17.13 Creating a `Stream<String>` from a File

Figure 17.22 uses lambdas and streams to summarize the number of occurrences of each word in a file, then display a summary of the words in alphabetical order grouped by starting letter. This is commonly called a concordance:

```
http://en.wikipedia.org/wiki/Concordance_(publishing)
```

Concordances are often used to analyze published works. For example, concordances of William Shakespeare's and Christopher Marlowe's works (among others) have been used to question whether they are the same person. Figure 17.23 shows the program's output. Line 14 of Fig. 17.22 creates a regular expression `Pattern` that we'll use to split lines of text into their individual words. The `Pattern \s+` represents one or more consecutive white-space characters—recall that because `\` indicates an escape sequence in a `String`, we must specify each `\` in a regular expression as `\\`. As written, this program assumes that the file it reads contains no punctuation, but you could use regular-expression techniques from Section 14.7 to remove punctuation.

```
// Fig. 17.22: StreamOfLines.java
// Counting word occurrences in a text file.
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Map;
import java.util.TreeMap;
import java.util.regex.Pattern;
import java.util.stream.Collectors;

public class StreamOfLines {
   public static void main(String[] args) throws
      // Regex that matches one or more consecut
      Pattern pattern = Pattern.compile("\\s+");

      // count occurrences of each word in a Str
      Map<String, Long> wordCounts =
         Files.lines(Paths.get("Chapter2Paragrap
              .flatMap(line -> pattern.splitAsSt
              .collect(Collectors.groupingBy(Str
                 TreeMap::new, Collectors.counti

      // display the words grouped by startin
      wordCounts.entrySet()
         .stream()
         .collect(
            Collectors.groupingBy(entry -> en
               TreeMap::new, Collectors.toLis
         .forEach((letter, wordList) -> {
            System.out.printf("%n%C%n", lette
            wordList.stream().forEach(word ->
               "%13s: %d%n", word.getKey(), w
         });
   }
}
```

Fig. 17.22

Counting word occurrences in a text file.

A	I	R
a: 2	inputs: 1	result: 1
and: 3	instruct: 1	results: 2
application: 2	introduces: 1	run: 1
arithmetic: 1		
	J	S
B	java: 1	save: 1
begin: 1	jdk: 1	screen: 1
		show: 1
C	L	sum: 1
calculates: 1	last: 1	
calculations: 1	later: 1	T
chapter: 1	learn: 1	that: 3
chapters: 1		the: 7
commandline: 1	M	their: 2
compares: 1	make: 1	then: 2
comparison: 1	messages: 2	this: 2
compile: 1		to: 4
computer: 1	N	tools: 1

```
D                        numbers: 2          two: 2
     decisions: 1
   demonstrates: 1  O                  U
      display: 1         obtains: 1          use: 2
     displays: 2            of: 1           user: 1
                            on: 1
E                         output: 1    W
      example: 1                              we: 2
     examples: 1    P                       with: 1
                         perform: 1
F                        present: 1    Y
        for: 1           program: 1        you’ll: 2
       from: 1         programming: 1
                        programs: 2
H
        how: 2
```

Fig. 17.23

Output of Fig. 17.22 arranged in three columns.

Summarizing the

Occurrences of Each Word in the File

The stream pipeline in lines 17–21

```
Map<String, Long> wordCounts =
   Files.lines(Paths.get("Chapter2Paragraph.txt"))
        .flatMap(line -> pattern.splitAsStream(line))
        .collect(Collectors.groupingBy(String::toLowe
           TreeMap::new, Collectors.counting()));
```

summarizes the contents of the text file `"Chapter2Paragraph.txt"` (which is located in the folder with the example) into a `Map<String, Long>` in which each `String` key is a word in the file and the corresponding `Long` value is the number of occurrences of that word. The pipeline performs the following tasks:

- 8

  Line 18 calls `Files` method `lines` (added in Java SE 8) which returns a `Stream<String>` that reads lines of text from a file and returns each line as a `String`. Class `Files` (package `java.nio.file`) is one of many classes throughout the Java APIs which provide methods that return `Stream`s.

- 8

  Line 19 uses `Stream` method `flatMap` to break each line of text into its separate words. Method `flatMap` receives a `Function` that maps an object into a stream of elements. In this case, the object is a `String` containing words and the result is a `Stream<String>` for the individual words. The lambda in line 19 passes the `String` representing a line of text to `Pattern` method `splitAsStream` (added in Java SE 8), which uses the regular expression specified in the `Pattern` (line 14) to tokenize the `String` into its individual words. The result of line 19 is a `Stream<String>` for the individual words in all the lines of text. (This lambda could be replaced with the method reference `pattern::splitAsStream`.)

- Lines 20–21 use `Stream` method `collect` to count the frequency of each word and place the words and their counts into a `TreeMap<String, Long>`—a `TreeMap` because maintains its keys in sorted order. Here, we use a version of `Collectors` method `groupingBy` that receives three arguments—a classifier, a `Map` factory and a downstream `Collector`. The classifier is a `Function` that returns objects for use as keys in the resulting `Map`—the method reference `String::toLowerCase` converts each word to lowercase. The `Map` factory is an object that implements interface `Supplier` and returns a new `Map` collection—here we use the **constructor reference** `TreeMap::new`, which returns a `TreeMap` that maintains its keys in sorted order. The compiler converts a constructor reference into a parameterless lambda that returns a new `TreeMap`. `Collectors.counting()` is the downstream `Collector` that determines the number of occurrences of each key in the stream. The `TreeMap`'s key type is determined by the classifier `Function`'s return type (`String`), and the `TreeMap`'s value type is determined by the downstream collector—`Collectors.counting()` returns a `Long`.

Displaying the Summary Grouped by Starting Letter

Next, the stream pipeline in lines 24–33 groups the key–value pairs in the `Map wordCounts` by the keys' first letter:

```
wordCounts.entrySet()
          .stream()
          .collect(
             Collectors.groupingBy(entry -> entry.get
               TreeMap::new, Collectors.toList()))
          .forEach((letter, wordList) -> {
             System.out.printf("%n%C%n", letter);
             wordList.stream().forEach(word -> System
                "%13s: %d%n", word.getKey(), word.get
          });
```

This produces a new `Map` in which each key is a `Character` and the corresponding value is a `List` of the key–value pairs in `wordCounts` in which the key starts with the `Character`. The statement performs the following tasks:

- First we need to get a `Stream` for processing the key–value pairs in `wordCounts`. Interface `Map` does not contain any methods that return `Stream`s. So, line 24 calls `Map` method `entrySet` on `wordCounts` to get a `Set` of `Map.Entry` objects that each contain one key–value pair from `wordCounts`. This produces an object of type `Set<Map.Entry<String, Long>>`.
- Line 25 calls `Set` method `stream` to get a `Stream<Map.Entry<String, Long>>`.
- Lines 26–28 call `Stream` method `collect` with three arguments—a classifier, a `Map` factory and a downstream `Collector`. The classifier `Function` in this case gets the key from the `Map.Entry` then uses `String` method `charAt` to get the key's first character—this becomes a `Character` key in the resulting `Map`. Once again, we use the constructor reference `TreeMap::new` as the `Map` factory to create a `TreeMap` that maintains its keys in sorted order. The downstream `Collector` (`Collectors.toList()`) places the `Map.Entry` objects into a `List` collection. The result of `collect` is a `Map<Character, List<Map.Entry<String, Long>>>`.
- Finally, to display the summary of the words and their counts by letter (i.e., the concordance), lines 29–33 pass a lambda to `Map` method `forEach`. The lambda (a `BiConsumer`) receives two parameters —`letter` and `wordList` represent the `Character` key and the `List` value, respectively, for each key–value pair in the `Map` produced by the

preceding `collect` operation. The body of this lambda has two statements, so it *must* be enclosed in curly braces. The statement in line 30 displays the `Character` key on its own line. The statement in lines 31–32 gets a `Stream<Map.Entry<String, Long>>` from the `wordList`, then calls `Stream` method `forEach` to display the key and value from each `Map.Entry` object.