

CHAPTER

30

AGGREGATE OPERATIONS FOR COLLECTION STREAMS

Objectives

- To use aggregate operations on collection streams to simplify coding and improve performance (§30.1).
- To create a stream pipeline, apply lazy intermediate methods (**skip**, **limit**, **filter**, **distinct**, **sorted**, **map**, and **mapToInt**), and terminal methods (**count**, **sum**, **average**, **max**, **min**, **forEach**, **findFirst**, **firstAny**, **anyMatch**, **allMatch**, **noneMatch**, and **toArray**) on a stream (§30.2).
- To process primitive data values using the **IntStream**, **LongStream**, and **DoubleStream** (§30.3).
- To create parallel streams for fast execution (§30.4).
- To reduce the elements in a stream into a single result using the **reduce** method (§30.5).
- To place the elements in a stream into a mutable collection using the **collect** method (§30.6).
- To group the elements in a stream and apply aggregate methods for the elements in the groups (§30.7).
- To use a variety of examples to demonstrate how to simplify coding using streams (§30.8).



30.1 Introduction



Using aggregate operations on collection streams can greatly simplify coding and improve performance.

Often, you need to process data in an array or a collection. Suppose, for instance, that you need to count the number of elements in a set that is greater than **60**. You may write the code using a foreach loop as follows:

```
Double[] numbers = {2.4, 55.6, 90.12, 26.6};
Set<Double> set = new HashSet<>(Arrays.asList(numbers));
int count = 0;
for (double e: set)
    if (e > 60)
        count++;
System.out.println("Count is " + count);
```

The code is fine. However, Java provides a better and simpler way for accomplishing the task. Using the aggregate operations, you can rewrite the code as follows:

```
System.out.println("Count is "
    + set.stream().filter(e -> e > 60).count());
```

Invoking the `stream()` method on a set returns a **Stream** for the elements from a set. The `filter` method specifies a condition for selecting the elements whose value is greater than **60**. The `count()` method returns the number of elements in the stream that satisfy the condition.

A *collection stream* or simply *stream* is a sequence of elements. The operations on a stream is called *aggregate operations* (also known as *stream operations*) because they apply to all the data in the stream. The `filter` and `count` are the examples of aggregate operations. The code written using a foreach loop describes the process how to obtain the count, that is, for each element, if it is greater than **60**, increase the count. The code written using the aggregate operations tells the program to return the count for the elements greater than **60**, but it does not specify how the count is obtained. Clearly, using the aggregate operations leaves the detailed implementation to the computer, therefore, makes the code concise and simpler. Moreover, the aggregate operations on a stream can be executed in parallel to take advantage of multiple processors. So, the code written using aggregate operations usually run faster than the ones using a foreach loop.

Java provides many aggregate operations and many different ways of using aggregate operations. This chapter gives a comprehensive coverage on aggregate operations and streams.



30.1.1 What are the benefits of using aggregate operations on collection streams for processing data?

30.2 Stream Pipelines



A stream pipeline consists of a stream created from a data source, zero or more intermediate methods, and a final terminal method.

An array or a collection is an object for storing data. A stream is a transient object for processing data. After data is processed, the stream is destroyed. Java 8 introduced a new default `stream()` method in the **Collection** interface to return a **Stream** object. The **Stream** interface extends the **BaseStream** interface and contains the aggregate methods and the utility methods as shown in Figure 30.1.

stream
aggregate operations

why using aggregate
operations?

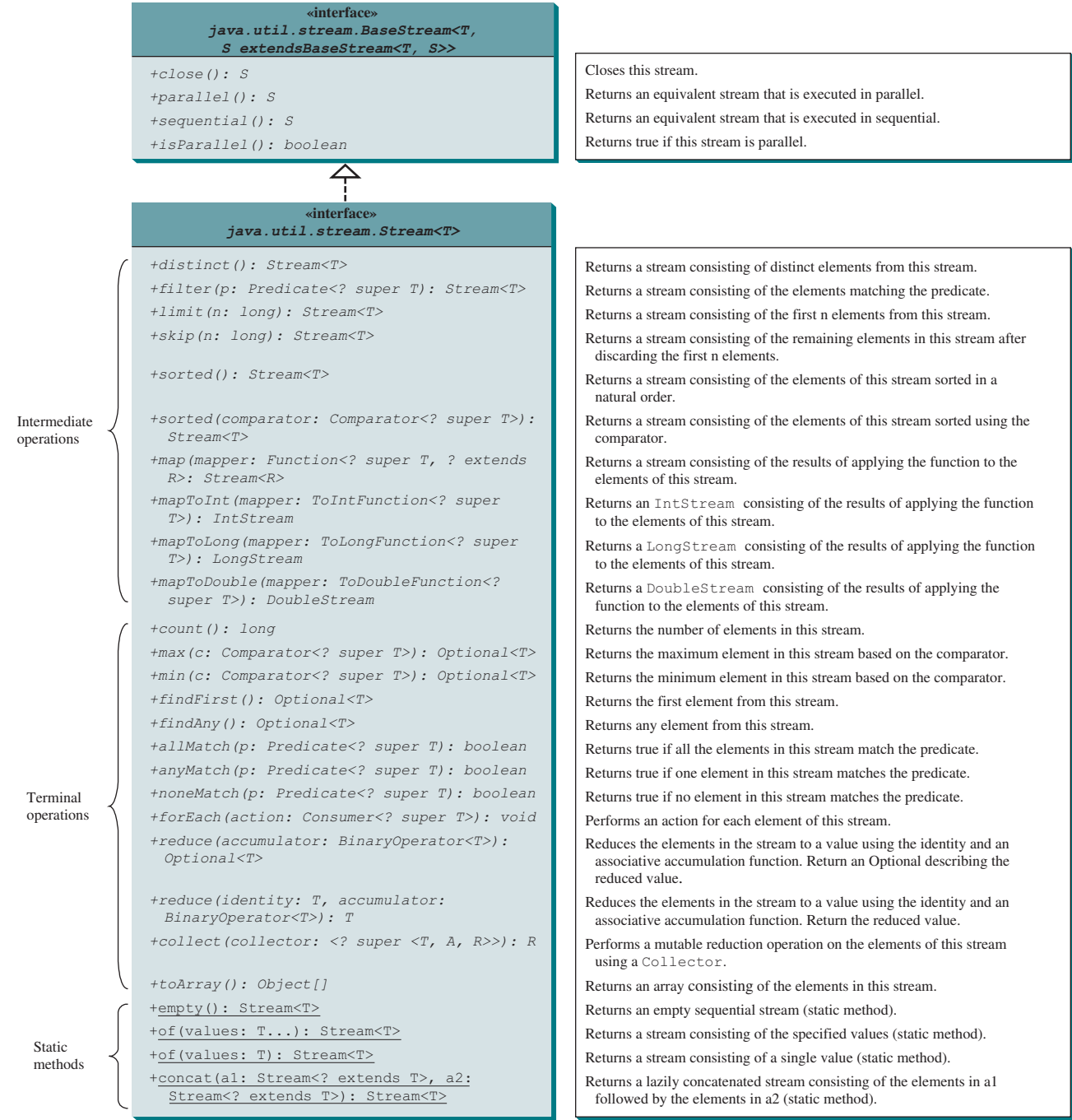


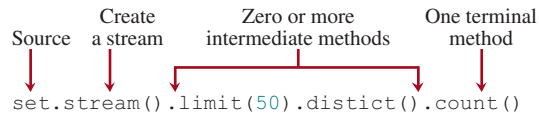
FIGURE 30.1 The `Stream` interface defines the aggregate operations for the elements in a stream.

The methods in the `Stream` interface are divided into three groups: intermediate methods, terminal methods, and static methods. An *intermediate method* transforms the stream into another stream. A *terminal method* returns a result or performs actions. After a terminal method is executed, the stream is closed automatically. A *static method* creates a stream.

intermediate method
terminal method
static method

stream pipeline

The methods are invoked using a stream pipeline. A *stream pipeline* consists of a source (e.g., a list, a set, or an array), a method that creates a stream, zero or more intermediate methods, and a final terminal method. The following is a stream pipeline example:



Here, `set` is the source of the data, invoking `stream()` creates a stream for the data from the source, invoking `limit(50)` returns the first 50 elements from the stream, invoking `distinct()` obtains a stream of distinct elements from the stream of the 50 elements, and invoking `count()` returns the number of elements in the final stream.

lazy evaluation

Streams are lazy, which means that the computation is performed only when the terminal operation is initiated. This allows the JVM to optimize computation.

Most of the arguments for stream methods are instances of functional interfaces. So the arguments can be created using lambda expressions or method references. Listing 30.1 gives an example that demonstrates creating a stream and applying methods on the streams.

LISTING 30.1 StreamDemo.java

```

1  import java.util.stream.Stream;
2
3  public class StreamDemo {
4      public static void main(String[] args) {
5          String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
6                          "George", "Alan", "Stacy", "Michelle", "john"};
7
8          // Display the first four names sorted
9          Stream.of(names).limit(4).sorted()
10             .forEach(e -> System.out.print(e + " "));
11
12          // Skip four names and display the rest sorted ignore case
13          System.out.println();
14          Stream.of(names).skip(4)
15             .sorted((e1, e2) -> e1.compareToIgnoreCase(e2))
16             .forEach(e -> System.out.print(e + " "));
17
18          System.out.println();
19          Stream.of(names).skip(4)
20             .sorted(String::compareToIgnoreCase)
21             .forEach(e -> System.out.print(e + " "));
22
23          System.out.println("\nLargest string with length > 4: "
24             + Stream.of(names)
25                 .filter(e -> e.length() > 4)
26                 .max(String::compareTo).get());
27
28          System.out.println("Smallest string alphabetically: "
29             + Stream.of(names).min(String::compareTo).get());
30
31          System.out.println("Stacy is in names? "
32             + Stream.of(names).anyMatch(e -> e.equals("Stacy")));
33
34          System.out.println("All names start with a capital letter? "
35             + Stream.of(names)
36                 .allMatch(e -> Character.isUpperCase(e.charAt(0))));
37

```

create an array

Stream.of
forEach

skip
sorted

method reference

filter
method reference

min

anyMatch

allMatch

```

38     System.out.println("No name begins with Ko? "
39         + Stream.of(names).noneMatch(e -> e.startsWith("Ko")));
40
41     System.out.println("Number of distinct case-insensitive strings: "
42         + Stream.of(names).map(e -> e.toUpperCase())
43             .distinct().count());
44
45     System.out.println("First element in this stream in lowercase: "
46         + Stream.of(names).map(String::toLowerCase).findFirst().get());
47
48     System.out.println("Skip 4 and get any element in this stream:"
49         + Stream.of(names).skip(4).sorted().findAny().get());
50
51     Object[] namesInLowerCase =
52         Stream.of(names).map(String::toLowerCase).toArray();
53     System.out.println(java.util.Arrays.toString(namesInLowerCase));
54 }
55 }

```

distinct
findFirst
findAny
toArray
display array

```

John Kim Peter Susan
Alan George Jen john Michelle Stacy
Alan George Jen john Michelle Stacy
Largest string with length > 4: Susan
Smallest string alphabetically: Alan
Stacy is in names? true
All names start with a capital letter? false
No name begins with Ko? true
Number of distinct case-insensitive strings: 9
First element in this stream in lowercase: john
Skip 4 and get any element in this stream: Alan
[john, peter, susan, kim, jen, george, alan, stacy, michelle, john]

```



30.2.1 The Stream.of, limit, and forEach Methods

The program creates an array of strings (lines 5–6). In lines 9–10, invoking the static **Stream.of(names)** returns a **Stream** consisting of strings from the **names** array, invoking **limit(4)** returns a new **Stream** consisting the first four elements in the stream, invoking **sorted()** sorts the stream, and invoking the **forEach** method displays each element in the stream. The argument passed to the **forEach** method is a lambda expression. As introduced in Section 15.6, a lambda expression is a concise syntax to replace an anonymous inner class that implements a functional interface. The argument passed to the **forEach** method is an instance of the functional interface **Consumer<? super T>** with an abstract function **accept(T t)**. The statement in line 10 using a lambda expression in (a) is equivalent to using an anonymous inner class in (b) as shown below. The lambda expression not only simplifies the code, but also the concept of the method. You can now simply say that for each element in the stream, perform the action as specified in the expression.

forEach method

lambda expression

```
forEach(e -> System.out.print(e + " "))
```

(a) Using a lambda expression

```

forEach(
    new java.util.function.Consumer<String>() {
        public void accept(String e) {
            System.out.print(e + " ");
        }
    }
)

```

(b) Using an anonymous inner class

30.2.2 The sorted Method

The `sorted` method in line 15 sorts the strings in the stream using a `Comparator`. The `Comparator` is a functional interface. A lambda expression is used to implement the interface and specifies that two strings are compared ignoring cases. This lambda expression in (a) is equivalent to the code using an anonymous inner class in (b). The lambda expression simply invokes a method in this case. So, it can be further simplified using a method reference in line 20 (also see in (c)). The method reference was introduced in Section 20.6.

method reference

```
sorted((e1, e2) ->
    e1.compareToIgnoreCase(e2))
```

(a) Using a lambda expression

```
sorted(String::compareToIgnoreCase)
```

(c) Using a method reference

```
sorted(
    new java.util.Comparator<String>() {
        public int compare(String e1, String e2) {
            return e1.compareToIgnoreCase(e2);
        }
    }
)
```

(b) Using an anonymous inner class

30.2.3 The filter Method

The `filter` method takes an argument of the `Predicate<? super T>` type, which is a functional interface with an abstract method `test(T t)` that returns a Boolean value. The method selects the elements from the stream that satisfies the predicate. Line 25 uses a lambda expression to implement the `Predicate` interface as shown in (a), which is equivalent to the code using an anonymous inner class as shown in (b).

```
filter(e -> e.length() > 4)
```

(a) Using a lambda expression

```
filter(
    new java.util.function.Predicate<String>() {
        public boolean test(String e) {
            return e.length() > 4;
        }
    }
)
```

(b) Using an anonymous inner class

30.2.4 The max and min Methods

The `max` and `min` methods take an argument of the `Comparator<? Super T>` type. This argument specifies how the elements are compared in order to obtain the maximum and minimum elements. The program uses the method reference `String::compareTo` to simplify the code for creating a `Comparator` (lines 26 and 29). The `max` and `min` methods return an `Optional<T>` that describes the element. You need to invoke the `get()` method from the `Optional` class to return the element.

Optional <T>
get method

30.2.5 The anyMatch, allMatch, and noneMatch Methods

The `anyMatch`, `allMatch`, and `noneMatch` methods take an argument of the `Predicate<? super T>` type to test if the stream contains an element, all elements, or no element that satisfies the predicate. The program tests whether the name `Stacy` is in the stream (line 32), whether all the names in the stream start with a capital letter (line 36), and whether any names starts with string `Ko` (line 39).

30.2.6 The map, distinct, and count Methods

The `map` method returns a new stream by mapping each element in the stream into a new element. So, the `map` method in line 42 returns a new stream with all uppercase strings. The `distinct()` method obtains a new stream with all distinct elements. The `count()` method

counts the number of the elements in the stream. So, the stream pipeline in line 43 counts the number of distinct strings in the array `names`.

The `map` method takes an argument of the `Function<? super T, ? super R>` type to return an instance of the `Stream<R>`. The `Function` is a functional interface with an abstract method `apply(T t)` that maps `t` into a value of the type `R`. Line 42 uses a lambda expression to implement the `Function` interface as shown in (a), which is equivalent to the code using an anonymous inner class as shown in (b). You can further simplify it using a method reference as shown in (c).

```
map(e -> e.toUpperCase())
```

(a) Using a lambda expression

```
map(String::toUpperCase)
```

(c) Using a method reference

```
map(
    new java.util.function.Function<String, String>() {
        public String apply(String e) {
            return e.toUpperCase();
        }
    }
)
```

(b) Using an anonymous inner class

30.2.7 The `findFirst`, `findAny`, and `toArray` Methods

The `findFirst()` method (line 46) returns the first element in the stream wrapped in an instance of `Optional<T>`. The actual element value is then returned by invoking the `get()` method in the `Optional<T>` class. The `findAny()` method (line 49) returns any element in the stream. Which element is selected depends on the internal state of the stream. The `findAny()` method is more efficient than the `findFirst()` method.

The `toArray()` method (line 52) returns an array of objects from the stream.



Note

The `BaseStream` interface defines the `close()` method, which can be invoked to close a stream. You don't need to use it because the stream is automatically closed after the terminal method is executed.

`close` method

30.2.1 Show the output of the following code:

```
Character[] chars = {'D', 'B', 'A', 'C'};
System.out.println(Stream.of(chars).sorted().findFirst().get());
System.out.println(Stream.of(chars).sorted(
    java.util.Comparator.reverseOrder()).findFirst().get());
System.out.println(Stream.of(chars)
    .limit(2).sorted().findFirst().get());
System.out.println(Stream.of(chars).distinct()
    .skip(2).filter(e -> e > 'A').findFirst().get());
System.out.println(Stream.of(chars)
    .max(Character::compareTo).get());
System.out.println(Stream.of(chars)
    .max(java.util.Comparator.reverseOrder()).get());
System.out.println(Stream.of(chars)
    .filter(e -> e > 'A').findFirst().get());
System.out.println(Stream.of(chars)
    .allMatch(e -> e >= 'A'));
System.out.println(Stream.of(chars)
    .anyMatch(e -> e > 'F'));
System.out.println(Stream.of(chars)
    .noneMatch(e -> e > 'F'));
Stream.of(chars).map(e -> e + "").map(e -> e.toLowerCase())
    .forEach(System.out::println);
```



Check
Point

```
Object[] temp = Stream.of(chars).map(e -> e + "Y")
    .map(e -> e.toLowerCase()).sorted().toArray();
System.out.println(java.util.Arrays.toString(temp));
```

30.2.2 What is wrong in the following code?

```
Character[] chars = {'D', 'B', 'A', 'C'};
Stream<Character> stream = Stream.of(chars).sorted();
System.out.println(stream.findFirst());
System.out.println(stream.skip(2).findFirst());
```

30.2.3 Rewrite (a) using a method reference and an anonymous inner class and (b) using lambda expression and an anonymous inner class:

```
(a) sorted((s1, s2) -> s1.compareToIgnoreCase(s2))
(b) forEach(System.out::println)
```

30.2.4 Given a `map` of the type `Map<String, Double>`, write an expression that returns the sum of all the values in `map`. For example, if the `map` contains `{"john", 1.5}` and `{"Peter", 1.1}`, the sum is `2.6`.

30.3 `IntStream`, `LongStream`, and `DoubleStream`

`IntStream`, `LongStream`, and `DoubleStream` are special type of streams for processing a sequence of primitive `int`, `long`, and `double` values.

Stream represents a sequence of objects. In addition to **Stream**, Java provides **IntStream**, **LongStream**, and **DoubleStream** for representing a sequence of `int`, `long`, and `double` values. These streams are also subinterfaces of **BaseStream**. You can use these streams in the same way like a **Stream**. Additionally, you can use the `sum()`, `average()`, and `summaryStatistics()` methods for returning the sum, average, and various statistics of the elements in the stream. You can use the `mapToInt` method to convert a **Stream** to an **IntStream** and use the `map` method to convert any stream including an **IntStream** to a **Stream**.

Listing 30.2 gives an example of using the **IntStream**.

LISTING 30.2 `IntStreamDemo.java`

```
1 import java.util.IntSummaryStatistics;
2 import java.util.stream.IntStream;
3 import java.util.stream.Stream;
4
5 public class IntStreamDemo {
6     public static void main(String[] args) {
7         int[] values = {3, 4, 1, 5, 20, 1, 3, 3, 4, 6};
8
9         System.out.println("The average of distinct even numbers > 3: " +
10             IntStream.of(values).distinct()
11                 .filter(e -> e > 3 && e % 2 == 0).average().getAsDouble());
12
13         System.out.println("The sum of the first 4 numbers is " +
14             IntStream.of(values).limit(4).sum());
15
16         IntSummaryStatistics stats =
17             IntStream.of(values).summaryStatistics();
18
19         System.out.printf("The summary of the stream is\n%-10s%10d\n" +
20             "%-10s%10d\n%-10s%10d\n%-10s%10d\n%-10s%10.2f\n",
```



`IntStream`
`LongStream`
`DoubleStream`

create an int array

`IntStream.of`
average

sum

summaryStatistics


```

21     " Count:", stats.getCount(), " Max:", stats.getMax(),
22     " Min:", stats.getMin(), " Sum:", stats.getSum(),
23     " Average:", stats.getAverage());
24
25     String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
26     "George", "Alan", "Stacy", "Michelle", "john"};
27
28     System.out.println("Total character count for all names is "
29     + Stream.of(names).mapToInt(e -> e.length()).sum());
30
31     System.out.println("The number of digits in array values is " +
32     Stream.of(values).map(e -> e + "").
33     .mapToInt(e -> e.length()).sum());
34 }
35 }

```

mapToInt
map
mapToInt

```

The average of distinct even numbers > 3: 10.0
The sum of the first 4 numbers is 13
The summary of the stream is

```

```

Count:      10
Max:        20
Min:         1
Sum:         50
Average:    5.00

```

```

Total character count for all names is 47
The number of digits in array values is 11

```



The program creates an array of `int` values (line 7). The stream pipeline in line 10 applies the intermediate methods `distinct` and `filter` with a terminal method `average`. The `average()` method returns the average value in the stream as an `OptionalDouble` object (line 11). The actual average value is obtained by invoking the `getAsDouble()` method.

`OptionalDouble` class
`average` method

The stream pipeline in line 14 applies the intermediate method `limit` with a terminal method `sum`. The `sum()` method returns the sum of all values in the stream.

`sum` method

If you need to obtain multiple summary values from the stream, using the `summaryStatistics()` method is more efficient. This method (line 17) returns an instance of `IntSummaryStatistics` that contains summary values for count, min, max, sum, and average (lines 19–23). Note `sum()`, `average()`, and `summaryStatistics()` methods are only applicable to the `IntStream`, `LongStream`, and `DoubleStream`.

`summaryStatistics`
method

The `mapToInt` method returns an `IntStream` by mapping each element in the stream into an `int` value. The `mapToInt` method in the stream pipeline in line 29 maps each string into an `int` value that is the length of a string, and the `sum` method obtains the sum of all the `int` values in the `IntStream`. The stream pipeline in line 29 obtains the total count for all characters in the stream.

The `mapToInt` method takes an argument of the `ToIntFunction<? super T>` type to return an instance of the `IntStream`. The `ToIntFunction` is a functional interface with an abstract method `applyAsInt(T t)` that maps `t` into a value of the type `int`. Line 33 uses a lambda expression to implement the `ToIntFunction` interface as shown in (a), which is equivalent to the code using an anonymous inner class as shown in (b). You can also further simplify it using a method reference as shown in (c).

`mapToInt` method

```
mapToInt(e -> e.length())
```

(a) Using a lambda expression

```
mapToInt(String::length)
```

(c) Using a method reference

```
mapToInt(
    new java.util.function.ToIntFunction<String>() {
        public int applyAsInt(String e) {
            return e.length();
        }
    }
)
```

(b) Using an anonymous inner class

The `map` method in line 32 returns a new stream of strings. Each string is converted from an integer in array `values`. The `mapToInt` method in line 33 returns a new stream of integers. Each integer represents the length of a string. The `sum()` method returns the sum of all `int` values in the final stream. So the stream pipeline in lines 32–33 obtains the total number of digits in the array `values`.



30.3.1 Show the output of the following code:

```
int[] numbers = {1, 4, 2, 3, 1};
System.out.println(IntStream.of(numbers)
    .sorted().findFirst().getAsInt());
System.out.println(IntStream.of(numbers)
    .limit(2).sorted().findFirst().getAsInt());
System.out.println(IntStream.of(numbers).distinct()
    .skip(1).filter(e -> e > 2).sum());
System.out.println(IntStream.of(numbers).distinct()
    .skip(1).filter(e -> e > 2).average().getAsDouble());
System.out.println(IntStream.of(numbers).max().getAsInt());
System.out.println(IntStream.of(numbers).max().getAsInt());
System.out.println(IntStream.of(numbers)
    .filter(e -> e > 1).findFirst().getAsInt());
System.out.println(IntStream.of(numbers)
    .allMatch(e -> e >= 1));
System.out.println(IntStream.of(numbers)
    .anyMatch(e -> e > 4));
System.out.println(IntStream.of(numbers).noneMatch(e -> e > 4));
IntStream.of(numbers).mapToObj(e -> (char)(e + 50))
    .forEach(System.out::println);

Object[] temp = IntStream.of(numbers)
    .mapToObj(e -> (char)(e + 'A')).toArray();
System.out.println(java.util.Arrays.toString(temp));
```

30.3.2 What is wrong in the following code?

```
int[] numbers = {1, 4, 2, 3, 1};
DoubleSummaryStatistics stats =
    DoubleStream.of(numbers).summaryStatistics();
System.out.printf("The summary of the stream is\n%-10s%10d\n" +
    "%-10s%10.2f\n%-10s%10.2f\n%-10s%10.2f\n%-10s%10.2f\n",
    " Count:", stats.getCount(), " Max:", stats.getMax(),
    " Min:", stats.getMin(), " Sum:", stats.getSum(),
    " Average:", stats.getAverage());
```

30.3.3 Rewrite the following code that maps an `int` to a `Character` using an anonymous inner class:

```
mapToObj(e -> (char)(e + 50))
```

30.3.4 Show the output of the following code:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}};
System.out.println(Stream.of(m)
    .mapToInt(e -> IntStream.of(e).sum()).sum());
```

30.3.5 Given an array `names` in Listing 30.1, write the code to display the total number of characters in `names`.

30.4 Parallel Streams

Streams can be executed in parallel mode to improve performance.



The widespread use of multicore systems has created a revolution in software. In order to benefit from multiple processors, software needs to run in parallel. All stream operations can execute in parallel to utilize the multicore processors. The `stream()` method in the `Collection` interface returns a sequential stream. To execute operations in parallel, use the `parallelStream()` method in the `Collection` interface to obtain a parallel stream. Any stream can be turned into a parallel stream by invoking the `parallel()` method defined in the `BaseStream` interface. Likewise, you can turn a parallel stream into a sequential stream by invoking the `sequential()` method.

Intermediate methods can be further divided into *stateless* and *stateful* methods. A stateless method such as `filter` and `map` can be executed independently from other elements in the stream. A stateful method such as `distinct` and `sorted` must be executed to take the entire stream into consideration. For example, to produce a result, the `distinct` method must consider all elements in the stream. Stateless methods are inherently parallelizable and can be executed in one pass in parallel. Stateful methods have to be executed in multiple passes in parallel.

stateless methods
stateful methods

Listing 30.3 gives an example to demonstrate the benefits of using parallel streams.

LISTING 30.3 ParallelStreamDemo.java

```
1  import java.util.Arrays;
2  import java.util.Random;
3  import java.util.stream.IntStream;
4
5  public class ParallelStreamDemo {
6      public static void main(String[] args) {
7          Random random = new Random();
8          int[] list = random.ints(200_000_000).toArray();
9
10         System.out.println("Number of processors: " +
11             Runtime.getRuntime().availableProcessors());
12
13         long startTime = System.currentTimeMillis();
14         int[] list1 = IntStream.of(list).filter(e -> e > 0).sorted()
15             .limit(5).toArray();
16         System.out.println(Arrays.toString(list1));
17         long endTime = System.currentTimeMillis();
18         System.out.println("Sequential execution time is " +
19             (endTime - startTime) + " milliseconds");
20
21         startTime = System.currentTimeMillis();
22         int[] list2 = IntStream.of(list).parallel().filter(e -> e > 0)
23             .sorted().limit(5).toArray();
24         System.out.println(Arrays.toString(list2));
25         endTime = System.currentTimeMillis();
26         System.out.println("Parallel execution time is " +
```

create an array

available processors

sequential stream

parallel stream

```

27         (endTime - startTime) + " milliseconds");
28     }
29 }

```



```

Number of processors: 8
[4, 9, 38, 42, 52]
Sequential execution time is 12362 milliseconds
[4, 9, 38, 42, 52]
Parallel execution time is 3448 milliseconds

```

The **Random** class introduced in Section 9.6.2 can be used to generate random numbers. You can use its **ints(n)** method to generate an **IntStream** consisting of **n** number of random **int** values (line 8). You can also use **ints(n, r1, r2)** to generate an **IntStream** with **n** element in the range from **r1** (inclusive) to **r2** (exclusive), use **doubles(n)** and **doubles(n, r1, r2)** to generate a **DoubleStream** of random floating-point numbers. Invoking the **toArray()** method on an **IntStream** (line 8) returns an array of **int** values from the stream. Recall that you can use underscores in an integer **200_000_000** to improve readability (see Section 2.10.1).

Invoking **Runtime.getRuntime()** returns a **Runtime** object (line 11). Invoking **Runtime** object's **availableProcessors()** returns the number of available processors for the JVM. In this case, the system has 8 processors.

An **IntStream** is created using **IntStream.of(list)** (line 14). The intermediate **filter(e -> e > 0)** method selects positive integers from the stream. The intermediate **sorted()** method sorts the filtered stream. The intermediate **limit(5)** method selects the first five integers in the sorted stream (line 15). Finally, the terminal method **toArray()** returns an array from the 5 integers in the stream. This is a sequential stream. To turn it into a parallel stream, simply invoke the **parallel()** method (line 22), which sets the stream for parallel execution. As you see from the sample run, the parallel execution is much faster than the sequential execution.

Several interesting questions arise.

lazy intermediate methods

1. The intermediate methods are lazy and are executed when the terminal method is initiated. This can be confirmed in the following code:

```

1     long startTime = System.currentTimeMillis();
2     IntStream stream = IntStream.of(list).filter(e -> e > 0).sorted()
3         .limit(5);
4     System.out.println("The time for the preceding method is " +
5         (System.currentTimeMillis() - startTime) + " milliseconds");
6     int[] list1 = stream.toArray();
7     System.out.println("The execution time is " +
8         (System.currentTimeMillis() - startTime) + " milliseconds");

```

When you run the code, you will see that almost no time is spent on lines 2–3 because the intermediate methods are not executed yet. When the terminal method **toArray()** is invoked, all the methods for the stream pipeline are executed. So, the actual execution time for the stream pipeline is in line 6.

order of methods

2. Does the order of the intermediate methods in a stream pipeline matter? Yes, it matters. For example, if the methods **limit(5)** and **sorted()** are swapped, the result will be different. It also matters to the performance even though the result is the same. For example, if the **sorted()** method is placed before **filter(e -> e > 0)**, the result will be the same, but it would take more time to execute the stream because sorting a large number of elements takes more time to complete. Applying **filter** before **sorted** would eliminate roughly half of the elements for sorting.

parallel vs. sequential streams

3. Is a parallel stream always faster? Not necessarily. Parallel execution requires synchronization, which carries some overhead. If you replace **IntStream.of(list)** in lines 14

and 22 by `random.ints(200_000_000)`, the parallel stream in lines 22–23 takes longer time to execute than the sequential stream in lines 14–15. The reason is that the algorithm for generating a sequence of pseudo-random numbers is highly sequential. The overhead of a parallel stream is far greater than the time saving on parallel processing. So, you should test both sequential and parallel streams before choosing parallel streams for deployment.

4. When executing a stream method in parallel, the elements in the stream may be processed in any order. So, the following code may display the numbers in the stream in a random order: order of parallel execution

```
IntStream.of(1, 2, 3, 4, 5).parallel()
    .forEach(e -> System.out.print(e + " "));
```

However, if it is executed sequentially, the numbers will be displayed as 1 2 3 4 5.

30.4.1 What is a stateless method? What is a stateful method?

30.4.2 How do you create a parallel stream?

30.4.3 Suppose `names` is a set of strings, which of the following two streams is better?

```
Object[] s = set.parallelStream().filter(e -> e.length() > 3)
    .sorted().toArray();
```

```
Object[] s = set.parallelStream().sorted()
    .filter(e -> e.length() > 3).toArray();
```

30.4.4 What will be the output of the following code?

```
int[] values = {3, 4, 1, 5, 20, 1, 3, 3, 4, 6};
System.out.print("The values are ");
IntStream.of(values)
    .forEach(e -> System.out.print(e + " "));
```

30.4.5 What will be the output of the following code?

```
int[] values = {3, 4, 1, 5, 20, 1, 3, 3, 4, 6};
System.out.print("The values are ");
IntStream.of(values).parallel()
    .forEach(e -> System.out.print(e + " "));
```

30.4.6 Write a statement to obtain an array of **1000** random double values between **0.0** and **1.0**, excluding **1.0**.



30.5 Stream Reduction Using the **reduce** Method

*You can use the **reduce** method to reduce the elements in a stream into a single value.*

Often you need to process all the elements in a collection to produce a summary value such as the sum, the maximum, or the minimum. For example, the following code obtains the sum of all elements in set `s`:

```
int total = 0;
for (int e: s) {
    total += e;
}
```

This is a simple code, but it specifies the exact steps on how to obtain the sum and it is highly sequential. The **reduce** method on a stream can be used to write the code in a high level for parallel execution. reduction

A *reduction* takes the elements from a stream to produce a single value by repeated application of a binary operation such as addition, multiplication, or finding the maximum between



two elements. Using reduction, you can write the code for finding the sum of all elements in a set as follows:

```
int sum = s.parallelStream().reduce(0, (e1, e2) -> e1 + e2);
```

Here, the **reduce** method takes two arguments. The first is an identity, that is, the starting value. The second argument is an object of the functional interface **IntBinaryOperator**. This interface contains the abstract method **applyAsInt(int e1, int e2)** that returns an **int** value from applying a binary operation. The preceding lambda expression in (a) is equivalent to the code using an anonymous inner class in (b).

```
reduce(0, e -> (e1, e2) -> e1 + e2)
```

(a) Using a lambda expression

```
reduce(0,
    new java.util.function.IntBinaryOperator() {
        public int applyAsInt(int e1, int e2) {
            return e1 + e2;
        }
    }
)
```

(b) Using an anonymous inner class

The preceding **reduce** method is semantically equivalent to an imperative code as follows:

```
int total = identity (i.e., 0, in this case);
for (int e: s) {
    total = applyAsInt(total, e);
}
```

The **reduce** method makes the code concise. Moreover, the code can be parallelizable, because multiple processors can simultaneously invoke the **applyAsInt** method on two integers repeatedly.

Using the **reduce** method, you can write the following code to return the maximum element in the set:

```
int result = s.parallelStream()
    .reduce(Integer.MIN_VALUE, (e1, e2) -> Math.max(e1, e2));
```

In fact, the **sum**, **max**, and **min** methods are implemented using the **reduce** method.

Listing 30.4 gives an example of using the **reduce** method.

LISTING 30.4 StreamReductionDemo.java

```
1 import java.util.stream.IntStream;
2 import java.util.stream.Stream;
3
4 public class StreamReductionDemo {
5     public static void main(String[] args) {
6         int[] values = {3, 4, 1, 5, 20, 1, 3, 3, 4, 6};
7
8         System.out.print("The values are ");
9         IntStream.of(values).forEach(e -> System.out.print(e + " "));
10
11         System.out.println("\nThe result of multiplying all values is " +
12             IntStream.of(values).parallel().reduce(1, (e1, e2) -> e1 * e2));
13
14         System.out.print("The values are " +
15             IntStream.of(values).mapToObj(e -> e + "")
16             .reduce((e1, e2) -> e1 + ", " + e2).get());
17
```

create an array

forEach

reduce

mapToObj

reduce


```

18     String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
19         "George", "Alan", "Stacy", "Michelle", "john"};
20     System.out.print("\nThe names are: ");
21     System.out.println(Stream.of(names)
22         .reduce((x, y) -> x + ", " + y).get());
23
24     System.out.print("Concat names: ");
25     System.out.println(Stream.of(names)
26         .reduce((x, y) -> x + y).get());
27
28     System.out.print("Total number of characters: ");
29     System.out.println(Stream.of(names)
30         .reduce((x, y) -> x + y).get().length());
31 }
32 }

```

The values are 3, 4, 1, 5, 20, 1, 3, 3, 4, 6
 The result of multiplying all values is 259200
 The values are 3, 4, 1, 5, 20, 1, 3, 3, 4, 6
 The names are John, Peter, Susan, Kim, Jen, George, Alan, Stacy,
 Michelle, john
 Concat names: JohnPeterSusanKimJenGeorgeAlanStacyMichellejohn
 Total number of characters: 47



The program creates an array of **int** values (line 6). The stream pipeline creates an **IntStream** from the **int** array and invokes the **forEach** method to display each integer in the stream (line 9).

The program creates a parallel stream pipeline for the **int** array and applies the **reduce** method to obtain the product of **int** values in the stream (line 12).

The **mapToObj** method returns a stream of string objects from the **IntStream** (line 15). The **reduce** method can be called without an identity. In this case, it returns an object of **Optional<T>**. The **reduce** method in line 16 reduces the strings in the stream into one composite string that consists of all strings in the stream separated by commas.

The **reduce** method in line 26 combines all strings in the stream together into one long string. **Stream.of(names).reduce((x, y) -> x + y).get()** returns a string that concatenates all strings in the stream into one string, and invoking the **length()** method on the string returns the number of characters in the string (line 30).

Note **reduce((x, y) -> x + y)** in line 30 can be simplified using a method reference as **reduce(String::concat)**.

30.5.1 Show the output of the following code:

```

int[] values = {1, 2, 3, 4};
System.out.println(IntStream.of(values)
    .reduce(0, (e1, e2) -> e1 + e2));
System.out.println(IntStream.of(values)
    .reduce(1, (e1, e2) -> e1 * e2));
System.out.println(IntStream.of(values).map(e -> e * e)
    .reduce(0, (e1, e2) -> e1 + e2));
System.out.println(IntStream.of(values).mapToObj(e -> "" + e)
    .reduce((e1, e2) -> e1 + " " + e2).get());
System.out.println(IntStream.of(values).mapToObj(e -> "" + e)
    .reduce((e1, e2) -> e1 + ", " + e2).get());

```



30.5.2 Show the output of the following code:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}};
System.out.println(Stream.of(m)
    .map(e -> IntStream.of(e).reduce(1, (e1, e2) -> e1 * e2))
    .reduce(1, (e1, e2) -> e1 * e2));
```

30.5.3 Show the output of the following code:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}, {1, 3}};
Stream.of(m).map(e -> IntStream.of(e))
    .reduce((e1, e2) -> IntStream.concat(e1, e2))
    .get().distinct()
    .forEach(e -> System.out.print(e + " "));
```

30.5.4 Show the output of the following code:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}, {1, 3}};
System.out.println(
    Stream.of(m).map(e -> IntStream.of(e))
        .reduce((e1, e2) -> IntStream.concat(e1, e2))
        .get().distinct().mapToObj(e -> e + "")
        .reduce((e1, e2) -> e1 + ", " + e2).get());
```

30.6 Stream Reduction Using the collect Method



*You can use the **collect** method to reduce the elements in a stream into a mutable container.*

In the preceding example, the **String**'s **concat** method is used in the **reduce** method for **Stream.of(names).reduce((x, y) -> x + y)**. This operation causes a new string to be created when concatenating two strings, which is very inefficient. A better approach is to use a **StringBuilder** and accumulate the result into a **StringBuilder**. This can be accomplished using the **collect** method.

The **collect** method collects the elements in a stream into a mutable container such as a **Collection** object using the following syntax:

```
<R> R collect(Supplier<R> supplier,
    BiConsumer<R, ? super T> accumulator,
    BiConsumer<R, R> combiner)
```

The method takes three functional arguments: 1) a supplier function to construct a new instance of the result container, 2) an accumulator function to incorporate the elements from the stream to the result container, and 3) a combining function to merge the contents of one result container into another.

For example, to combine strings into a **StringBuilder**, you may write the following code using a **collect** method like this:

```
String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
    "George", "Alan", "Stacy", "Michelle", "john"};
StringBuilder sb = Stream.of(names).collect(() -> new StringBuilder(),
    (c, e) -> c.append(e), (c1, c2) -> c1.append(c2));
```

The lambda expression **() -> new StringBuilder()** creates a **StringBuilder** object for storing the result, which can be simplified using the method reference **StringBuilder::new**. The lambda expression **(c, e) -> c.append(e)** adds a string **e** to a **StringBuilder** **c**, which can be simplified using a method reference **StringBuilder::append**.

The lambda expression `(c1, c2) -> c1.append(c2)` merges the contents in `c2` into `c1`, which also can be simplified using a method reference `StringBuilder::append`. So, you can simplify the preceding statement as follows:

```
StringBuilder sb = Stream.of(names).collect(StringBuilder::new,
    StringBuilder::append, StringBuilder::append);
```

The sequential `foreach` loop implementation for this `collect` method might be as follows:

```
StringBuilder sb = new StringBuilder();
for (String s: Stream.of(names)) {
    sb.append(s);
}
```

Note that the combiner function `(c1, c2) -> c1.append(c2)` is not used in the sequential implementation. It is used when the stream pipeline is executed in parallel. When executing a `collect` method in parallel, multiple result `StringBuilder` are created and then merged using a combiner function. So, the purpose of the combiner function is for parallel processing.

Here is another example that creates an `ArrayList` from strings in the stream:

```
ArrayList<String> list = Stream.of(names).collect(ArrayList::new,
    ArrayList::add, ArrayList::addAll);
```

The supplier function is the `ArrayList` constructor. The accumulator is the `add` method that adds an element to the `ArrayList`. The combiner function merges an `ArrayList` into another `ArrayList`. The three arguments—supplier, accumulator, and combiner—are tightly coupled and are defined using standard methods. For simplicity, Java provides another `collect` method that takes an argument of the `Collector` type, called a *collector*. The `Collector` interface defines the methods for returning a supplier, an accumulator, and a combiner. You can use a static factory method `toList()` in the `Collectors` class to create an instance of the `Collector` interface. So, the preceding statement can be simplified using a standard collector as follows:

```
List<String> list = Stream.of(names).collect(Collectors.toList());
```

Listing 30.5 gives an example of using the `collect` methods and the `Collectors`' factory methods.

LISTING 30.5 CollectDemo.java

```
1 import java.util.ArrayList;
2 import java.util.List;
3 import java.util.Map;
4 import java.util.Set;
5 import java.util.stream.Collectors;
6 import java.util.stream.Stream;
7
8 public class CollectDemo {
9     public static void main(String[] args) {
10         String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
11             "George", "Alan", "Stacy", "Michelle", "john"};           create an array
12         System.out.println("The number of characters for all names: " +
13             Stream.of(names).collect(StringBuilder::new,              collect into StringBuilder
14                 StringBuilder::append, StringBuilder::append).length());
15
16         List<String> list = Stream.of(names).collect(ArrayList::new,    collect into ArrayList
17             ArrayList::add, ArrayList::addAll);
18         System.out.println(list);
```

```

19
Collectors.toList() 20    list = Stream.of(names).collect(Collectors.toList());
21    System.out.println(list);
22
23    Set<String> set = Stream.of(names).map(e -> e.toUpperCase()).
Collectors.toSet() 24        collect(Collectors.toSet());
25    System.out.println(set);
26
27    Map<String, Integer> map = Stream.of(names).collect(
Collectors.toMap() 28        Collectors.toMap(e -> e, e -> e.length()));
29    System.out.println(map);
30
31    System.out.println("The total number of characters is " +
32        Stream.of(names).
sum of integers 33        collect(Collectors.summingInt(e -> e.length())));
summary information 34
35    java.util.IntSummaryStatistics stats = Stream.of(names).
36        collect(Collectors.summarizingInt(e -> e.length()));
37    System.out.println("Max is " + stats.getMax());
38    System.out.println("Min is " + stats.getMin());
39    System.out.println("Average is " + stats.getAverage());
40    }
41    }

```



```

The number of characters for all names: 47
[John, Peter, Susan, Kim, Jen, George, Alan, Stacy, Michelle, john]
[John, Peter, Susan, Kim, Jen, George, Alan, Stacy, Michelle, john]
[JEN, GEORGE, ALAN, SUSAN, JOHN, PETER, MICHELLE, KIM, STACY]
{Michelle=8, Stacy=5, Jen=3, George=6, Susan=5, Alan=4, John=4,
 john=4, Peter=5, Kim=3}
The total number of characters is 47
Max is 8
Min is 3
Average is 4.7

```

The program creates an array of strings, `names` (lines 10–11). The `collect` method (line 13) specifies a supplier (`StringBuilder::new`) for creating a `StringBuilder`, an accumulator (`StringBuilder::append`) for adding a string to the `StringBuilder`, and a combiner (`StringBuilder::append`) for combining two `StringBuilders` (lines 13–14). The stream pipeline obtains a `StringBuilder` that contains all strings in the stream. The `length()` method returns the length of the characters in the `StringBuilder`.

The `collect` method (line 16) specifies a supplier (`ArrayList::new`) for creating an `ArrayList`, an accumulator (`ArrayList::add`) for adding a string to the `ArrayList`, and a combiner (`ArrayList::addAll`) for combining two `ArrayLists` (lines 16–17). The stream pipeline obtains an `ArrayList` that contains all strings in the stream. This statement is simplified using a standard collector `Collectors.toList()` for a list (line 20).

The program creates a string stream, maps each string to uppercase (line 23), and creates a set using the `collect` method with a standard collector `Collectors.toSet()` for a set (lines 24). There are two uppercase strings `JOHN` in the stream. Since a set contains no duplicate elements, only one `JOHN` is stored in the set.

The program creates a string stream and creates a map using the `collect` method with a standard collector for a map (lines 28). The key of the map is the string and the value is the length of the string. Note key must be unique. If two strings are identical in the stream, a runtime exception would occur.

The **Collectors** class also contains the method for returning collectors that produce summary information. For example, **Collectors.summingInt** produces the sum of integer values in the stream (line 33), and **Collectors.summarizingInt** produces an **IntSummaryStatistics** for the integer values in the stream (lines 35–36).

30.6.1 Show the output of the following code:

```
int[] values = {1, 2, 3, 4, 1};
List<Integer> list = IntStream.of(values).mapToObj(e -> e)
    .collect(Collectors.toList());
System.out.println(list);

Set<Integer> set = IntStream.of(values).mapToObj(e -> e)
    .collect(Collectors.toSet());
System.out.println(set);

Map<Integer, Integer> map = IntStream.of(values).distinct()
    .mapToObj(e -> e)
    .collect(Collectors.toMap(e -> e, e -> e.hashCode()));
System.out.println(map);

System.out.println(
    IntStream.of(values).mapToObj(e -> e)
        .collect(Collectors.summingInt(e -> e)));

System.out.println(
    IntStream.of(values).mapToObj(e -> e)
        .collect(Collectors.averagingDouble(e -> e)));
```



30.7 Grouping Elements Using the **groupingBy** Collector

You can use the **groupingBy** collector along with the **collect** method to collect the elements by groups.

The elements in a stream can be divided into groups using the **groupingBy** collector and then applying aggregate collectors on each group. For example, you can group all the strings by their first letter and obtain the count of the elements in the group as follows:

```
String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
    "George", "Alan", "Stacy", "Steve", "john"};
Map<Character, Long> map = Stream.of(names).collect(
    Collectors.groupingBy(e -> e.charAt(0), Collectors.counting()));
```

The first argument in the **groupingBy** method specifies the criteria for grouping, known as a *classifier*. The second argument specifies how the elements in a group are processed, known as a group *processor*. A processor is commonly a summary collector such as **counting()**. Using the **groupingBy** collector, the **collect** method returns a map with the classifier as the key. You may also specify a supplier in the **groupingBy** method such as the following:

```
Map<Character, Long> map = Stream.of(names).collect(
    Collectors.groupingBy(e -> e.charAt(0),
        TreeMap::new, Collectors.counting()));
```

In this case, a tree map is used to store the map entries.

Listing 30.6 gives an example of using the **groupingBy** method.



Key
Point

group classifier
group processor

LISTING 30.6 CollectGroupDemo.java

```

1  import java.util.Map;
2  import java.util.TreeMap;
3  import java.util.stream.Collectors;
4  import java.util.stream.IntStream;
5  import java.util.stream.Stream;
6
7  public class CollectGroupDemo {
8      public static void main(String[] args) {
9          String[] names = {"John", "Peter", "Susan", "Kim", "Jen",
10             "George", "Alan", "Stacy", "Steve", "john"};
11
12         Map<String, Long> map1 = Stream.of(names).
13             map(e -> e.toUpperCase()).collect(
14                 Collectors.groupingBy(e -> e, Collectors.counting()));
15         System.out.println(map1);
16
17         Map<Character, Long> map2 = Stream.of(names).collect(
18             Collectors.groupingBy(e -> e.charAt(0), TreeMap::new,
19                 Collectors.counting()));
20         System.out.println(map2);
21
22         int[] values = {2, 3, 4, 1, 2, 3, 2, 3, 4, 5, 1, 421};
23         IntStream.of(values).mapToObj(e -> e).collect(
24             Collectors.groupingBy(e -> e, TreeMap::new,
25                 Collectors.counting())).
26             forEach((k, v) -> System.out.println(k + " occurs " + v +
27                 (v > 1 ? " times " : " time ")));
28
29         MyStudent[] students = {new MyStudent("John", "Lu", "CS", 32, 78),
30             new MyStudent("Susan", "Yao", "Math", 31, 85.4),
31             new MyStudent("Kim", "Johnson", "CS", 30, 78.1)};
32
33         System.out.printf("%10s%10s\n", "Department", "Average");
34         Stream.of(students).collect(Collectors.
35             groupingBy(MyStudent::getMajor, TreeMap::new,
36                 Collectors.averagingDouble(MyStudent::getScore))).
37             forEach((k, v) -> System.out.printf("%10s%10.2f\n", k, v));
38     }
39 }
40
41 class MyStudent {
42     private String firstName;
43     private String lastName;
44     private String major;
45     private int age;
46     private double score;
47
48     public MyStudent(String firstName, String lastName, String major,
49         int age, double score) {
50         this.firstName = firstName;
51         this.lastName = lastName;
52         this.major = major;
53         this.age = age;
54         this.score = score;
55     }
56
57     public String getFirstName() {
58         return firstName;

```

create an array

groupingBy

groupingBy

groupingBy

Student class


```

59     }
60
61     public String getLastName() {
62         return lastName;
63     }
64
65     public String getMajor() {
66         return major;
67     }
68
69     public int getAge() {
70         return age;
71     }
72
73     public double getScore() {
74         return score;
75     }
76 }

```

```

{JEN=1, ALAN=1, GEORGE=1, SUSAN=1, JOHN=2, STEVE=1, PETER=1, STACY=1,
 KIM=1}
{A=1, G=1, J=2, K=1, P=1, S=3, j=1}
1 occurs 2 times
2 occurs 3 times
3 occurs 3 times
4 occurs 2 times
5 occurs 1 time
421 occurs 1 time
Department    Average
           CS      78.05
           Math     85.40

```



The program creates a string stream, maps its elements to uppercase (lines 12–13), and collects the elements into a map with the string as the key and the occurrence of the string as the value (line 14). Note the **counting()** collector uses a value of the **Long** type. So, the **Map** is declared **Map<String, Long>**.

The program creates a string stream and collects the elements into a map with the first character in the string as the key and the occurrence of the first character as the value (lines 17–18). The key and value entries are stored in a **TreeMap**.

The program creates an **int** array (line 22). A stream is created from this array and the elements are mapped to **Integer** objects using the **mapToObj** method (line 23). The **collect** method returns a map with the **Integer** value as the key and the occurrence of the integer as the value (lines 24–25). The **forEach** method displays the key and value entries. Note the **collect** method is a terminal method. It returns an instance of **TreeMap** in this case. **TreeMap** has the **forEach** method for performing an action on each element in the collection.

The program creates an array of **Student** objects (lines 29–31). The **Student** class is defined with properties **firstName**, **lastName**, **major**, **age**, and **score** in lines 41–76. The program creates a stream for the array and the **collect** method groups the students by their major and returns a map with the major as the key and the average scores for the group as the value (lines 34–36). The method reference **MyStudent::getMajor** is used to specify the group by classifier. The method reference **TreeMap::new** specifies a supplier for the result map. The method reference **MyStudent::getScore** specifies the value for averaging.



30.7.1 Show the output of the following code:

```
int[] values = {1, 2, 2, 3, 4, 2, 1};
IntStream.of(values).mapToObj(e -> e).collect(
    Collectors.groupingBy(e -> e, TreeMap::new,
        Collectors.counting()))
    .forEach((k, v) -> System.out.println(k + " occurs " + v
        + (v > 1 ? " times " : " time ")));

IntStream.of(values).mapToObj(e -> e).collect(
    Collectors.groupingBy(e -> e, TreeMap::new,
        Collectors.summingInt(e -> e)))
    .forEach((k, v) -> System.out.println(k + ": " + v));

MyStudent[] students = {
    new MyStudent("John", "Johnson", "CS", 23, 89.2),
    new MyStudent("Susan", "Johnson", "Math", 21, 89.1),
    new MyStudent("John", "Peterson", "CS", 21, 92.3),
    new MyStudent("Kim", "Yao", "Math", 22, 87.3),
    new MyStudent("Jeff", "Johnson", "CS", 23, 78.5)};

Stream.of(students)
    .sorted(Comparator.comparing(MyStudent::getLastName)
        .thenComparing(MyStudent::getFirstName))
    .forEach(e -> System.out.println(e.getLastName() + ", " +
        e.getFirstName()));

Stream.of(students).collect(Collectors
    .groupingBy(MyStudent::getAge, TreeMap::new,
        Collectors.averagingDouble(MyStudent::getScore)))
    .forEach((k, v) -> System.out.printf("%10s%10.2f\n", k, v));
```

30.8 Case Studies



Many programs for processing arrays and collections can now be simplified and run faster using aggregate methods on streams.

You can write programs without using streams. However, using streams enables you to write shorter and simpler programs that can be executed faster in parallel by utilizing multiple processors. Many of the programs that involve arrays and collections in the early chapters can be simplified using streams. This section presents several case studies.

30.8.1 Case Study: Analyzing Numbers

Section 7.3 gives a program that prompts the user to enter values, obtains their average, and displays the number of values greater than the average. The program can be simplified using a **DoubleStream** as shown in Listing 30.7.

LISTING 30.7 AnalyzeNumbersUsingStream.java

```
1 import java.util.stream.*;
2
3 public class AnalyzeNumbersUsingStream {
4     public static void main(String[] args) {
5         java.util.Scanner input = new java.util.Scanner(System.in);
6         System.out.print("Enter the number of items: ");
7         int n = input.nextInt();
8         double[] numbers = new double[n];
9         double sum = 0;
```

create array

```

10
11     System.out.print("Enter the numbers: ");
12     for (int i = 0; i < n; i++) {
13         numbers[i] = input.nextDouble();           store number in array
14     }
15
16     double average = DoubleStream.of(numbers).average().getAsDouble();   get average
17     System.out.println("Average is " + average);
18     System.out.println("Number of elements above the average is "
19         + DoubleStream.of(numbers).filter(e -> e > average).count());   above average?
20 }
21 }

```

```

Enter the number of items: 10 ↵ Enter
Enter the numbers: 3.4 5 6 1 6.5 7.8 3.5 8.5 6.3 9.5 ↵ Enter
Average is 5.75
Number of elements above the average is 6

```



The program obtains the input from the user and stores the values in an array (lines 8–14), and obtains the average of the values using a stream (line 16), and finds the number of values greater than the average using a filtered stream (line 19).

30.8.2 Case Study: Counting the Occurrences of Each Letter

Listing 7.4, `CountLettersInArrays.java`, gives a program that randomly generates 100 lower-case letters and counts the occurrences of each letter.

The program can be simplified using a **Stream** as shown in Listing 30.8.

LISTING 30.8 `CountLettersUsingStream.java`

```

1  import java.util.Random;
2  import java.util.TreeMap;
3  import java.util.stream.Collectors;
4  import java.util.stream.Stream;
5
6  public class CountLettersUsingStream {
7      private static int count = 0;
8
9      public static void main(String[] args) {
10         Random random = new Random();
11         Object[] chars = random.ints(100, (int)'a', (int)'z' + 1).   create array
12             mapToObj(e -> (char)e).toArray();
13
14         System.out.println("The lowercase letters are:");
15         Stream.of(chars).forEach(e -> {                               display array
16             System.out.print(e + (++count % 20 == 0 ? "\n" : " "));
17         });
18
19         count = 0; // Reset the count for columns
20         System.out.println("\nThe occurrences of each letter are:");
21         Stream.of(chars).collect(Collectors.groupingBy(e -> e,
22             TreeMap::new, Collectors.counting())).forEach((k, v) -> {   count occurrence
23             System.out.print(v + " " + k
24                 + (++count % 10 == 0 ? "\n" : " "));
25         });
26     }
27 }

```



```
The lowercase letters are:
e y l s r i b k j v j h a b z n w b t v
s c c k r d w a m p w v u n q a m p l o
a z g d e g f i n d x m z o u l o z j v
h w i w n t g x w c d o t x h y v z y z
q e a m f w p g u q t r e n n w f c r f
```

```
The occurrences of each letter are:
5 a 3 b 4 c 4 d 4 e 4 f 4 g 3 h 3 i 3 j
2 k 3 l 4 m 6 n 4 o 3 p 3 q 4 r 2 s 4 t
3 u 5 v 8 w 3 x 3 y 6 z
```

The program generates a stream of 100 random integers. These integers are in the range between `(char) 'a'` and `(char) 'z'` (line 11). They are ASCII code for the lowercase letters. The `mapToObj` method maps the integers to their corresponding lowercase letters (line 12). The `toArray()` method returns an array consisting of these lowercase letters.

The program creates a stream of lowercase letters (line 15) and uses the `forEach` method to display each letter (line 16). The letters are displayed 20 per line. The static variable `count` is used to count the letters printed.

The program resets the count to 0 (line 19), creates a stream of lowercase letters (line 21), returns a map with lowercase letters as the key and the occurrences of each letter as the value (lines 21–22), and invokes the `forEach` method to display each key and value 10 per line (lines 23–24).

The code has 66 lines in Listing 7.4. The new code has only 27 lines, which greatly simplified coding. Furthermore, using streams is more efficient.

30.8.3 Case Study: Counting the Occurrences of Each Letter in a String

The preceding example randomly generates lowercase letters and counts the occurrence of each letter. This example counts the occurrence of each letter in a string. The program given in Listing 30.9 prompts the user to enter the string, converts all letters to uppercase, and displays the count of each letter in the string.

LISTING 30.9 CountOccurrenceOfLettersInAString.java

```
1 import java.util.*;
2 import java.util.stream.Stream;
3 import java.util.stream.Collectors;
4
5 public class CountOccurrenceOfLettersInAString {
6     private static int count = 0;
7
8     public static void main(String[] args) {
9         Scanner input = new Scanner(System.in);
10        System.out.print("Enter a string: ");
11        String s = input.nextLine();
12
13        count = 0; // Reset the count for columns
14        System.out.println("The occurrences of each letter are:");
15        Stream.of(toCharacterArray(s.toCharArray()))
16            .filter(ch -> Character.isLetter(ch))
17            .map(ch -> Character.toUpperCase(ch))
18            .collect(Collectors.groupingBy(e -> e,
```

read a string

convert char[] to
Character[]


select letters
map to uppercase
collect and count

```

19         TreeMap::new, Collectors.counting()))
20         .forEach((k, v) -> { System.out.print(v + " " + k
21             + (++count % 10 == 0 ? "\n" : " "));
22         });
23     }
24
25     public static Character[] toCharacterArray(char[] list) {
26         Character[] result = new Character[list.length];
27         for (int i = 0; i < result.length; i++) {
28             result[i] = list[i];
29         }
30         return result;
31     }
32 }

```

display value and key

Enter a string: Welcome to JavaAA 
 The occurrences of each letter are:
 4 A 1 C 2 E 1 J 1 L 1 M 2 O 1 T 1 V 1 W



The program reads a string `s` and obtains an array of `char` from the string by invoking `s.toCharArray()`. To create a stream of characters, you need to convert `char[]` into `Character[]`. So, the program defines the `toCharacterArray` method (lines 25–31) for obtaining a `Character[]` from `char[]`.

`char[]` to `Character[]`

The program creates a stream of `Character` object (line 15), eliminates nonletters from the stream using the `filter` method (line 16), converts all letters to uppercase using the `map` method (line 17), and obtains a `TreeMap` using the `collect` method (lines 18–19). In the `TreeMap`, the key is the letter and the value is the count for the letter. The `forEach` method (lines 20–21) in `TreeMap` is used to display the value and key.

30.8.3 Case Study: Processing All Elements in a Two-Dimensional Array

You can create a stream from a one-dimensional array. Can you create a stream for processing two-dimensional arrays? Listing 30.10 gives an example of processing two-dimensional arrays using streams.

LISTING 30.10 TwoDimensionalArrayStream.java

```

1  import java.util.IntSummaryStatistics;
2  import java.util.stream.IntStream;
3  import java.util.stream.Stream;
4
5  public class TwoDimensionalArrayStream {
6      private static int i = 0;
7      public static void main(String[] args) {
8          int[][] m = {{1, 2}, {3, 4}, {4, 5}, {1, 3}};
9
10         int[] list = Stream.of(m).map(e -> IntStream.of(e)).
11             reduce((e1, e2) -> IntStream.concat(e1, e2)).get().toArray();
12
13         IntSummaryStatistics stats =
14             IntStream.of(list).summaryStatistics();
15         System.out.println("Max: " + stats.getMax());
16         System.out.println("Min: " + stats.getMin());

```

create an array

reduce to one-dimensional

obtain statistical information

sum of each row

```

17     System.out.println("Sum: " + stats.getSum());
18     System.out.println("Average: " + stats.getAverage());
19
20     System.out.println("Sum of row ");
21     Stream.of(m).mapToInt(e -> IntStream.of(e).sum())
22         .forEach(e ->
23             System.out.println("Sum of row " + i++ + ": " + e));
24 }
25 }

```



```

Max: 5
Min: 1
Sum: 23
Average: 2.875
Sum of row
Sum of row 0: 3
Sum of row 1: 7
Sum of row 2: 9
Sum of row 3: 4

```

The program creates a two-dimensional array `m` in line 8. Invoking `Stream.of(m)` creates a stream consisting of rows as elements (line 10). The `map` method maps each row to an `IntStream`. The `reduce` method concatenates these streams into one large stream (line 11). This large stream now contains all the elements in `m`. Invoking `toArray()` from the stream returns an array consisting of all the integers in the stream (line 11).

The program obtains a statistical summary for an `IntStream` created from the array (lines 13–14) and displays the maximum, minimum, sum, and average of the integers in the stream (lines 15–18).

Finally, the program creates a stream from `m` that consists of rows in `m` (line 21). Each row is mapped to an `int` value using the `mapToInt` method (line 21). The `int` value is the sum of the elements in the row. The `forEach` method displays the sum for each row (lines 22–23).

30.8.4 Case Study: Finding the Directory Size

Listing 18.7 gives a recursive program that finds the size of a directory. The size of a directory is the sum of the sizes of all the files in the directory. The program can be implemented using streams as shown in Listing 30.11.

LISTING 30.11 DirectorySizeStream.java

```

1  import java.io.File;
2  import java.nio.file.Files;
3  import java.util.Scanner;
4
5  public class DirectorySizeStream {
6      public static void main(String[] args) throws Exception {
7          // Prompt the user to enter a directory or a file
8          System.out.print("Enter a directory or a file: ");
9          Scanner input = new Scanner(System.in);
10         String directory = input.nextLine();
11
12         // Display the size
13         System.out.println(getSize(new File(directory)) + " bytes");
14     }
15 }

```

invoke method


```

16  public static long getSize(File file) {
17      if (file.isFile()) {
18          return file.length();
19      }
20      else {
21          try {
22              return Files.list(file.toPath()).parallel().
23                  mapToLong(e -> getSize(e.toFile())).sum();
24          } catch (Exception ex) {
25              return 0;
26          }
27      }
28  }
29  }

```

getSize method
is file?
return file length

get subpaths
recursive call

Enter a directory or a file: c:\book ↵ Enter
48619631 bytes



Enter a directory or a file: c:\book\Welcome.java ↵ Enter
172 bytes



Enter a directory or a file: c:\book\NonExistentFile ↵ Enter
0 bytes



The program prompts the user to enter a file or a directory name (lines 8–10) and invokes the `getSize(File file)` method to return the size of the file or a directory (line 13). The `File` object can be a directory or a file. If it is a file, the `getSize` method returns the size of the file (lines 17–19). If it is a directory, invoking `Files.list(file.toPath())` returns a stream consisting of the `Path` objects. Each `Path` object represents a subpath in the directory (line 22). Since JDK 1.8, new methods are added in some existing classes to return streams. The `Files` class now has the static `list(Path)` method that returns a stream of subpaths in the path. For each subpath `e`, the `mapToLong` method maps `e` into the size of `e` by invoking `getSize(e)`. Finally, the terminal `sum()` method returns the size of the whole directory.

Both `getSize` methods in Listing 18.7 and Listing 30.9 are recursive. The `getSize` method in Listing 30.9 can be executed in a parallel stream. So, Listing 30.9 has better performance.

30.8.5 Case Study: Counting Keywords

Listing 21.7 gives a program to count the keywords in a Java source file. The program reads the words from a text file and tests if the word is a keyword. You can rewrite the code using streams as shown in Listing 30.12.

LISTING 30.12 CountKeywordStream.java

```

1  import java.util.*;
2  import java.io.*;
3  import java.nio.file.Files;
4  import java.util.stream.Stream;
5
6  public class CountKeywordStream {
7      public static void main(String[] args) throws Exception {
8          Scanner input = new Scanner(System.in);

```

```

9      System.out.print("Enter a Java source file: ");
enter a filename 10      String filename = input.nextLine();
11
12      File file = new File(filename);
file exists? 13      if (file.exists()) {
14          System.out.println("The number of keywords in " + filename
count keywords 15          + " is " + countKeywords(file));
16      }
17      else {
18          System.out.println("File " + filename + " does not exist");
19      }
20  }
21
22  public static long countKeywords(File file) throws Exception {
keywords 23      // Array of all Java keywords + true, false and null
24      String[] keywordString = {"abstract", "assert", "boolean",
25          "break", "byte", "case", "catch", "char", "class", "const",
26          "continue", "default", "do", "double", "else", "enum",
27          "extends", "for", "final", "finally", "float", "goto",
28          "if", "implements", "import", "instanceof", "int",
29          "interface", "long", "native", "new", "package", "private",
30          "protected", "public", "return", "short", "static",
31          "strictfp", "super", "switch", "synchronized", "this",
32          "throw", "throws", "transient", "try", "void", "volatile",
33          "while", "true", "false", "null"};
34
keyword set 35      Set<String> keywordSet =
36          new HashSet<>(Arrays.asList(keywordString));
37
38      return Files.lines(file.toPath()).parallel().mapToLong(line ->
count keyword 39          Stream.of(line.split("[\\s++]")).
40              filter(word -> keywordSet.contains(word)).count()).sum();
41  }
42  }

```



```

Enter a Java source file: c:\Welcome.java Enter
The number of keywords in c:\Welcome.java is 5

```



```

Enter a Java source file: c:\TTT.java Enter
File c:\TTT.java does not exist

```

The program prompts the user to enter a filename (lines 9–10). If the file exists, it invokes **countKeywords(file)** to return the number of keywords in the file (line 15). The keywords are stored in a set **keywordSet** (lines 35–36). Invoking **Files.lines(file.toPath())** returns a streams of lines form the file (line 38). Each line in the stream is mapped to a **long** value that counts the number of the keywords in the line. The line is split into an array of words using **line.split("[\\s++]")** and this array is used to create a stream (line 39). The **filter** method is applied to select the keyword from the stream. Invoking **count()** returns the number of the keywords in a line. The **sum()** method returns the total number of keywords in all lines (line 40).

The real benefit of using parallel streams in this example is for improving performance (line 38).

30.8.6 Case Study: Occurrences of Words

Listing 21.9 gives a program that counts the occurrences of words in a text. You can rewrite the code using streams as shown in Listing 30.13.

LISTING 30.13 CountOccurrenceOfWordsStream.java

```

1  import java.util.*;
2  import java.util.stream.Collectors;
3  import java.util.stream.Stream;
4
5  public class CountOccurrenceOfWordsStream {
6      public static void main(String[] args) {
7          // Set text in a string
8          String text = "Good morning. Have a good class. "
9                      + "Have a good visit. Have fun!";
10
11         Stream.of(text.split("[\\s+\\p{P}]")).parallel()
12             .filter(e -> e.length() > 0).collect(
13                 Collectors.groupingBy(String::toLowerCase, TreeMap::new,
14                     Collectors.counting()))
15             .forEach((k, v) -> System.out.println(k + " " + v));
16     }
17 }

```

text

split into words
filter empty words
group by words

display counts

a	2
class	1
fun	1
good	3
have	3
morning	1
visit	1



The text is split into words using a whitespace `\s` or punctuation `\p{P}` as a delimiter (line 11) and a stream is created for the words. The `filter` method is used to select nonempty words (line 12). The `collect` method groups the words by converting each into lowercase and returns a `TreeMap` with the lowercase words as the key and their count as the value (lines 13–14). The `forEach` method displays the key and its value (line 15).

The code in Listing 30.11 is about half in size as the code in Listing 21.9. The new program greatly simplifies coding and improves performance.

30.8.1 Can the following code be used to replace line 19 in Listing 30.7?

```

DoubleStream.of(numbers).filter(e -> e >
    DoubleStream.of(numbers).average()).count());

```



30.8.2 Can the following code be used to replace lines 15–16 in Listing 30.8?

```

Stream.of(chars).forEach(e -> {
    int count = 0;
    System.out.print(e + (++count % 20 == 0 ? "\n" : " ")); });

```

30.8.3 Show the output of the following code:

```

String s = "ABC";
Stream.of(s.toCharArray()).forEach(ch ->
    System.out.println(ch));

```

30.8.4 Show the output of the following code (The `toCharacterArray` method is presented in Listing 30.9.):

```
String s = "ABC";
Stream.of(toCharacterArray(s.toCharArray())) .forEach(ch ->
    System.out.println(ch));
```

30.8.5 Write the code to obtain a one-dimensional array `list` of strings from a two-dimensional array `matrix` of strings.

CHAPTER SUMMARY

1. Java 8 introduces aggregate methods on collection streams to simplify coding and improve performance.
2. A stream pipeline creates a stream from a data source, consists of zero or more intermediate methods (`skip`, `limit`, `filter`, `distinct`, `sorted`, `map`, and `mapToInt`), and a final terminal method (`count`, `sum`, `average`, `max`, `min`, `forEach`, `findFirst`, `firstAny`, `anyMatch`, `allMatch`, `noneMatch`, and `toArray`).
3. The execution of a stream is lazy, which means that the methods in the stream are not executed until the final terminal method is initiated.
4. The streams are transient objects. They are destroyed once the terminal method is executed.
5. The `Stream<T>` class defines the streams for a sequence of objects of the `T` type. `IntStream`, `LongStream`, and `DoubleStream` are the streams for a sequence of primitive `int`, `long`, and `double` values.
6. An important benefit of using streams is for performance. Streams can be executed in parallel mode to take advantages of multi-core architecture. You can switch a stream into a parallel or sequential mode by invoking the `parallel()` or `sequential()` method.
7. You can use the `reduce` method to reduce a stream into a single value and use the `collect` method to place the elements in the stream into a collection.
8. You can use the `groupingBy` collector to group the elements in the stream and apply aggregate methods for the elements in the group.



Quiz

Answer the quiz for this chapter online at the book Companion Website.

PROGRAMMING EXERCISES

- 30.1** (*Assign grades*) Rewrite Programming Exercise 7.1 using streams.
- 30.2** (*Count occurrence of numbers*) Rewrite Programming Exercise 7.3 using streams.
- 30.3** (*Analyze scores*) Rewrite Programming Exercise 7.4 using streams.

- 30.4** (*Print distinct numbers*) Rewrite Programming Exercise 7.5 using streams. Display the numbers in increasing order.
- 30.5** (*Count single digits*) Rewrite Programming Exercise 7.7 using streams.
- 30.6** (*Average an array*) Rewrite Programming Exercise 7.8 using streams.
- 30.7** (*Find the smallest element*) Rewrite Programming Exercise 7.9 using streams.
- 30.8** (*Eliminate duplicates*) Rewrite Programming Exercise 7.15 using streams and sort the elements in the new array in increasing order.
- 30.9** (*Sort students*) Rewrite Programming Exercise 7.17 using streams. Define a class named **Student** with data fields **name** and **score** and their getter methods. Store each student in a **Student** object.
- 30.10** (*Count the number of odd digits*) Write a program that prompts the user to enter any number as a string and displays the total number of odd digits in the string. Use **Stream**'s **reduce** method to count the odd digits.
- 30.11** (*Count the number of digits divisible by 5*) Write a program that prompts the user to enter any number as a string and displays the total number of digits that are divisible by 5 in the string. Use **Stream**'s **reduce** method to count the number of digits that are divisible by 5.
- 30.12** (*Sum the digits in an integer*) Rewrite Programming Exercise 6.2 using streams.
- 30.13** (*Count the letters in a string*) Rewrite Programming Exercise 6.20 using streams.
- 30.14** (*Occurrences of a specified character*) Rewrite Programming Exercise 6.23 using streams.
- 30.15** (*Display words in ascending alphabetical order*) Rewrite Programming Exercise 20.1 using streams.
- 30.16** (*Display distinct scores higher than average*) Use streams to write a program that displays distinct scores that are higher than the average scores in the **scores** array in Section 8.8. Display the scores in decreasing order, separated by a space.
- 30.17** (*Count consonants and vowels*) Rewrite Programming Exercise 21.4 using streams.
- 30.18** (*Count the occurrences of words in a text file*) Rewrite Programming Exercise 21.8 using streams.
- 30.19** (*Summary information*) Using the **scores** array in Section 8.8, write a program to obtain the average, maximum, and minimum scores of the students in the multiple-choice examination only, essay examination only, and both examinations.

This page is intentionally left blank

APPENDIXES

Appendix A

Java Keywords and Reserved Words

Appendix B

The ASCII Character Set

Appendix C

Operator Precedence Chart

Appendix D

Java Modifiers

Appendix E

Special Floating-Point Values

Appendix F

Number Systems

Appendix G

Bitwise Operations

Appendix H

Regular Expressions

Appendix I

Enumerated Types

Appendix J

The Big-O, Big-Omega, and Big-Theta Notations