (x, y) -> x * 10 + y))
    .findFirst()
    .orElse(0);
```

The `fold` gatherer produces a stream with one element, or none if `digits` was empty.

As you can see from the example, you have to provide a supplier for the initial value, and the operation that is repeatedly applied.

---

**Note:** It seems odd that folding is implemented as a gatherer and not a collector. It would be more natural as a terminal operation since it produces a single result. But collectors are unable to signal that they must receive stream elements sequentially. As you will see in Section 1.11.2, gatherers have this capability. But a gatherer must yield a stream; in this case, one of length zero or one.

---

There is also a `scan` gatherer that produces a stream of all intermediate results. For example,

```
Stream.of(1, 7, 2, 9)
    .gather(Gatherers.scan(() -> 0, (x, y) -> x * 10 + y))
    .toList() // [1, 17, 172, 1729]
```

### 1.11.2. Implementing Gatherers

Implementing a gatherer is similar to implementing a collector (Section 1.9.5). A gatherer has a generic “intermediate state” object instead of the “result collection” of a collector. It too has four methods yielding function objects. Here `T` is the type of the stream elements, `A` the type of the intermediate state, and `R` the element type of the stream that the gatherer produces.

- `Supplier<A> initializer()` supplies an intermediate state instance.
- `Gatherer.Integrator<A, T, R> integrator()` processes a stream element.
- `BinaryOperator<A> combiner()` combines two intermediate states int one.
- `BiConsumer<A, Gatherer.Downstream<? super R>> finisher()` carries out a final operation after processing all stream elements.

`Gatherer.Integrator` is a functional interface with this method:

```
boolean integrate(A state, T element, Gatherer.Downstream<? super R> downstream)
```

In the integrator and finisher, you call the `push` method of the `Gatherer.Downstream` interface in order to send values to the stream that the gatherer produces. You can call `push` as often as you like, or not at all.

Very simple gatherers can be stateless. Then the initializer is not needed.

A combiner is only needed for parallel streams.

The finisher is only needed if you need to push values after the last element has been integrated. For example, the `windowFixed` gatherer pushes the short list of remaining elements in the finisher.

As an example, let us implement a gatherer that samples every *n*th element of a stream.

Here is how we want to use it:

```
List<String> samples = wordList.stream().gather(sampling(50)).toList();
```

The result is a list holding every 50th word from the original list.

The `sampling` method yields the gatherer:

```
static class IntermediateState {
    long index = -1;
}

<T> Gatherer<T, IntermediateState, T> sampling(int n) {
    return Gatherer.ofSequential(
        IntermediateState::new,
        Gatherer.Integrator.of((state, e, downstream) -> {
            state.index++;
            if (state.index % n == 0)
                return downstream.push(e);
            else
                return true;
            }));
    }
```

The `Gatherer.ofSequential` method yields a gatherer that calls the integrator in encounter order. There is also a `Gatherer.of` method for constructing parallelizable gatherers.

This gatherer doesn't have a combiner or finisher. In general, you don't provide arguments for unneeded initializers, combiners and finishers. Use one of the overloaded versions with just the gatherer operations you need.

---

**Tip:** Use these factory methods rather than declaring an explicit class implementing the `Gatherer` interface. It is less work, and it is easier to tell whether your gatherer is sequential or parallelizable.

You may wonder how the stream implementation distinguishes the two. A gatherer is sequential if the `combiner` method yields the exact same do-nothing instance as the one returned by the static `defaultCombiner` method. The factory methods handle this detail.

If you declare your own class, you need to be careful. A sequential gatherer should *not* override `combiner` but instead keep the default method, which returns `defaultCombiner()`. If you have a parallelizable gatherer that doesn't need a combiner, you need to declare one anyway.

---

Now let us look more closely at the `sampling` gatherer. The state holds the index of the element. The initializer constructs a new state instance.

The integrator is declared using the `Gatherer.Integrator.of` factory method. Whenever an element is processed, the index is incremented. The element is pushed when the index is divisible by the sampling count.

Note that the integrator returns a `boolean`. You should return `false` when the pipeline "short-circuits" and no longer wants to accept elements. This can happen for two reasons. Your gatherer might want to short-circuit on its own. Simply return `false` in that case. (In the `sampling` example, that never happens.) Alternatively, a downstream operation can short-circuit. Then the call to `push` returns `false`. The integrator needs to return that downstream status.

---

**Note:** The `Gatherer.Downstream` interface has a method `isRejecting`. It allows an integrator to check whether a value would be accepted before it embarks on a costly computation.

Do not use the `isRejecting` method for reporting the downstream status. You want to keep the integrator code as simple as possible. Then it is more likely that the just-in-time compiler will inline it.

---

For optimizing the performance of a stream pipeline, it is important to know whether it contains any short-circuiting operations. An integrator that doesn't short-circuit should signal that fact by implementing the `Gatherer.Integrator.Greedy` interface. There is a factory method for that:

```
<T> Gatherer<T, IntermediateState, T> sampling(int n) {
    return Gatherer.ofSequential(
        IntermediateState::new,
        Gatherer.Integrator.ofGreedy((state, e, downstream) -> {
            state.index++;
            if (state.index % n == 0)
                return downstream.push(e);
            else
                return true;
            }));
}
```

---

**Tip:** You could create a class that implements the interface, but using a factory method is easier.

---

**Tip:** The `Gatherer.Integrator.of` method that you saw in the first implementation was not strictly necessary. We could have just used a lambda expression. It would have been converted to a `Gatherer.Integrator`. But the factory method tells the reader more clearly that this is not a greedy integrator.

---

As you have seen, writing your own gatherer may seem a bit daunting at first, but it is not all that complex. Follow these steps:

1. Decide whether your gatherer can execute in parallel (`Gatherer.of`), or whether it must see the elements sequentially (`Gatherer.ofSequential`).

2. Decide whether the integrator may short-circuit (`Gatherer.Integrator.of`) or if it always accepts elements (`Gatherer.Integrator.ofGreedy`).
3. Do you need state? If so, provide a class for the state, and if your gatherer is not sequential, implement a combiner.
4. In the integrator, return `false` when short-ciruiting, or when the downstream push is rejected.

---

**Note:** Unlike collectors, gatherers don't use enumerations for their characteristics. They use these alternative mechanisms:

1. The integrator of a greedy gatherer implements `Gatherer.Integrator.Greedy`. Otherwise the gatherer is short-circuiting.
2. The combiner of a sequential gatherer is the object returned by `Gatherer.defaultCombiner()`. Otherwise the gatherer is parallelizable.
3. The initializer of a stateless gatherer is the object returned by `Gatherer.defaultInitializer()`. Otherwise the gatherer is stateful.

---

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

- `<R> Stream<R> gather(Gatherer<? super T, ?, R> gatherer)` **24**
  yields a stream that results from processing the elements of this stream with the given gatherer.

***java.util.stream.Gatherers*** **24**

- `static <TR> Gatherer<TR,?,List<TR>> windowFixed(int windowSize)`
  `static <TR> Gatherer<TR,?,List<TR>> windowSliding(int windowSize)`
  yields a gatherer that groups stream elements into separate or overlapping windows of the given size.
- `static <T,R> Gatherer<T,?,R> fold(Supplier<R> initial, BiFunction<? super R, ? super T, ? extends R> folder)`
  `static <T,R> Gatherer<T,?,R> scan(Supplier<R> initial, BiFunction<? super R, ? super T, ? extends R> scanner)`
  yield a gatherer that repeatedly applies the binary operation to the preceding result and the next stream element. `fold` produces only the final result, while `scan` yields all intermediate results.
- `static <T,R> Gatherer<T,?,R> mapConcurrent(int maxConcurrency, Function<? super T, ? extends R> mapper)`
  yields a gatherer that applies the given function, running the computations in at most `maxConcurrency` virtual threads, producing a stream of the results.

***java.util.stream.Gatherer.Downstream<T>*** **24**

- `boolean push(T element)`
  pushes the element to the next stage of the processing pipeline. Returns `false` if the element was rejected.

- `boolean isRejecting()`
  returns `true` if the processing pipeline no longer accepts elements.

***java.util.stream.Gatherer<T, A, R>*** **24**

- `Supplier<A> initializer()`
  `Gatherer.Integrator<A,T,R> integrator()`
  `BinaryOperator<A> combiner()`
  `BiConsumer<A, Gatherer.Downstream<? super R>> finisher()`
  return the initializer, integrator, combiner, and finisher of this gatherer.
- `static <T,R> Gatherer<T,Void,R> of(Gatherer.Integrator<Void,T,R> integrator)`
  `static <T,R> Gatherer<T,Void,R> of(Gatherer.Integrator<Void,T,R> integrator, BiConsumer<Void, Gatherer.Downstream<? super R>> finisher)`
  `static <T,A,R> Gatherer<T,A,R> of(Supplier<A> initializer, Gatherer.Integrator<A,T,R> integrator, BinaryOperator<A> combiner, BiConsumer<A, Gatherer.Downstream<? super R>> finisher)`
  yield a parallelizable gatherer with the given initializer, integrator, combiner, and finisher. When there is no initializer, the gatherer is stateless.
- `static <T,R> Gatherer<T,Void,R> ofSequential(Gatherer.Integrator<Void,T,R> integrator)`
  `static <T,A,R> Gatherer<T,A,R> ofSequential(Supplier<A> initializer, Gatherer.Integrator<A,T,R> integrator)`
  `static <T,R> Gatherer<T,Void,R> ofSequential(Gatherer.Integrator<Void,T,R> integrator, BiConsumer<Void, Gatherer.Downstream<? super R>> finisher)`
  `static <T,A,R> Gatherer<T,A,R> ofSequential(Supplier<A> initializer, Gatherer.Integrator<A,T,R> integrator, BiConsumer<A, Gatherer.Downstream<? super R>> finisher)`
  yield a sequential gatherer with the given initializer, integrator, and finisher. When there is no initializer, the gatherer is stateless.

***java.util.stream.Gatherer.Integrator<T>*** **24**

- `boolean integrate(A state, T element, Gatherer.Downstream<? super R> downstream)`
  This method is called for each stream element. Update the state, and push a result downstream when appropriate. Return `false` if further elements will be rejected.
- `static <A,T,R> Gatherer.Integrator<A,T,R> of(Gatherer.Integrator<A,T,R> integrator)`
  `static <A,T,R> Gatherer.Integrator.Greedy<A,T,R> ofGreedy(Gatherer.Integrator.Greedy<A,T,R> greedy)`
  yield a short-circuiting or greedy integrator. The argument is intended to be a lambda expression.

## 1.12. Primitive Type Streams

So far, we have collected integers in a `Stream<Integer>`, even though it is clearly inefficient to wrap each integer into a wrapper object. The same is true for the other primitive types—`double`, `float`, `long`, `short`, `char`, `byte`, and `boolean`. The stream library has specialized types `IntStream`, `LongStream`, and `DoubleStream` that store primitive values directly, without using wrappers. If you want to store `short`, `char`, `byte`, and `boolean`, use an `IntStream`; for `float`, use a `DoubleStream`.

To create an IntStream, call the IntStream.of and Arrays.stream methods:

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to); // values is an int[] array
```

As with object streams, you can also use the static generate and iterate methods. In addition, IntStream and LongStream have static methods range and rangeClosed that generate integer ranges with step size one:

```
IntStream zeroToNinetyNine = IntStream.range(0, 100); // Upper bound is excluded
IntStream zeroToHundred = IntStream.rangeClosed(0, 100); // Upper bound is included
```

The CharSequence interface has methods codePoints and chars that yield an IntStream of the Unicode codes of the characters or of the code units in the UTF-16 encoding. (See Chapter 2 for the sordid details.)

```
String greeting = "Ahoy 🏴‍☠️";
IntStream codes = greeting.codePoints();
    // The stream with values 67, 105, 97, 111, 32, 127988, 8205, 9760, 65039
```

The RandomGenerator interface has methods ints, longs, and doubles that return primitive type streams of random numbers.

```
IntStream randomIntegers = RandomGenerator.getDefault().ints();
```

When you have a stream of objects, you can transform it to a primitive type stream with the mapToInt, mapToLong, or mapToDouble methods. For example, if you have a stream of strings and want to process their lengths as integers, you might as well do it in an IntStream:

```
Stream<String> words = . . .;
IntStream lengths = words.mapToInt(String::length);
```

**Caution:** When converting a Stream<Integer> to an IntStream, use

```
numbers.mapToInt(i -> i)
```

and not

```
numbers.mapToInt(Function.identity()) // ERROR
```

The latter is a compile-time error, since there is no automatic unboxing.

To convert a primitive type stream to an object stream, use the boxed method:

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

Generally, the methods on primitive type streams are analogous to those on object streams. Here are the most notable differences:

- The toArray methods return primitive type arrays.

- Methods that yield an optional result return an `OptionalInt`, `OptionalLong`, or `OptionalDouble`. These classes are analogous to the `Optional` class, but they have methods `getAsInt`, `getAsLong`, and `getAsDouble` instead of the `get` method.
- There are methods `sum`, `average`, `max`, and `min` that return the sum, count, average, maximum, and minimum. These methods are not defined for object streams.
- The `summaryStatistics` method yields an object of type `IntSummaryStatistics`, `LongSummaryStatistics`, or `DoubleSummaryStatistics` that can simultaneously report the sum, count, average, maximum, and minimum of the stream.

The program in Listing 1.9 gives examples for the API of primitive type streams.

**Listing 1.9: `v2ch01/streams/PrimitiveTypeStreams.java`**

```
package v2ch01.streams;

import module java.base;

class PrimitiveTypeStreams {
    void main() throws Exception {
        IntStream is1 = IntStream.generate(() -> (int) (Math.random() * 100));
        show("is1", is1);
        IntStream is2 = IntStream.range(5, 10);
        show("is2", is2);
        IntStream is3 = IntStream.rangeClosed(5, 10);
        show("is3", is3);

        String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));

        Stream<String> words = Stream.of(contents.split("\\PL+"));
        IntStream is4 = words.mapToInt(String::length);
        show("is4", is4);
        String sentence = "\uD835\uDD46 is the set of octonions.";
        IO.println(sentence);
        IntStream codes = sentence.codePoints();
        IO.println(codes.mapToObj(c -> "%X ".formatted(c)).collect(Collectors.joining()));

        Stream<Integer> integers = IntStream.range(0, 100).boxed();
        IntStream is5 = integers.mapToInt(Integer::intValue);
        show("is5", is5);
    }

    void show(String title, IntStream stream) {
        final int SIZE = 10;
        int[] firstElements = stream.limit(SIZE + 1).toArray();
        IO.print(title + ": ");
        for (int i = 0; i < firstElements.length; i++) {
            if (i > 0) IO.print(", ");
            if (i < SIZE)
                IO.print(firstElements[i]);
            else
                IO.print("...");
        }
        IO.println();
    }
}
```

***java.util.stream.IntStream* 8**

- `static IntStream range(int startInclusive, int endExclusive)`
- `static IntStream rangeClosed(int startInclusive, int endInclusive)`
  yield an `IntStream` with the integers in the given range.
- `static IntStream of(int... values)`
  yields an `IntStream` with the given elements.
- `int[] toArray()`
  yields an array with the elements of this stream.
- `int sum()`
- `OptionalDouble average()`
- `OptionalInt max()`
- `OptionalInt min()`
- `IntSummaryStatistics summaryStatistics()`
  yield the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these values can be obtained.
- `Stream<Integer> boxed()`
  yields a stream of wrapper objects for the elements in this stream.

***java.util.stream.LongStream* 8**

- `static LongStream range(long startInclusive, long endExclusive)`
- `static LongStream rangeClosed(long startInclusive, long endInclusive)`
  yield a `LongStream` with the integers in the given range.
- `static LongStream of(long... values)`
  yields a `LongStream` with the given elements.
- `long[] toArray()`
  yields an array with the elements of this stream.
- `long sum()`
- `OptionalDouble average()`
- `OptionalLong max()`
- `OptionalLong min()`
- `LongSummaryStatistics summaryStatistics()`
  yield the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these values can be obtained.
- `Stream<Long> boxed()`
  yields a stream of wrapper objects for the elements in this stream.

***java.util.stream.DoubleStream* 8**

- `static DoubleStream of(double... values)`
  yields a `DoubleStream` with the given elements.
- `double[] toArray()`
  yields an array with the elements of this stream.

- `double sum()`
- `OptionalDouble average()`
- `OptionalDouble max()`
- `OptionalDouble min()`
- `DoubleSummaryStatistics summaryStatistics()`
  yield the sum, average, maximum, or minimum of the elements in this stream, or an object from which all four of these values can be obtained.
- `Stream<Double> boxed()`
  yields a stream of wrapper objects for the elements in this stream.

**`java.lang.CharSequence`** **1.0**

- `IntStream codePoints()` **8**
  yields a stream of all Unicode code points.

**`java.util.random.RandomGenerator`** **17**

- `IntStream ints()`
- `IntStream ints(int randomNumberOrigin, int randomNumberBound)`
- `IntStream ints(long streamSize)`
- `IntStream ints(long streamSize, int randomNumberOrigin, int randomNumberBound)`
- `LongStream longs()`
- `LongStream longs(long randomNumberOrigin, long randomNumberBound)`
- `LongStream longs(long streamSize)`
- `LongStream longs(long streamSize, long randomNumberOrigin, long randomNumberBound)`
- `DoubleStream doubles()`
- `DoubleStream doubles(double randomNumberOrigin, double randomNumberBound)`
- `DoubleStream doubles(long streamSize)`
- `DoubleStream doubles(long streamSize, double randomNumberOrigin, double randomNumberBound)`
  yield streams of random numbers. If `streamSize` is provided, the stream is finite with the given number of elements. When bounds are provided, the elements are between `randomNumberOrigin` (inclusive) and `randomNumberBound` (exclusive).
- `DoubleStream equiDoubles(double left, double right, boolean isLeftIncluded, boolean isRightIncluded)` **22**
  yields a stream of floating-point numbers between `left` and `right`, uniformly drawn from the largest possible sequence of equidistant numbers in the target interval. In contrast, the `doubles` methods draw uniformly from floating-point numbers in the interval between 0 and 1, then transforming into the target interval. That transformation can introduce a slight distortion.

**java.util.OptionalInt 8**

**java.util.OptionalLong 8**

**java.util.OptionalDouble 8**

- `static Optional`*Xxx* `of(`*xxx* `value)`
  yields an optional object with the supplied primitive type value.
- *xxx* `getAs`*Xxx*`()`
  yields the value of this optional object, or throws a `NoSuchElementException` if it is empty.
- *xxx* `orElse(`*xxx* `other)`
- *xxx* `orElseGet(`*Xxx*`Supplier other)`
  yield the value of this optional object, or the alternative value if this object is empty.
- `void ifPresent(`*Xxx*`Consumer consumer)`
  If this optional object is not empty, passes its value to `consumer`.

**java.util.IntSummaryStatistics 8**

**java.util.LongSummaryStatistics 8**

**java.util.DoubleSummaryStatistics 8**

- `long getCount()`
- *xxx* `getSum()`
- `double getAverage()`
- *xxx* `getMax()`
- *xxx* `getMin()`
  yield the count, sum, average, maximum, and minimum of the collected elements.

## 1.13. Parallel Streams

Streams make it easy to parallelize bulk operations. The process is mostly automatic, but you need to follow a few rules. First of all, you must have a parallel stream. You can get a parallel stream from any collection with the `Collection.parallelStream()` method:

```
Stream<String> parallelWords = words.parallelStream();
```

Moreover, the `parallel` method converts any sequential stream into a parallel one.

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

As long as the stream is in parallel mode when the terminal method executes, all intermediate stream operations will be parallelized.

When stream operations run in parallel, the intent is that the same result is returned as if they had run serially. It is important that the operations are *stateless* and can be executed in an arbitrary order.

Here is an example of something you cannot do. Suppose you want to count all short words in a stream of strings:

```
var shortWords = new int[12];
words.parallelStream().forEach(
    s -> { if (s.length() < 12) shortWords[s.length()]++; });
        // ERROR--race condition!
IO.println(Arrays.toString(shortWords));
```

This is very, very bad code. The function passed to `forEach` runs concurrently in multiple threads, each updating a shared array. As you saw in Chapter 10 of Volume I, that's a classic *race condition*. If you run this program multiple times, you are quite likely to get a different sequence of counts in each run—each of them wrong.

It is your responsibility to ensure that any functions you pass to parallel stream operations are safe to execute in parallel. The best way to do that is to stay away from mutable state. In this example, you can safely parallelize the computation if you group strings by length and count them:

```
Map<Integer, Long> shortWordCounts
    = words.parallelStream()
        .filter(s -> s.length() < 12)
        .collect(groupingBy(
            String::length,
            counting()));
```

By default, streams that arise from ordered collections (arrays and lists), from ranges, generators, and iterators, or from calling `Stream.sorted`, are *ordered*. Results are accumulated in the order of the original elements, and are entirely predictable. If you run the same operations twice, you will get exactly the same results.

Ordering does not preclude efficient parallelization. For example, when computing `stream.map(fun)`, the stream can be partitioned into *n* segments, each of which is concurrently processed. Then the results are reassembled in order.

Some operations can be more effectively parallelized when the ordering requirement is dropped. By calling the `Stream.unordered` method, you indicate that you are not interested in ordering. One operation that can benefit from this is `Stream.distinct`. On an ordered stream, `distinct` retains the first of all equal elements. That impedes parallelization—the thread processing a segment can't know which elements to discard until the preceding segment has been processed. If it is acceptable to retain *any* of the unique elements, all segments can be processed concurrently (using a shared set to track duplicates).

You can also speed up the `limit` method by dropping ordering. If you just want any `n` elements from a stream and you don’t care which ones you get, call

```
Stream<String> sample = words.parallelStream().unordered().limit(n);
```

As discussed in Section 1.9.2, merging maps is expensive. For that reason, the `Collectors.groupingByConcurrent` method uses a shared concurrent map. To benefit from parallelism, the order of the map values will not be the same as the stream order.

```
Map<Integer, List<String>> result = words.parallelStream().collect(
    Collectors.groupingByConcurrent(String::length));
    // Values aren't collected in stream order
```

Of course, you won't care if you use a downstream collector that is independent of the ordering, such as

```
Map<Integer, Long> wordCounts
    = words.parallelStream()
        .collect(
            groupingByConcurrent(
                String::length,
                counting()));
```

Don't turn all your streams into parallel streams in the hope of speeding up operations. Keep these issues in mind:

- There is a substantial overhead to parallelization that will only pay off for very large data sets.
- Parallelizing a stream is only a win if the underlying data source can be effectively split into multiple parts.
- The thread pool that is used by parallel streams can be starved by blocking operations such as file I/O or network access.

Parallel streams work best with huge in-memory collections of data and computationally intensive processing.

---

**Tip:** If you process the lines of a huge file, parallelizing the stream may improve performance. The `Files.lines` method uses a memory-mapped file, to make splitting effective. (If you simply read a line at a time, then you would have to read the first half of the file before the second half, and parallelization would not work.)

---

**Note:** If you need random numbers in parallel streams, obtain a generator from the `RandomGenerator.SplittableGenerator.of` factory method.

---

**Note:** By default, parallel streams use the global fork-join pool returned by `ForkJoinPool.commonPool`. That is fine if your operations don't block and you don't share the pool with other tasks. There is a trick to substitute a different fork-join pool. Execute the parallel stream operations in a thread from that pool:

```
ForkJoinPool customPool = . . .;
result = customPool.submit(() ->
    stream.parallel().map(. . .).collect(. . .)).get();
```

Or, asynchronously:

```
CompletableFuture.supplyAsync(() ->
        stream.parallel().map(. . .).collect(. . .), customPool)
    .thenAccept(result -> . . .);
```

This only works with a fork-join pool, not with arbitrary executors.

---

The example program in Listing 1.10 demonstrates how to work with parallel streams.

**Listing 1.10:** `v2ch01/parallel/ParallelStreams.java`

```
package v2ch01.parallel;

import static java.util.stream.Collectors.*;

import module java.base;

class ParallelStreams {
    void main() throws Exception {
        String contents = Files.readString(Path.of("gutenberg", "alice30.txt"));
        List<String> wordList = List.of(contents.split("\\PL+"));

        // Very bad code ahead
        var shortWords = new int[10];
        wordList.parallelStream().forEach(s -> {
            if (s.length() < 10) shortWords[s.length()]++;
        });
        IO.println(Arrays.toString(shortWords));

        // Try again--the result will likely be different (and also wrong)
        Arrays.fill(shortWords, 0);
        wordList.parallelStream().forEach(s -> {
            if (s.length() < 10) shortWords[s.length()]++;
        });
        IO.println(Arrays.toString(shortWords));

        // Remedy: Group and count
        Map<Integer, Long> shortWordCounts = wordList.parallelStream()
                .filter(s -> s.length() < 10).collect(groupingBy(String::length, counting()));

        IO.println(shortWordCounts);

        // Downstream order not deterministic
        Map<Integer, List<String>> result = wordList.parallelStream()
                .collect(Collectors.groupingByConcurrent(String::length));

        IO.println(result.get(14));

        result = wordList.parallelStream()
                .collect(Collectors.groupingByConcurrent(String::length));

        IO.println(result.get(14));

        Map<Integer, Long> wordCounts = wordList.parallelStream()
                .collect(groupingByConcurrent(String::length, counting()));
```

```

        IO.println(wordCounts);
    }
}
```

***java.util.stream.BaseStream<T,S extends BaseStream<T,S>>* 8**

- `S parallel()`
  yields a parallel stream with the same elements as this stream.
- `S unordered()`
  yields an unordered stream with the same elements as this stream.

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

- `Stream<E> parallelStream()` **8**
  yields a parallel stream with the elements of this collection.

In this chapter, you have learned how to put the stream library to use. The next chapter covers another important topic: processing input and output.

This page intentionally left blank

# Index

## Symbols

## A

# C

# D

# F

# G

# H

## M

# N

# O

# P

# S

# U

# V

# X