Chapter 17
Lambdas and Streams

Java How to Program, 11/e

Questions? E-mail paul.deitel@deitel.com

OBJECTIVES

In this chapter you'll:

- Learn various functional-programming techniques and how they complement object-oriented programming.
- Use lambdas and streams to simplify tasks that process sequences of elements.
- Learn what streams are and how stream pipelines are formed from stream sources, intermediate operations and terminal operations.

OBJECTIVES (cont.)

- Create streams representing ranges of `int` values and random `int` values.
- Implement functional interfaces with lambdas.
- Perform on `IntStream`s intermediate operations `filter`, `map`, `mapToObj` and `sorted`, and terminal operations `forEach`, `count`, `min`, `max`, `sum`, `average` and `reduce`.

OBJECTIVES

- Perform on `Streams` intermediate operations `distinct`, `filter`, `map`, `mapToDouble` and `sorted`, and terminal operations `collect`, `forEach`, `findFirst` and `reduce`.
- Process infinite streams.
- Implement event handlers with lambdas.

17.1 Introduction

17.2 Streams and Reduction

17.2.1 Summing the Integers from 1 through 10 with a `for` Loop

17.2.2 External Iteration with `for` Is Error Prone

17.2.3 Summing with a Stream and Reduction

17.2.4 Internal Iteration

17.3 Mapping and Lambdas

17.3.1 Lambda Expressions

17.3.2 Lambda Syntax

17.3.3 Intermediate and Terminal Operations

17.4 Filtering

17.5 How Elements Move Through Stream Pipelines

17.6 Method References

17.6.1 Creating an `IntStream` of Random Values

17.6.2 Performing a Task on Each Stream Element with `forEach` and a Method Reference

17.6.3 Mapping Integers to String Objects with `mapToObj`

17.6.4 Concatenating Strings with `collect`

17.7 `IntStream` Operations

17.7.1 Creating an `IntStream` and Displaying Its Values

17.7.2 Terminal Operations `count`, `min`, `max`, `sum` and `average`

17.7.3 Terminal Operation `reduce`

17.7.4 Sorting `IntStream` Values

17.8 Functional Interfaces

17.9 Lambdas: A Deeper Look

17.10 Stream<Integer> Manipulations

17.10.1 Creating a Stream<Integer>
17.10.2 Sorting a Stream and Collecting the Results
17.10.3 Filtering a Stream and Storing the Results for Later Use
17.10.4 Filtering and Sorting a Stream and Collecting the Results
17.10.5 Sorting Previously Collected Results

17.11 Stream<String> Manipulations

17.11.1 Mapping Strings to Uppercase
17.11.2 Filtering Strings Then Sorting Them in Case-Insensitive Ascending Order
17.11.3 Filtering Strings Then Sorting Them in Case-Insensitive Descending Order

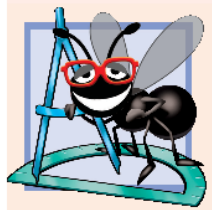
17.12 `Stream<Employee>` Manipulations

17.12.1 Creating and Displaying a `List<Employee>`
17.12.2 Filtering `Employee`s with Salaries in a Specified Range
17.12.3 Sorting `Employee`s By Multiple Fields
17.12.4 Mapping `Employee`s to Unique-Last-Name `String`s
17.12.5 Grouping `Employee`s By Department
17.12.6 Counting the Number of `Employee`s in Each Department
17.12.7 `Summing` and Averaging `Employee` Salaries

17.13 Creating a `Stream<String>` from a File
17.14 Streams of Random Values
17.15 Infinite Streams
17.16 Lambda Event Handlers
17.17 Additional Notes on Java SE 8 Interfaces
17.18 Wrap-Up

17.1 Introduction

- The way you think about Java programming is about to change profoundly
- Prior to Java SE 8, Java supported three programming paradigms—*procedural programming*, *object-oriented programming* and *generic programming*
- Java SE 8 added *lambdas* and *streams*—key technologies of *functional programming*
- In this chapter, we'll use lambdas and streams to write certain kinds of programs faster, simpler, more concisely and with fewer bugs than with previous techniques
- In Chapter 23, Concurrency, you'll see that such programs can be easier to *parallelize* (i.e., perform multiple operations simultaneously) so that you can take advantage of multi-core architectures to enhance performance—a key goal of lambdas and streams



Software Engineering Observation 17.1

You'll see in Chapter 23, Concurrency that it's hard to create parallel tasks that operate correctly if those tasks modify a program's state (that is, its variables' values). So the techniques that you'll learn in this chapter focus on **immutability**—not modifying the data source being processed or any other program state.

17.1 Introduction (cont.)

- This chapter presents many examples of lambdas and streams (Fig. 17.1), beginning with several showing better ways to implement tasks you programmed in Chapter 5
- The first several examples are presented in a manner that allows them to be covered in the context of earlier chapters
- For this reason, some terminology is discussed later in this chapter
- Figure 17.2 shows additional lambdas and streams coverage in later chapters

Section	May be covered after
Sections 17.2–17.4 introduce basic lambda and streams capabilities that process ranges of integers and eliminate the need for counter-controlled repetition.	Chapter 5, Control Statements: Part 2; Logical Operators
Section 17.6 introduces method references and uses them with lambdas and streams to process ranges of integers	Chapter 6, Methods: A Deeper Look
Section 17.7 presents lambda and streams capabilities that process one-dimensional arrays.	Chapter 7, Arrays and `ArrayLists`

Fig. 17.1 | This chapter's lambdas and streams discussions and examples. (Part 1 of 3.)

Section	May be covered after
Sections 17.8–17.9 discuss key functional interfaces and additional lambda concepts, and tie these into the chapter's earlier examples. Section 10.10 introduced Java SE 8's enhanced interface features (`default` methods, `static` methods and the concept of functional interfaces) that support functional-programming techniques in Java.	Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces
Section 17.16 shows how to use a lambda to implement a JavaFX event-listener functional interface.	Chapter 12, JavaFX Graphical User Interfaces: Part 1,
Section 17.11 shows how to use lambdas and streams to process collections of `String` objects.	Chapter 14, Strings, Characters and Regular Expressions

Fig. 17.1 | This chapter's lambdas and streams discussions and examples. (Part 2 of 3.)

Section	May be covered after
Section 17.13 shows how to use lambdas and streams to process lines of text from a file—the example in this section also uses some regular expression capabilities from Chapter 14.	Chapter 15, Files, Input/Output Streams, NIO and XML Serialization

Fig. 17.1 | This chapter's lambdas and streams discussions and examples. (Part 3 of 3.)

Coverage	Chapter
Uses lambdas to implement Swing event-listener functional interfaces.	Chapter 35, Swing GUI Components: Part 2
Shows that functional programs are easier to parallelize so that they can take advantage of multi-core architectures to enhance performance. Demonstrates parallel stream processing. Shows that `Arrays` method `parallelSort` can improve performance on multi-core vs. single-core architectures when sorting large arrays.	Chapter 23, Concurrency
Uses lambdas to implement Swing event-listener functional interfaces.	Chapter 26, Swing GUI Components: Part 1
Uses streams to process database query results.	Chapter 29, Java Persistence API (JPA)

Fig. 17.2 | Later lambdas and streams coverage.

17.2 Streams and Reduction

- [This section demonstrates how streams can be used to simplify programming tasks that you learned in Chapter 5, Control Statements: Part 2; Logical Operators.]
- In counter-controlled iteration, you typically determine *what* you want to accomplish then specify precisely *how* to accomplish it using a `for` loop
- In this section, we'll investigate that approach, then show you a better way to accomplish the same tasks

17.2.1 Summing the Integers from 1 through 10 with a `for` Loop

- Assume that *what* you want to accomplish is to sum the integers from 1 through 10
- In Chapter 5, you saw that you can do this with a counter-controlled loop:
  - ```
    int total = 0;

    for (int number = 1; number <= 10; number++) {
       total += number;
    }
    ```
- This loop specifies precisely *how* to perform the task—with a `for` statement that processes each value of control variable `number` from 1 through 10, adding `number`'s current value to `total` once per loop iteration and incrementing number after each addition operation
- Known as external iteration, because *you* specify all the iteration details
    ```

17.2.2 External Iteration with `for` Is Error Prone

- Let's consider potential problems with the preceding code
- As implemented, the loop requires two variables (`total` and `number`) that the code *mutates* (that is, modifies) during each loop iteration
- Every time you write code that modifies a variable, it's possible to introduce an error into your code
- There are several opportunities for error in the preceding code

17.2.2 External Iteration with `for` Is Error Prone (cont.)

- For example, you could:
  - initialize the variable `total` incorrectly
  - initialize the `for` loop's control variable `number` incorrectly
  - use the wrong loop-continuation condition
  - increment control variable `number` incorrectly or
  - incorrectly add each value of `number` to the `total`
- In addition, as the tasks you perform get more complicated, understanding *how* the code works gets in the way of understanding *what* it does
- This makes the code harder to read, debug and modify, and more likely to contain errors

17.2.3 Summing with a Stream and Reduction

- Let's specify *what* to do rather than *how* to do it
- In Fig. 17.3, we specify only *what* we want to accomplish—sum the integers from 1 through 10—then simply let Java's `IntStream` class (package `java.util.stream`) deal with *how* to do it
- The key to this program is the following expression in lines 9–10
  - ```
    IntStream.rangeClosed(1, 10)
             .sum()
    ```
- Read as, "for the stream of `int` values in the range 1 through 10, calculate the sum" or more simply "sum the numbers from 1 through 10." In this code, notice that there is neither a counter-control variable nor a variable to store the total—this is because `IntStream` conveniently defines `rangeClosed` and `sum`
    ```

```
1  // Fig. 17.3: StreamReduce.java
2  // Sum the integers from 1 through 10 with IntStream.
3  import java.util.stream.IntStream;
4
5  public class StreamReduce {
6     public static void main(String[] args) {
7        // sum the integers from 1 through 10
8        System.out.printf("Sum of 1 through 10 is: %d%n",
9           IntStream.rangeClosed(1, 10)
10                .sum());
11    }
12 }
```

```
Sum of 1 through 10 is: 55
```

Fig. 17.3 | Sum the integers from 1 through 10 with `IntStream`.

17.2.3 Summing with a Stream and Reduction (cont.)

Streams and Stream Pipelines

- The chained method calls in lines 9–10 create a stream pipeline
- A stream is a sequence of elements on which you perform tasks, and the stream pipeline moves the stream's elements through a sequence of tasks (or *processing steps*)

Good Programming Practice 17.1

When using chained method calls, align the dots (.) vertically for readability as we did in lines 9–10 of Fig. 17.3.

17.2.3 Summing with a Stream and Reduction (cont.)

Specifying the Data Source

- A stream pipeline typically begins with a method call that creates the stream—this is known as the *data source*
- Line 9 specifies the data source with the method call
  - `IntStream.rangeClosed(1, 10)`
  - Creates an `IntStream` representing an ordered range of `int` values
- Here, we use the `static` method `rangeClosed` to create an `IntStream` containing the ordered sequence of `int` elements `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` and `10`

17.2.3 Summing with a Stream and Reduction (cont.)

- The method is named `rangeClosed`, because it produces a *closed range* of values—that is, a range of elements that includes *both* of the method's arguments (`1` and `10`)
- `IntStream` also provides method `range`, which produces a *half-open range* of values from its first argument up to, but not including, its second argument—for example,
  - `IntStream.range(1, 10)`
- produces an `IntStream` containing the ordered sequence of `int` elements `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` and `9`, but *not* 10.

17.2.3 Summing with a Stream and Reduction (cont.)

Calculating the Sum of the `IntStream`'s Elements

- Next, line 10 completes the stream pipeline with the processing step
  - `.sum()`
- This invokes the `IntStream`'s `sum` instance method, which returns the sum of all the `int`s in the stream—in this case, the sum of the integers from 1 through 10
- The processing step performed by method `sum` is known as a reduction—it reduces the stream of values to a *single* value (the sum)
- This is one of several predefined `IntStream` reductions—Section 17.7 presents the predefined reductions `count`, `min`, `max`, `average` and `summaryStatistics`, as well as the `reduce` method for defining your own reductions

17.2.3 Summing with a Stream and Reduction (cont.)

Processing the Stream Pipeline

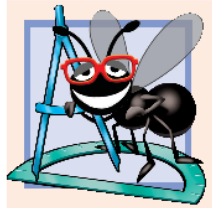
- A terminal operation initiates a stream pipeline's processing and produces a result
- `IntStream` method `sum` is a terminal operation that produces the sum of the stream's elements
- Similarly, the reductions `count`, `min`, `max`, `average`, `summaryStatistics` and `reduce` are all terminal operations

17.2.4 Internal Iteration

- The key to the preceding example is that it specifies *what* we want the task to accomplish—calculating the sum of the integers from 1 through 10—rather than *how* to accomplish it
- This is an example of declarative programming (specifying *what*) vs. imperative programming (specifying *how*)
- We broke the goal into two simple tasks—producing the numbers in a closed range (1–10) and calculating their sum
- Internally, the `IntStream` (that is, the data source itself) already knows how to perform each of these tasks

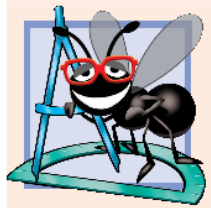
17.2.4 Internal Iteration (cont.)

- We did *not* need to specify *how* to iterate through the elements or declare and use *any* mutable variables
- This is known as internal iteration, because `IntStream` handles all the iteration details—a key aspect of functional programming
- Unlike external iteration with the `for` statement, the primary potential for error in line 9 of Fig. 17.3 is specifying the incorrect starting and/or ending values as arguments
- Once you're used to it, stream pipeline code also can be easier to read.



Software Engineering Observation 17.2

Functional-programming techniques enable you to write higher-level code, because many of the details are implemented for you by the Java streams library. Your code becomes more concise, which improves productivity and can help you rapidly prototype programs.



Software Engineering Observation 17.3

Functional-programming techniques eliminate large classes of errors, such as off-by-one errors (because iteration details are hidden from you by the libraries) and incorrectly modifying variables (because you focus on immutability and thus do not modify data). This makes it easier to write correct programs.

17.3 Mapping and Lambdas

- ***[This section demonstrates how streams can be used to simplify programming tasks that you learned in Chapter 5.]***
- Most stream pipelines also contain intermediate operations that specify tasks to perform on a stream's elements before a terminal operation produces a result.
- In this example, we introduce a common intermediate operation called mapping, which transforms a stream's elements to new values
- The result is a stream with the same number of elements containing the transformation's results
- Sometimes the mapped elements are of different types from the original stream's elements

17.3 Mapping and Lambdas (cont.)

- To demonstrate mapping, let's revisit the program of Fig. 5.5 in which we calculated the sum of the even integers from 2 through 20 using external iteration, as follows:

```
int total = 0;

for (int number = 2; number <= 20; number += 2) {
   total += number;
}
```

- Figure 17.4 reimplements this task using streams and internal iteration.

```
1  // Fig. 17.4: StreamMapReduce.java
2  // Sum the even integers from 2 through 20 with IntStream.
3  import java.util.stream.IntStream;
4
5  public class StreamMapReduce {
6     public static void main(String[] args) {
7        // sum the even integers from 2 through 20
8        System.out.printf("Sum of the even ints from 2 through 20 is: %d%n",
9           IntStream.rangeClosed(1, 10)                  // 1...10
10             .map((int x) -> {return x * 2;}) // multiply by 2
11             .sum());                         // sum
12    }
13 }
```

```
Sum of the even ints from 2 through 20 is: 110
```

Fig. 17.4 | Sum the even integers from 2 through 20 with `IntStream`.

17.3 Mapping and Lambdas (cont.)

- The stream pipeline in lines 9–11 performs three chained method calls:
  - Line 9 creates the data source—an `IntStream` containing the elements `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9` and `10`.
  - Line 10, which we'll discuss in detail momentarily, performs a processing step that maps each element (`x`) in the stream to that element multiplied by 2. The result is a stream of the even integers `2`, `4`, `6`, `8`, `10`, `12`, `14`, `16`, `18` and `20`.
  - Line 11 reduces the stream's elements to a single value—the `sum` of the elements. This is the terminal operation that initiates the pipeline's processing, then `sum`s the stream's elements.

17.3 Mapping and Lambdas (cont.)

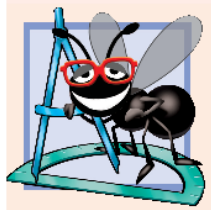
- The new feature here is the mapping operation in line 10, which in this case multiplies each stream element by 2
- `IntStream` method `map` receives as its argument (line 10)
  - `(int x) -> {return x * 2;}`
- You'll see in the next section is an alternate notation for "a *method* that receives an `int` parameter `x` and returns that value multiplied by `2`."
- For each element in the stream, `map` calls this method, passing to it the current stream element
- The method's return value becomes part of the new stream that `map` returns

17.3.1 Lambda Expressions

- As you'll see throughout this chapter, many intermediate and terminal stream operations receive methods as arguments
- Method `map`'s argument in line 10
  - `(int x) -> {return x * 2;}`
- is called a lambda expression (or simply a lambda), which represents an *anonymous method*—that is, a *method without a name*
- Though a lambda expression's syntax does not look like the methods you've seen previously, the left side does look like a method parameter list and the right side does look like a method body

17.3.1 Lambda Expressions (cont.)

- Lambda expressions enable you to create methods that can be treated as data
- You can:
  - pass lambdas as arguments to other methods (like `map`, or even other lambdas)
  - assign lambda expressions to variables for later use and
  - return lambda expressions from methods.
- You'll see that these are powerful capabilities



Software Engineering Observation 17.4

Lambdas and streams enable you to combine many benefits of functional-programming techniques with the benefits of object-oriented programming.

17.3.2 Lambda Syntax

- A lambda consists of a *parameter list* followed by the arrow token (`->`) and a body, as in:
  - `(`*parameterList*`) -> {`*statements*`}`
- Lambda in line 10 receives an `int`, multiplies its value by `2` and returns the result
  - `(int x) -> {return x * 2;}`
- In this case, the body is a *statement block* that may contain statements enclosed in curly braces
- The compiler *infers* from the lambda that it returns an `int`, because the parameter `x` is an `int` and the literal `2` is an `int`—multiplying an `int` by an `int` yields an `int` result
- Lambdas specify parameters in a comma-separated list

17.3.2 Lambda Syntax (cont.)

- The preceding lambda is similar to the following method but the lambda does not have a name and the compiler infers its return type

```
int multiplyBy2(int x) {
    return x * 2;
}
```

- There are several variations of the lambda syntax

17.3.2 Lambda Syntax (cont.)

Eliminating a Lambda's Parameter Type(s)

- A lambda's parameter type(s) usually may be omitted, as in:
  - `(x) -> {return x * 2;}`
- in which case, the compiler infers the parameter and return types by the lambda's context—we'll say more about this later
- If for any reason the compiler cannot infer the parameter or return types (e.g., if there are multiple type possibilities), it generates an error.

17.3.2 Lambda Syntax (cont.)

Simplifying the Lambda's Body

- If the body contains only one expression, the `return` keyword, curly braces and semicolon may be omitted, as in:
  - `(x) -> x * 2`
- In this case, the lambda *implicitly* returns the expression's value

17.3.2 Lambda Syntax (cont.)

Simplifying the Lambda's Parameter List

- If the parameter list contains only one parameter, the parentheses may be omitted, as in:
  - `x -> x * 2`

Lambdas with Empty Parameter Lists

- To define a lambda with an empty parameter list, use empty parentheses to the left of the arrow token (`->`), as in:
  - `() -> System.out.println("Welcome to lambdas!")`

Method References

- In addition, to the preceding lambda-syntax variations, there are specialized shorthand forms of lambdas that are known as *method references*, which we introduce in Section 17.6.

17.3.3 Intermediate and Terminal Operations

- In the stream pipeline shown in lines 9–11, `map` is an intermediate operation and `sum` is a terminal operation
- Method `map` is one of many intermediate operations that specify tasks to perform on a stream's elements

17.3.3 Intermediate and Terminal Operations (cont.)

Lazy and Eager Operations

- Intermediate operations use lazy evaluation—each intermediate operation results in a new stream object, but does not perform any operations on the stream's elements until a terminal operation is called to produce a result
- This allows library developers to optimize stream-processing performance
- For example, if you have 1,000,000 `Person` objects and you're looking for the *first* one with the last name `"Jones"`, rather than processing all 1,000,000 elements, stream processing can terminate as soon as the first matching `Person` object is found.

Performance Tip 17.1

Lazy evaluation helps improve performance by ensuring that operations are performed only if necessary.

17.3.3 Intermediate and Terminal Operations (cont.)

- Terminal operations are eager—they perform the requested operation when they're called
- We say more about lazy and eager operations as we encounter them throughout the chapter
- You'll see how lazy operations can improve performance in Section 17.5, which discusses how a stream pipeline's intermediate operations are applied to each stream element
- Figures 17.5 and 17.6 show some common intermediate and terminal operations, respectively

Common intermediate stream operations	
`filter`	Returns a stream containing only the elements that satisfy a condition (known as a *predicate*). The new stream often has fewer elements than the original stream.
`distinct`	Returns a stream containing only the unique elements—duplicates are eliminated.
`limit`	Returns a stream with the specified number of elements from the beginning of the original stream.
`map`	Returns a stream in which each of the original stream's elements is mapped to a new value (possibly of a different type)—for example, mapping numeric values to the squares of the numeric values or mapping numeric grades to letter grades (A, B C, D or F). The new stream has the same number of elements as the original stream.
`sorted`	Returns a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream. We'll show how to specify both ascending and descending order.

Fig. 17.5 | Common intermediate stream operations.

Common terminal stream operations	
`forEach`	Performs processing on every element in a stream (for example, display each element).
Reduction operations—*Take all values in the stream and return a single value*	
`average`	Returns the *average* of the elements in a numeric stream.
`count`	Returns the *number of elements* in the stream.
`max`	Returns the *maximum* value in a stream.
`min`	Returns the *minimum* value in a stream.
`reduce`	Reduces the elements of a collection to a *single value* using an associative accumulation function (for example, a lambda that adds two elements and returns the sum).

Fig. 17.6 | Common terminal stream operations.

17.4 Filtering

- ***[This section demonstrates how streams can be used to simplify programming tasks that you learned in Chapter 5]***
- Another common intermediate stream operation is *filtering* elements to select those that match a condition—known as a *predicate*
- For example, the following code selects the even integers in the range 1–10, multiplies each by 3 and sums the results:
  - ```
    int total = 0;

    for (int x = 1; x <= 10; x++) {
       if (x % 2 == 0) { // if x is even
          total += x * 3;
       }
    }
    ```
- Figure 17.7 reimplements this loop using streams
    ```

```
1  // Fig. 17.7: StreamFilterMapReduce.java
2  // Triple the even ints from 2 through 10 then sum them with IntStream.
3  import java.util.stream.IntStream;
4
5  public class StreamFilterMapReduce {
6     public static void main(String[] args) {
7        // sum the triples of the even integers from 2 through 10
8        System.out.printf(
9           "Sum of the triples of the even ints from 2 through 10 is: %d%n",
10          IntStream.rangeClosed(1, 10)
11                   .filter(x -> x % 2 == 0)
12                   .map(x -> x * 3)
13                   .sum());
14    }
15 }
```

```
Sum of the triples of the even ints from 2 through 10 is: 90
```

Fig. 17.7 | Triple the even `int`s from 2 through 10 then sum them with `IntStream`.

17.4 Filtering (cont.)

- The stream pipeline in lines 10–13 performs four chained method calls:
  - Line 10 creates the data source—an `IntStream` for the closed range 1 through 10.
  - Line 11, which we'll discuss in detail momentarily, filters the stream's elements by selecting only the elements that are divisible by 2 (that is, the even integers), producing a stream of the even integers from 2, 4, 6, 8 and 10.
  - Line 12 maps each element (`x`) in the stream to that element times 3, producing a stream of the even integers from 6, 12, 18, 24 and 30.
  - Line 13 reduces the stream to the sum of its elements (90).
- The new feature here is the filtering operation in line 11

17.4 Filtering (cont.)

- `IntStream` method `filter` receives as its argument a method that takes one parameter and returns a `boolean` result
- If the result is `true` for a given element, that element is included in the resulting stream
- The lambda in line 11 determines whether its `int` argument is divisible by `2` (that is, the remainder after dividing by `2` is `0`) and, if so, returns `true`; otherwise, the lambda returns `false`
  - `x -> x % 2 == 0`
- For each element in the stream, `filter` calls the method that it receives as an argument, passing to the method the current stream element
- If the method's return value is `true`, the corresponding element becomes part of the intermediate stream that `filter` returns

17.4 Filtering (cont.)

- Line 11 creates an intermediate stream representing only the elements that are divisible by `2`
- Line 12 uses `map` to create an intermediate stream representing the even integers (2, 4, 6, 8 and 10) that are multiplied by 3 (6, 12, 18, 24 and 30)
- Line 13 initiates the stream processing with a call to the *terminal* operation `sum`
- The combined processing steps are applied to each element, then `sum` returns the total of the elements that remain in the stream

Error-Prevention Tip 17.1

The order of the operations in a stream pipeline matters. For example, `filter`ing the even numbers from 1–10 yields 2, 4, 6, 8, 10, then `map`ping them to twice their values yields 4, 8, 12, 16 and 20. On the other hand, `map`ping the numbers from 1–10 to twice their values yields 2, 4, 6, 8, 10, 12, 14, 16, 18 and 20, then `filter`ing the even numbers gives all of those values, because they're all even before the `filter` operation is performed.

17.4 Filtering (cont.)

- The stream pipeline shown in this example could have been implemented by using only `map` and `sum`
- Exercise 17.18 asks you to eliminate the `filter` operation.

17.5 How Elements Move Through Stream Pipelines

- Each new stream is simply an object representing the processing steps that have been specified to that point in the pipeline
- Chaining intermediate-operation method calls adds to the set of processing steps to perform on each stream element
- The last stream object in the stream pipeline contains all the processing steps to perform on each stream element.

17.5 How Elements Move Through Stream Pipelines (cont.)

- When you initiate a stream pipeline with a terminal operation, the intermediate operations' processing steps are applied for a given stream element *before* they are applied to the next stream element
- So the stream pipeline in Fig. 17.7 operates as follows:
  - *For each element*
  - *If the element is an even integer*
  - *Multiply the element by 3 and add the result to the total*

17.5 How Elements Move Through Stream Pipelines (cont.)

- Consider a modified version of Fig. 17.7's stream pipeline in which each lambda displays the intermediate operation's name and the current stream element's value:

```
IntStream.rangeClosed(1, 10)
         .filter(
            x -> {
               System.out.printf("%nfilter: %d%n", x);
               return x % 2 == 0;
            })
         .map(
            x -> {
               System.out.println("map: " + x);
               return x * 3;
            })
         .sum()
```

17.5 How Elements Move Through Stream Pipelines (cont.)

- The modified pipeline's output (we added the comments) shows that each even integer's `map` step is applied *before* the next stream element's `filter` step:

```
filter: 1 // odd so no map step is performed for this element

filter: 2 // even so a map step is performed next
map: 2

filter: 3 // odd so no map step is performed for this element

filter: 4 // even so a map step is performed next
map: 4

filter: 5 // odd so no map step is performed for this element
```

17.5 How Elements Move Through Stream Pipelines (cont.)

- `filter: 6 // even so a map step is performed next`
- `map: 6`

- `filter: 7 // odd so no map step is performed for this element`

- `filter: 8 // even so a map step is performed next`
- `map: 8`

- `filter: 9 // odd so no map step is performed for this element`

- `filter: 10 // even so a map step is performed next`
- `map: 10`

17.5 How Elements Move Through Stream Pipelines (cont.)

- For the odd elements, the `map` step was *not* performed
- When a `filter` step returns `false`, the element's remaining processing steps are *ignored* because that element is not included in the results
- (This version of Fig. 17.7 is located in a subfolder with that example.)

17.6 Method References

- ***[This section demonstrates how streams can be used to simplify programming tasks that you learned in Chapter 6, Methods: A Deeper Look.]***
- For a lambda that simply calls another method, you can replace the lambda with that method's name—known as a method reference
- The compiler converts a method reference into an appropriate lambda expression
- Like Fig. 6.6, Fig. 17.8 uses `SecureRandom` to obtain random numbers in the range 1–6
- The program uses streams to create the random values and method references to help display the results
- We walk through the code in Sections 17.6.1–17.6.4

```
1  // Fig. 17.8: RandomIntegers.java
2  // Shifted and scaled random integers.
3  import java.security.SecureRandom;
4  import java.util.stream.Collectors;
5
6  public class RandomIntegers {
7     public static void main(String[] args) {
8        SecureRandom randomNumbers = new SecureRandom();
9
10       // display 10 random integers on separate lines
11       System.out.println("Random numbers on separate lines:");
12       randomNumbers.ints(10, 1, 7)
13                    .forEach(System.out::println);
```

Fig. 17.8 | Shifted and scaled random integers. (Part 1 of 3.)

```
14
15          // display 10 random integers on the same line
16          String numbers =
17             randomNumbers.ints(10, 1, 7)
18                          .mapToObj(String::valueOf)
19                          .collect(Collectors.joining(" "));
20          System.out.printf("%nRandom numbers on one line: %s%n", numbers);
21
22       }
23    }
```

Fig. 17.8 | Shifted and scaled random integers. (Part 2 of 3.)

```
Random numbers on separate lines:
4
3
4
5
1
5
5
3
6
5

Random numbers on one line: 4 6 2 5 6 4 3 2 4 1
```

Fig. 17.8 | Shifted and scaled random integers. (Part 3 of 3.)

17.6.1 Creating an `IntStream` of Random Values

- Class `SecureRandom`'s `ints` method returns an `IntStream` of random numbers
- `randomNumbers.ints(10, 1, 7)`
  - creates an `IntStream` data source with the specified number of random `int` values (10) in the range starting with the first argument (1) up to, but not including, the second argument (7)
- So, line 12 produces an `IntStream` of 10 random integers in the range 1–6.

17.6.2 Performing a Task on Each Stream Element with `forEach` and a Method Reference

- Next, line 13 of the stream pipeline uses `IntStream` method `forEach` (a terminal- operation) to perform a task on each stream element
- Method `forEach` receives as its argument a method that takes one parameter and performs a task using the parameter's value
- The argument to `forEach` in this case is a method reference—a shorthand notation for a lambda that calls the specified method
  - `System.out::println`

17.6.2 Performing a Task on Each Stream Element with `forEach` and a Method Reference (cont._

- A method reference of the following form is a bound instance method reference—"bound" means the *specific* object to the left of `::` (`System.out`) *must* be used to call the instance method to the right of `::` (`println`)
  - *objectName*`::`*instanceMethodName*
- The compiler converts `System.out::println` into a one-parameter lambda like the following that passes the lambda's argument to the `System.out` object's `println` instance method
  - `x -> System.out.println(x)`
- The stream pipeline of lines 12–13 is equivalent to the following `for` loop:

```
for (int i = 1; i <= 10; i++) {
   System.out.println(1 + randomNumbers.nextInt(6));
}
```

17.6.3 Mapping Integers to String Objects with `mapToObj`

- The stream pipeline in lines 16–19
  - `String numbers =`
  - `   randomNumbers.ints(10, 1, 7)`
  - `                  .mapToObj(String::valueOf)`
  - `                  .collect(Collectors.joining(" "));`
- creates a `String` containing 10 random integers in the range 1–6 separated by spaces

17.6.3 Mapping Integers to String Objects with `mapToObj` (cont.)

- The pipeline performs three chained method calls:
  - Line 17 creates the data source—an `IntStream` of 10 random integers from 1–6.
  - Line 18 maps each `int` to its `String` representation, resulting in an intermediate stream of `String`s. The `IntStream` method `map` that we've used previously returns another `IntStream`. To map to `String`s, we use instead the `IntStream` method `mapToObj`, which enables you to map from `int`s to a stream of reference-type elements-. Like `map`, `mapToObject` expects a one-parameter method that returns a result. In this example, `mapToObj`'s argument is a `static` method reference of the form *ClassName*`::`*staticMethodName*. The compiler converts `String::valueOf` (which returns its argument's `String` representation) into a one-parameter lambda that calls `valueOf`, passing the current stream element as an argument, as in
    - `x -> String.valueOf(x)`
  - Line 19, which we discuss in more detail in Section 17.6.4, uses the `Stream` terminal operation `collect` to concatenate all the `String`s, separating each from the next with a space. Method `collect` is a form of reduction because it returns one object—in this case, a `String`.
- Line 20 then displays the resulting `String`.

17.6.4 Concatenating Strings with `collect`

- Consider line 19 of Fig. 17.8
- `Stream` terminal operation `collect` uses a *collector* to gather the stream's elements into a single object—often a collection
- Similar to a reduction, but `collect` returns an object containing the stream's elements, whereas `reduce`- returns a single value of the stream's element type
- In this example, we use a predefined collector returned by the `static` `Collectors` method `joining`
- This collector creates a concatenated `String` representation of the stream's elements, appending each element to the `String` separated from the previous element by the `joining` method's argument (in this case, a space)
- Method `collect` then returns the resulting `String`
- We discuss other collectors throughout this chapter

17.7 `IntStream` Operations

- ***[This section demonstrates how lambdas and streams can be used to simplify programming tasks like those you learned in Chapter 7.]***
- Figure 17.9 demonstrates additional `IntStream` operations on streams created from arrays
- The `IntStream` techniques shown in this and the prior examples also apply to `LongStream`s and `DoubleStream`s for `long` and `double` values, respectively
- We walk through the code in Sections 17.7.1–17.7.4

```
1  // Fig. 17.9: IntStreamOperations.java
2  // Demonstrating IntStream operations.
3  import java.util.Arrays;
4  import java.util.stream.Collectors;
5  import java.util.stream.IntStream;
6
7  public class IntStreamOperations {
8     public static void main(String[] args) {
9        int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
10
11       // display original values
12       System.out.print("Original values: ");
13       System.out.println(
14          IntStream.of(values)
15                   .mapToObj(String::valueOf)
16                   .collect(Collectors.joining(" ")));
```

Fig. 17.9 | Demonstrating `IntStream` operations. (Part 1 of 4.)

```
17
18          // count, min, max, sum and average of the values
19          System.out.printf("%nCount: %d%n", IntStream.of(values).count());
20          System.out.printf("Min: %d%n",
21             IntStream.of(values).min().getAsInt());
22          System.out.printf("Max: %d%n",
23             IntStream.of(values).max().getAsInt());
24          System.out.printf("Sum: %d%n", IntStream.of(values).sum());
25          System.out.printf("Average: %.2f%n",
26             IntStream.of(values).average().getAsDouble());
27
```

Fig. 17.9 | Demonstrating `IntStream` operations. (Part 2 of 4.)

```
28          // sum of values with reduce method
29          System.out.printf("%nSum via reduce method: %d%n",
30             IntStream.of(values)
31                      .reduce(0, (x, y) -> x + y));
32
33          // product of values with reduce method
34          System.out.printf("Product via reduce method: %d%n",
35             IntStream.of(values)
36                      .reduce((x, y) -> x * y).getAsInt());
37
38          // sum of squares of values with map and sum methods
39          System.out.printf("Sum of squares via map and sum: %d%n%n",
40             IntStream.of(values)
41                      .map(x -> x * x)
42                      .sum());
```

Fig. 17.9 | Demonstrating `IntStream` operations. (Part 3 of 4.)

```
43
44          // displaying the elements in sorted order
45          System.out.printf("Values displayed in sorted order: %s%n",
46             IntStream.of(values)
47                      .sorted()
48                      .mapToObj(String::valueOf)
49                      .collect(Collectors.joining(" ")));
50       }
51    }
```

```
Original values: 3 10 6 1 4 8 2 5 9 7

Count: 10
Min: 1
Max: 10
Sum: 55
Average: 5.50

Sum via reduce method: 55
Product via reduce method: 3628800
Sum of squares via map and sum: 385

Values displayed in sorted order: 1 2 3 4 5 6 7 8 9 10
```

Fig. 17.9 | Demonstrating `IntStream` operations. (Part 4 of 4.)

17.7.1 Creating an `IntStream` and Displaying Its Values

- `IntStream` `static` method `of` (line 14) receives an `int` array argument and returns an `IntStream` for processing the array's values
- The stream pipeline in lines 14–16
  - `IntStream.of(values)`
  - `.mapToObj(String::valueOf)`
  - `.collect(Collectors.joining(" ")));`
- displays the stream's elements
- First, line 14 creates an `IntStream` for the `values` array, then lines 15–16 use the `mapToObj` and `collect` methods as shown Fig. 17.8 to obtain a `String` representation of the stream's elements separated by spaces

17.7.1 Creating an `IntStream` and Displaying Its Values (cont.)

- This example repeatedly creates an `IntStream` from the array `values` using:
  - `IntStream.of(values)`
- You might think that we could simply store the stream and reuse it
- However, once a stream pipeline is processed with a terminal operation, *the stream cannot be reused*, because it does not maintain a copy of the original data source

17.7.2 Terminal Operations `count`, `min`, `max`, `sum` and `average`

- Class `IntStream` provides various terminal operations for common stream reductions on streams of `int` values:
  - `count` (line 19) returns the number of elements in the stream.
  - `min` (line 21) returns an `OptionalInt` (package `java.util`) possibly containing the smallest `int` in the stream. For any stream, it's possible that there are *no elements* in the stream. Returning `OptionalInt` enables method `min` to return the minimum value if the stream contains *at least one element*. In this example, we know the stream has 10 elements, so we call class `OptionalInt`'s `getAsInt` method to obtain the minimum value. If there were no *elements*, the `OptionalInt` would not contain an `int` and `getAsInt` would throw a `NoSuchElementException`. To prevent this, you can instead call method `orElse`, which returns the `OptionalInt`'s value if there is one, or the value you pass to `orElse`, otherwise.

17.7.2 Terminal Operations `count`, `min`, `max`, `sum` and `average` (cont.)

- `max` (line 23) returns an `OptionalInt` possibly containing the largest `int` in the stream. Again, we call the `OptionalInt`'s `getAsInt` method to get the largest value, because we know this stream contains elements.
- `sum` (line 24) returns the sum of all the `int`s in the stream.
- `average` (line 26) returns an `OptionalDouble` (package `java.util`) possibly containing the average of the `int`s in the stream as a value of type `double`. In this example, we know the stream has elements, so we call class `Optional-Double`'s `getAsDouble` method to obtain the average. If there were no *elements*, the `OptionalDouble` would not contain the average and `getAsDouble` would throw a `NoSuchElementException`. As with `OptionalInt`, to prevent this exception, you can instead call method `orElse`, which returns the `OptionalDouble`'s value if there is one, or the value you pass to `orElse`, otherwise.

17.7.2 Terminal Operations `count`, `min`, `max`, `sum` and `average` (cont.)

- Class `IntStream` also provides method `summaryStatistics` that performs the `count`, `min`, `max`, `sum` and `average` operations *in one pass* of an `IntStream`'s elements and returns the results as an `IntSummaryStatistics` object (package `java.util`)
- Provides a significant performance boost over reprocessing an `IntStream` repeatedly for each individual operation
- This object has methods for obtaining each result and a `toString` method that summarizes all the results
- `System.out.println(IntStream.of(values).summaryStatistics());`
  - Produces for the array `values` in Fig. 17.9
    `IntSummaryStatistics{count=10, sum=55, min=1, average=5.500000, max=10}`

17.7.3 Terminal Operation `reduce`

- So far, we've presented various predefined `IntStream` reductions
- You can define your own reductions via an `IntStream`'s `reduce` method—in fact, each terminal operation discussed in Section 17.7.2 is a specialized implementation of `reduce`
- The stream pipeline in lines 30–31
  - `IntStream.of(values)`
  - `.reduce(0, (x, y) -> x + y)`
- shows how to total an `IntStream`'s values using `reduce`, rather than `sum`

17.7.3 Terminal Operation `reduce` (cont.)

- The first argument to `reduce` (`0`) is the operation's identity value—a value that, when combined with any stream element (using the lambda in the `reduce`'s second argument), produces the element's original value
- For example, when summing the elements, the identity value is 0, because any `int` value added to 0 results in the original `int` value
- Similarly, when getting the product of the elements the identity value is 1, because any `int` value multiplied by 1 results in the original `int` value.
- Method `reduce`'s second argument is a method that receives two `int` values (representing the left and right operands of a binary operator), performs a calculation with the values and returns the result
- A lambda with two or more parameters *must* enclose them in parentheses

Error-Prevention Tip 17.2

The operation specified by a `reduce`'s argument must be associative—that is, the order in which `reduce` applies the operation to the stream's elements must not matter. This is important, because `reduce` is allowed to apply its operation to the stream elements in any order. A non-associative operation could yield different results based on the processing order. For example, subtraction is not an associative operation—the expression 7 – (5 – 3) yields 5 whereas the expression (7 – 5) – 3 yields –1. Associative `reduce` operations are critical for parallel streams (Chapter 23) that split operations across multiple cores for better performance. Exercise 23.19 explores this issue further.

17.7.3 Terminal Operation `reduce` (cont.)

- Based on the stream's elements
  - `3 10 6 1 4 8 2 5 9 7`
- the reduction's evaluation proceeds as follows:
  - `0 + 3 --> 3`
  - `3 + 10 --> 13`
  - `13 + 6 --> 19`
  - `19 + 1 --> 20`
  - `20 + 4 --> 24`
  - `24 + 8 --> 32`
  - `32 + 2 --> 34`
  - `34 + 5 --> 39`
  - `39 + 9 --> 48`
  - `48 + 7 --> 55`
- Notice that the first calculation uses the identity value (0) as the left operand and each subsequent calculation uses the result of the prior calculation as the left operand
- The reduction process continues producing a running total of the `IntStream`'s values until they've all been used, at which point the final sum is returned.

17.7.3 Terminal Operation `reduce` (cont.)

Calculating the Product of the Values with Method `reduce`

- The stream pipeline in lines 35–36
  - `IntStream.of(values)`
  - `.reduce((x, y) -> x * y).getAsInt()`
- uses the one-argument version of method `reduce`, which returns an `OptionalInt` that, if the stream has elements, contains the product of the `IntStream`'s values; otherwise, the `OptionalInt` does not contain a result

17.7.3 Terminal Operation `reduce` (cont.)

- Based on the stream's elements
  - `3 10 6 1 4 8 2 5 9 7`
- the reduction's evaluation proceeds as follows:
  - `3 * 10 --> 30`
  - `30 * 6 --> 180`
  - `180 * 1 --> 180`
  - `180 * 4 --> 720`
  - `720 * 8 --> 5,760`
  - `5,760 * 2 --> 11,520`
  - `11,520 * 5 --> 57,600`
  - `57,600 * 9 --> 518,400`
  - `518,400 * 7 --> 3,628,800`

17.7.3 Terminal Operation `reduce` (cont.)

- This process continues producing a running product of the `IntStream`'s values until they've all been used, at which point the final product is returned.
- We could have used the two-parameter `reduce` method, as in:
  - `IntStream.of(values)`
  - `        .reduce(1, (x, y) -> x * y)`
- However, if the stream were empty, this version of `reduce` would return the identity value (`1`), which would not be the expected result for an empty stream

17.7.3 Terminal Operation `reduce` (cont.)

Summing the Squares of the Values

- Now consider summing the squares of the stream's elements
- When implementing your stream pipelines, it's helpful to break down the processing steps into easy-to-understand tasks
- Summing the squares of the stream's elements requires two distinct tasks:
  - squaring the value of each stream element
  - summing the resulting values.

17.7.3 Terminal Operation `reduce` (cont.)

- Rather than defining this with a `reduce` method call, the stream pipeline in lines 40–42
  - `IntStream.of(values)`
  - `.map(x -> x * x)`
  - `.sum());`
- uses the `map` and `sum` methods to compose the sum-of-squares operation
- First `map` produces a new `IntStream` containing the original element's squares, then `sum` totals the resulting stream's elements.

17.7.4 Sorting `IntStream` Values

- In Section 7.15, you learned how to sort arrays with the `sort` `static` method of class `Arrays`
- You also may sort the elements of a stream
- The stream pipeline in lines 46–49
  - `IntStream.of(values)`
  - `.sorted()`
  - `.mapToObj(String::valueOf)`
  - `.collect(Collectors.joining(" ")));`
- sorts the stream's elements and displays each value followed by a space

17.7.4 Terminal Operation `reduce` (cont.)

- `IntStream` intermediate operation `sorted` orders the elements of the stream into *ascending* order by default
- Like `filter`, `sorted` is a *lazy* operation that's performed only when a terminal operation initiates the stream pipeline's processing

17.8 Functional Interfaces

- ***[This section requires the interface concepts introduced in Sections 10.9–10.10.]***
- Section 10.10 introduced Java SE 8's enhanced interface features—`default` methods and `static` methods—and discussed the concept of a *functional interface*—an interface that contains exactly one `abstract` method (and may also contain `default` and `static` methods)
- Such interfaces are also known as *single abstract method* (SAM) interfaces
- Functional interfaces are used extensively in functional-style Java programming

17.8 Functional Interfaces (cont.)

- Functional programmers work with so-called *pure functions* that have *referential transparency*—that is, they:
  - depend only on their parameters
  - have no side-effects and
  - do not maintain any state.
- Pure functions are methods that implement functional interfaces—typically defined as lambdas, like those you've seen so far in this chapter's examples
- State changes occur by passing data from method to method
- No data is shared

Software Engineering Observation 17.5

Pure functions are safer because they do not modify a program's state (variables). This also makes them less error prone and thus easier to test, modify and debug.

17.8 Functional Interfaces (cont.)

Functional Interfaces in Package `java.util.function`

- Package `java.util.function` contains several functional interfaces
- Figure 17.10 shows the six basic generic functional interfaces, several of which you've already used in this chapter's examples
- Throughout the table, `T` and `R` are generic type names that represent the type of the object on which the functional interface operates and the return type of a method, respectively

Interface	Description
`BinaryOperator<T>`	Represents a method that takes two parameters of the same type and returns a value of that type. Performs a task using the parameters (such as a calculation) and returns the result. The lambdas you passed to `IntStream` method `reduce` (Section 17.7) implemented `IntBinaryOperator`—an `int` specific version of `BinaryOperator`.
`Consumer<T>`	Represents a one-parameter method that returns `void`. Performs a task using its parameter, such as outputting the object, invoking a method of the object, etc. The lambda you passed to `IntStream` method `forEach` (Section 17.6) implemented interface `IntConsumer`—an `int`-specialized version of `Consumer`. Later sections present several more examples of `Consumers`.

Fig. 17.10 | The six basic generic functional interfaces in package `java.util.function`.

Interface	Description
`Function<T,R>`	Represents a one-parameter method that performs a task on the parameter and returns a result—possibly of a different type than the parameter. The lambda you passed to `IntStream` method `mapToObj` (Section 17.6) implemented interface `IntFunction`—an `int`-specialized version of `Function`. Later sections present several more examples of `Functions`.
`Predicate<T>`	Represents a one-parameter method that returns a `boolean` result. Determines whether the parameter satisfies a condition. The lambda you passed to `IntStream` method `filter` (Section 17.4) implemented interface `IntPredicate`—an `int`-specialized version of `Predicate`. Later sections present several more examples of `Predicates`.

Fig. 17.10 | The six basic generic functional interfaces in package `java.util.function`.

Interface	Description
`Supplier<T>`	Represents a no-parameter method that returns a result. Often used to create a collection object in which a stream operation's results are placed. You'll see several examples of `Suppliers` starting in Section 17.13.
`UnaryOperator<T>`	Represents a one-parameter method that returns a result of the same type as its parameter. The lambdas you passed in Section 17.3 to `IntStream` method `map` implemented `IntUnaryOperator`—an `int`-specialized version of `UnaryOperator`. Later sections present several more examples of `UnaryOperators`.

Fig. 17.10 | The six basic generic functional interfaces in package `java.util.function`.

17.8 Functional Interfaces (cont.)

- Many other functional interfaces in package `java.util.function` are specialized versions of those in Fig. 17.10
- Most are for use with `int`, `long` and `double` primitive values
- There are also generic customizations of `Consumer`, `Function` and `Predicate` for binary operations—that is, methods that take two arguments
- For each `IntStream` method we've shown that receives a lambda, the method's parameter is actually an `int`-specialized version of one of these interfaces

17.9 Lambdas: A Deeper Look

Type Inference and a Lambda's Target Type

- Lambda expressions can be used anywhere functional interfaces are expected
- The Java compiler can usually *infer* the types of a lambda's parameters and the type returned by a lambda from the context in which the lambda is used
- This is determined by the lambda's target type—the functional-interface type that's expected where the lambda appears in the code
- In the call to `IntStream` method `map` from Fig. 17.4 the target type is `IntUnaryOperator`, which represents a method that takes one `int` parameter and returns an `int` result

```
IntStream.rangeClosed(1, 10)
         .map((int x) -> {return x * 2;})
         .sum()
```

- In this case, the lambda parameter's type is explicitly declared to be `int` and the compiler *infers* the lambda's return type as int, because that's what an `IntUnaryOperator` requires.

17.9 Lambdas: A Deeper Look (cont.)

- The compiler also can *infer* a lambda parameter's type
- For example, in the call to `IntStream` method `filter` from stream pipeline in Fig. 17.7 the target type is `IntPredicate`, which represents a method that takes one `int` parameter and returns a `boolean` result
  - ```
    IntStream.rangeClosed(1, 10)
             .filter(x -> x % 2 == 0)
             .map(x -> x * 3)
             .sum()
    ```
- In this case, the compiler *infers* the lambda parameter `x`'s type as `int`, because that's what an `IntPredicate` requires
- We generally let the compiler *infer* the lambda parameter's type in our examples
    ```

17.9 Lambdas: A Deeper Look (cont.)

Scope and Lambdas

- Unlike methods, lambdas do not have their own scope
- So, for example, you cannot shadow an enclosing method's local variables with lambda parameters that have the same names
- A compilaton error occurs in this case, because the method's local variables and the lambda parameters are in the *same* scope

17.9 Lambdas: A Deeper Look (cont.)

Capturing Lambdas and `final` Local Variables

- A lambda that refers to a local variable from the enclosing method (known as the lambda's *lexical scope*) is a capturing lambda
- For such a lambda, the compiler captures the local variable's value and stores it with the lambda to ensure that the lambda can use the value when the lambda *eventually* executes
- This is important, because you can pass a lambda to another method that executes the lambda *after* its lexical scope *no longer exists*

17.9 Lambdas: A Deeper Look (cont.)

- Any local variable that a lambda references in its lexical scope must be `final`
- Such a variable either can be explicitly declared `final` or it can be *effectively* `final` (Java SE 8)
- For an effectively `final` variable, the compiler *infers* that the local variable could have been declared `final`, because its enclosing method never modifies the variable after it's declared and initialized

17.10 `Stream<Integer>` Manipulations

- ***[This section requires the interface concepts introduced in Sections 10.9–10.10.]***
- A `Stream` performs tasks on reference-type objects
- `IntStream` is simply an `int`-optimized `Stream` that provides methods for common `int` operations
- Figure 17.11 performs *filtering* and *sorting* on a `Stream<Integer>`, using techniques similar to those in prior examples, and shows how to place a stream pipeline's results into a new collection for subsequent processing
- We'll work with `Stream`s of other reference types in subsequent examples

```
1  // Fig. 17.11: ArraysAndStreams.java
2  // Demonstrating lambdas and streams with an array of Integers.
3  import java.util.Arrays;
4  import java.util.List;
5  import java.util.stream.Collectors;
6
7  public class ArraysAndStreams {
8     public static void main(String[] args) {
9        Integer[] values = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};
10
11       // display original values
12       System.out.printf("Original values: %s%n", Arrays.asList(values));
13
```

Fig. 17.11 | Demonstrating lambdas and streams with an array of `Integers`. (Part 1 of 4.)

```
14          // sort values in ascending order with streams
15          System.out.printf("Sorted values: %s%n",
16             Arrays.stream(values)
17                   .sorted()
18                   .collect(Collectors.toList()));
19
20          // values greater than 4
21          List<Integer> greaterThan4 =
22             Arrays.stream(values)
23                   .filter(value -> value > 4)
24                   .collect(Collectors.toList());
25          System.out.printf("Values greater than 4: %s%n", greaterThan4);
26
```

Fig. 17.11 | Demonstrating lambdas and streams with an array of `Integers`. (Part 2 of 4.)

```
27          // filter values greater than 4 then sort the results
28          System.out.printf("Sorted values greater than 4: %s%n",
29             Arrays.stream(values)
30                   .filter(value -> value > 4)
31                   .sorted()
32                   .collect(Collectors.toList()));
33
34          // greaterThan4 List sorted with streams
35          System.out.printf(
36             "Values greater than 4 (ascending with streams): %s%n",
37             greaterThan4.stream()
38                   .sorted()
39                   .collect(Collectors.toList()));
40       }
41    }
```

Fig. 17.11 | Demonstrating lambdas and streams with an array of `Integers`. (Part 3 of 4.)

```
Original values: [2, 9, 5, 0, 3, 7, 1, 4, 8, 6]
Sorted values: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Values greater than 4: [9, 5, 7, 8, 6]
Sorted values greater than 4: [5, 6, 7, 8, 9]
Values greater than 4 (ascending with streams): [5, 6, 7, 8, 9]
```

Fig. 17.11 | Demonstrating lambdas and streams with an array of `Integers`. (Part 4 of 4.)

17.10 `Stream<Integer>` Manipulations (cont.)

- We use the `Integer` array `values` (line 9) that's initialized with `int` values—the compiler *boxes* each `int` into an `Integer` object
- Line 12 displays the contents of `values` before we perform any stream processing
- `Arrays` method `asList` creates a `List<Integer>` view of the `values` array
- The generic interface `List` (discussed in more detail in Chapter 16) is implemented by collections like `ArrayList` (Chapter 7)
- Line 12 displays the `List<Integer>`'s default `String` representation, which consists of square brackets (`[` and `]`) containing a comma-separated list of elements—we use this `String` representation throughout the example
- We walk through the remainder of the code in Sections 17.10.1–17.10.5

17.10.1 Creating a `Stream<Integer>`

- Class `Arrays` `stream` method can be used to create a `Stream` from an array of objects—for example, line 16 produces a `Stream<Integer>`, because `stream`'s argument is an array of `Integer`s
- Interface `Stream` (package `java.util.stream`) is a generic interface for performing stream operations on any *reference* type
- The types of objects that are processed are determined by the `Stream`'s source.
- Class `Arrays` also provides overloaded versions of method `stream` for creating `IntStream`s, `LongStream`s and `DoubleStream`s from `int`, `long` and `double` arrays or from ranges of elements in the arrays

17.10.2 Sorting a `Stream` and Collecting the Results

- The stream pipeline in lines 16–18
  - `Arrays.stream(values)`
  - `        .sorted()`
  - `        .collect(Collectors.toList())`
- uses stream techniques to sort the `values` array and collect the results in a `List<Integer>`
- First, line 16 creates a `Stream<Integer>` from `values`
- Next, line 17 calls `Stream` method `sorted` to sort the elements—this results in an intermediate `Stream<Integer>` with the values in *ascending* order

17.10.2 Sorting a `Stream` and Collecting the Results (cont.)

Creating a New Collection Containing a Stream Pipeline's Results

- When processing streams, you often create *new* collections containing the results so that you can perform operations on them later
- To do so, you can use `Stream`'s terminal operation `collect` (Fig. 17.11, line 18)
- As the stream pipeline is processed, method `collect` performs a mutable reduction operation that creates a `List`, `Map` or `Set` and modifies it by placing the stream pipeline's results into the collection
- You may also use the mutable reduction operation `toArray` to place the results in a new array of the `Stream`'s element type.

17.10.2 Sorting a `Stream` and Collecting the Results (cont.)

Creating a New Collection Containing a Stream Pipeline's Results

- The version of method `collect` in line 18 receives as its argument an object that implements interface `Collector` (package `java.util.stream`), which specifies how to perform the mutable reduction
- Class `Collectors` (package `java.util.stream`) provides `static` methods that return predefined `Collector` implementations
- `Collectors` method `toList` (line 18) returns a `Collector` that places the `Stream<Integer>`'s elements into a `List<Integer>` collection
- In lines 15–18, the resulting `List<Integer>` is displayed with an *implicit* call to its `toString` method

17.10.2 Sorting a `Stream` and Collecting the Results (cont.)

Creating a New Collection Containing a Stream Pipeline's Results

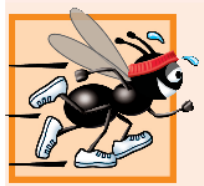
- A mutable reduction optionally performs a final data transformation
- For example, in Fig. 17.8, we called `IntStream` method `collect` with the object returned by `Collectors` method `joining`
- This `Collector` uses a `StringJoiner` (package `java.util`) to concatenate the stream elements' `String` representations, then called the `StringJoiner`'s `toString` method to transform the result into a `String`
- For more predefined `Collectors`, visit:
  - https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

17.10.3 Filtering a `Stream` and Storing the Results for Later Use

- The pipeline in lines 21–24 of Fig. 17.11 creates a `Stream<Integer>`, `filter`s the stream to locate all the values greater than 4 and `collect`s the results into a `List<Integer->`
  - `List<Integer> greaterThan4 =`
  - `   Arrays.stream(values)`
  - `         .filter(value -> value > 4)`
  - `         .collect(Collectors.toList());`
- `Stream` method `filter`'s argument implements the functional interface `Predicate` (package `java.util.function`), which represents a one-parameter method that returns a `boolean` indicating whether the parameter value satisfies the predicate
- We assign the stream pipeline's resulting `List<Integer>` to variable `greaterThan4`, which is used in line 25 to display the values greater than 4 and used again in lines 37–39 to perform additional operations on only the values greater than 4.

17.10.4 Filtering and Sorting a `Stream` and Collecting the Results

- The stream pipeline in lines 29–32 displays the values greater than 4 in sorted order
  - `Arrays.stream(values)`
  - `.filter(value -> value > 4)`
  - `.sorted()`
  - `.collect(Collectors.toList())`
- First, line 29 creates a `Stream<Integer>`
- Then line 30 `filter`s the elements to locate all the values greater than 4
- Next, line 31 indicates that we'd like the results `sorted`
- Finally, line 32 `collect`s the results into a `List<Integer>`, which is then displayed as a `String`



Performance Tip 17.2

Call `filter` before `sorted` so that the stream pipeline sorts only the elements that will be in the stream pipeline's result.

17.10.5 Sorting Previously Collected Results

- The stream pipeline in lines 37–39 uses the `greaterThan4` collection created in lines 21–24 to show additional processing on the results of a prior stream pipeline
  - `greaterThan4.stream()`
  - `.sorted()`
  - `.collect(Collectors.toList()));`
- `List` method `stream` creates the stream
- Then we sort the elements and `collect` the results into a new `List<Integer>` and display its `String` representation

17.11 `Stream<String>` Manipulations

- ***[This section demonstrates how lambdas and streams can be used to simplify programming tasks that you learned in Chapter 14.]***
- So far, we've manipulated only streams of `int` values and `Integer` objects
- Figure 17.12 performs similar stream operations on a `Stream<String>`
- In addition, we demonstrate *case-insensitive sorting* and sorting in *descending* order
- Throughout this example, we use the `String` array `strings` (lines 9–10) that's initialized with color names—some with an initial uppercase letter
- Line 13 displays the contents of `strings` *before* we perform any stream processing
- We walk through the rest of the code in Sections 17.11.1–17.11.3

```
1  // Fig. 17.12: ArraysAndStreams2.java
2  // Demonstrating lambdas and streams with an array of Strings.
3  import java.util.Arrays;
4  import java.util.Comparator;
5  import java.util.stream.Collectors;
6
7  public class ArraysAndStreams2 {
8     public static void main(String[] args) {
9        String[] strings =
10          {"Red", "orange", "Yellow", "green", "Blue", "indigo", "Violet"};
11
12       // display original strings
13       System.out.printf("Original strings: %s%n", Arrays.asList(strings));
14
```

Fig. 17.12 | Demonstrating lambdas and streams with an array of `Strings`. (Part 1 of 3.)

```
15          // strings in uppercase
16          System.out.printf("strings in uppercase: %s%n",
17             Arrays.stream(strings)
18                   .map(String::toUpperCase)
19                   .collect(Collectors.toList()));
20
21          // strings less than "n" (case insensitive) sorted ascending
22          System.out.printf("strings less than n sorted ascending: %s%n",
23             Arrays.stream(strings)
24                   .filter(s -> s.compareToIgnoreCase("n") < 0)
25                   .sorted(String.CASE_INSENSITIVE_ORDER)
26                   .collect(Collectors.toList()));
27
```

Fig. 17.12 | Demonstrating lambdas and streams with an array of `Strings`. (Part 2 of 3.)

```
28          // strings less than "n" (case insensitive) sorted descending
29          System.out.printf("strings less than n sorted descending: %s%n",
30             Arrays.stream(strings)
31                   .filter(s -> s.compareToIgnoreCase("n") < 0)
32                   .sorted(String.CASE_INSENSITIVE_ORDER.reversed())
33                   .collect(Collectors.toList()));
34       }
35    }
```

```
Original strings: [Red, orange, Yellow, green, Blue, indigo, Violet]
strings in uppercase: [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET]
strings less than n sorted ascending: [Blue, green, indigo]
strings less than n sorted descending: [indigo, green, Blue]
```

Fig. 17.12 | Demonstrating lambdas and streams with an array of `Strings`. (Part 3 of 3.)

17.11.1 Mapping `String`s to Uppercase

- The stream pipeline in lines 17–19
  - `Arrays.stream(strings)`
  - `        .map(String::toUpperCase)`
  - `        .collect(Collectors.toList()));`
- displays the `String`s in uppercase letters
- To do so, line 17 creates a `Stream<String>` from the array `strings`, then line 18 `map`s each `String` to its uppercase version by calling `String` instance method `toUpperCase` on each stream element

17.11.1 Mapping `Strings` to Uppercase (cont.)

- `Stream` method `map` receives an object that implements the `Function` functional interface, representing a one-parameter method that performs a task with its parameter then returns the result
- In this case, we pass to `map` an unbound instance method reference of the form *ClassName*`::`*instanceMethodName* (`String::toUpperCase`)
- "Unbound" means that the method reference does not indicate the specific object on which the method will be called—the compiler converts this to a one-parameter lambda that invokes the instance method on the lambda's parameter, which must have type *ClassName*
- In this case, the compiler converts `String::toUpperCase` into a lambda like
  - `s -> s.toUpperCase()`
- which returns the uppercase version of the lambda's argument
- Line 19 `collects` the results into a `List<String>` that we output as a `String`

17.11.2 Filtering `Strings` Then Sorting Them in Case-Insensitive Ascending Order

- The stream pipeline in lines 23–26
  - `Arrays.stream(strings)`
  - `.filter(s -> s.compareToIgnoreCase("n") < 0)`
  - `.sorted(String.CASE_INSENSITIVE_ORDER)`
  - `.collect(Collectors.toList())`
- filters and sort the `String`s
- Line 23 creates a `Stream<String>` from the array `strings`, then line 24 calls `Stream` method `filter` to locate all the `String`s that are less than `"n"`, using a *case-insensitive* comparison in the `Predicate` lambda

17.11.2 Filtering `String`s Then Sorting Them in Case-Insensitive Ascending Order (cont.)

- Line 25 sorts the results and line 26 collects them into a `List<String>` that we output as a `String`
- In this case, line 25 invokes the version of `Stream` method `sorted` that receives a `Comparator` as an argument
- A `Comparator` defines a `compare` method that returns a negative value if the first value being compared is less than the second, 0 if they're equal and a positive value if the first value is greater than the second
- By default, method `sorted` uses the *natural order* for the type—for `String`s, the natural order is case sensitive, which means that `"Z"` is less than `"a"`
- Passing the predefined `Comparator` `String.CASE_INSENSITIVE_ORDER` performs a *case-insensitive* sort

17.11.3 Filtering `String`s Then Sorting Them in Case-Insensitive Descending Order

- The stream pipeline in lines 30–33
  - `Arrays.stream(strings)`
  - `.filter(s -> s.compareToIgnoreCase("n") < 0)`
  - `.sorted(String.CASE_INSENSITIVE_ORDER.reversed())`
  - `.collect(Collectors.toList()));`
- performs the same tasks as lines 23–26, but sorts the `String`s in *descending* order
- Functional interface `Comparator` contains `default` method `reversed`, which reverses an existing `Comparator`'s ordering
- When you apply `reversed` to `String.CASE_INSENSITIVE_ORDER`, `sorted` performs a case-insensitive sort and places the `String`s in *descending* order

17.12 `Stream<Employee>` Manipulations

- ***[This section demonstrates how lambdas and streams can be used to simplify programming tasks that you learned in Chapter 16, Generic Collections.]***
- The previous examples in this chapter performed stream manipulations on primitive types (like `int`) and Java class library types (like `Integer` and `String`)
- Of course, you also may perform operations on collections of programmer-defined types
- The example in Figs. 17.13–17.21 demonstrates various lambda and stream capabilities using a `Stream<Employee>`
- Class `Employee` (Fig. 17.13) represents an employee with a first name, last name, salary and department and provides methods for getting these values
- In addition, the class provides a `getName` method (lines 39–41) that returns the combined first and last name as a `String`, and a `toString` method (lines 44–48) that returns a formatted `String` containing the employee's first name, last name, salary and department

```
1  // Fig. 17.13: Employee.java
2  // Employee class.
3  public class Employee {
4     private String firstName;
5     private String lastName;
6     private double salary;
7     private String department;
8
9     // constructor
10    public Employee(String firstName, String lastName,
11       double salary, String department) {
12       this.firstName = firstName;
13       this.lastName = lastName;
14       this.salary = salary;
15       this.department = department;
16    }
```

Fig. 17.13 | Employee class for use in Figs. 17.14–17.21. (Part 1 of 3.)

```
17
18      // get firstName
19      public String getFirstName() {
20         return firstName;
21      }
22
23      // get lastName
24      public String getLastName() {
25         return lastName;
26      }
27
28      // get salary
29      public double getSalary() {
30         return salary;
31      }
32
```

Fig. 17.13 | Employee class for use in Figs. 17.14–17.21. (Part 2 of 3.)

```
33      // get department
34      public String getDepartment() {
35         return department;
36      }
37
38      // return Employee's first and last name combined
39      public String getName() {
40         return String.format("%s %s", getFirstName(), getLastName());
41      }
42
43      // return a String containing the Employee's information
44      @Override
45      public String toString() {
46         return String.format("%-8s %-8s %8.2f   %s",
47            getFirstName(), getLastName(), getSalary(), getDepartment());
48      }
49   }
```

Fig. 17.13 | Employee class for use in Figs. 17.14–17.21. (Part 3 of 3.)

17.12.1 Creating and Displaying a `List<Employee>`

- Class `ProcessingEmployees` (Figs. 17.14–17.21) is split into several figures so we can keep the discussions of the example's lambda and streams operations close to the corresponding code
- Each figure also contains the portion of the program's output that correspond to code shown in that figure
- Figure 17.14 creates an array of `Employee`s (lines 15–22) and gets its `List` view (line 25)

17.12.1 Creating and Displaying a `List<Employee>` (cont.)

- Line 29 creates a `Stream<Employee>`, then uses `Stream` method `forEach` to display each `Employee`'s `String` representation
- `Stream` method `forEach` expects as its argument an object that implements the `Consumer` functional interface, which represents an action to perform on each element of the stream—the corresponding method receives one argument and returns `void`
- The bound instance method reference `System.out::println` is converted by the compiler into a one-parameter lambda that passes the lambda's argument—an `Employee`—to the `System.out` object's `println` method
- Figure 17.14's output shows the results of displaying each `Employee`'s `String` representation (line 29)—in this case, `Stream` method `forEach` passes each `Employee` to the `System.out` object's `println` method, which calls the `Employee`'s `toString` method

```
1  // Fig. 17.14: ProcessingEmployees.java
2  // Processing streams of Employee objects.
3  import java.util.Arrays;
4  import java.util.Comparator;
5  import java.util.List;
6  import java.util.Map;
7  import java.util.TreeMap;
8  import java.util.function.Function;
9  import java.util.function.Predicate;
10 import java.util.stream.Collectors;
11
```

Fig. 17.14 | Processing streams of `Employee` objects. (Part 1 of 3.)

```
12  public class ProcessingEmployees {
13     public static void main(String[] args) {
14        // initialize array of Employees
15        Employee[] employees = {
16           new Employee("Jason", "Red", 5000, "IT"),
17           new Employee("Ashley", "Green", 7600, "IT"),
18           new Employee("Matthew", "Indigo", 3587.5, "Sales"),
19           new Employee("James", "Indigo", 4700.77, "Marketing"),
20           new Employee("Luke", "Indigo", 6200, "IT"),
21           new Employee("Jason", "Blue", 3200, "Sales"),
22           new Employee("Wendy", "Brown", 4236.4, "Marketing")};
23
24        // get List view of the Employees
25        List<Employee> list = Arrays.asList(employees);
26
27        // display all Employees
28        System.out.println("Complete Employee list:");
29        list.stream().forEach(System.out::println);
30
```

Fig. 17.14 | Processing streams of `Employee` objects. (Part 2 of 3.)

```
Complete Employee list:
Jason    Red        5000.00   IT
Ashley   Green      7600.00   IT
Matthew  Indigo     3587.50   Sales
James    Indigo     4700.77   Marketing
Luke     Indigo     6200.00   IT
Jason    Blue       3200.00   Sales
Wendy    Brown      4236.40   Marketing
```

Fig. 17.14 | Processing streams of `Employee` objects. (Part 3 of 3.)

17.12.1 Creating and Displaying a `List<Employee>` (cont.)

Java SE 9: Creating an Immutable `List<Employee>` with `List` Method `of`

- In Fig. 17.14, we first created an array of `Employee`s (lines 15–22), then obtained a `List` view of the array (line 25)
- Recall from Chapter 16 that in Java SE 9, you can populate an immutable `List` directly via `List` `static` method `of`, as in:

```
List<Employee> list = List.of(
   new Employee("Jason", "Red", 5000, "IT"),
   new Employee("Ashley", "Green", 7600, "IT"),
   new Employee("Matthew", "Indigo", 3587.5, "Sales"),
   new Employee("James", "Indigo", 4700.77, "Marketing"),
   new Employee("Luke", "Indigo", 6200, "IT"),
   new Employee("Jason", "Blue", 3200, "Sales"),
   new Employee("Wendy", "Brown", 4236.4, "Marketing"));
```

17.12.2 Filtering `Employees` with Salaries in a Specified Range

- So far, we've used lambdas only by passing them directly as arguments to stream methods
- Figure 17.15 demonstrates storing a lambda in a variable for later use
- Lines 32–33 declare a variable of the functional interface type `Predicate<Employee>` and initialize it with a one-parameter lambda that returns a `boolean` (as required by `Predicate`)
- The lambda returns true if an `Employee`'s salary is in the range 4000 to 6000
- We use the stored lambda in lines 40 and 47 to filter `Employees`

```
31          // Predicate that returns true for salaries in the range $4000-$6000
32          Predicate<Employee> fourToSixThousand =
33             e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);
34
35          // Display Employees with salaries in the range $4000-$6000
36          // sorted into ascending order by salary
37          System.out.printf(
38             "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
39          list.stream()
40              .filter(fourToSixThousand)
41              .sorted(Comparator.comparing(Employee::getSalary))
42              .forEach(System.out::println);
43
44          // Display first Employee with salary in the range $4000-$6000
45          System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
46             list.stream()
47                 .filter(fourToSixThousand)
48                 .findFirst()
49                 .get());
50
```

Fig. 17.15 | Filtering `Employees` with salaries in the range $4000–$6000. (Part 1 of 2.)

```
Employees earning $4000-$6000 per month sorted by salary:
Wendy    Brown      4236.40   Marketing
James    Indigo     4700.77   Marketing
Jason    Red        5000.00   IT

First employee who earns $4000-$6000:
Jason    Red        5000.00   IT
```

Fig. 17.15 | Filtering `Employees` with salaries in the range $4000–$6000. (Part 2 of 2.)

17.12.2 Filtering `Employees` with Salaries in a Specified Range (cont.)

- The stream pipeline in lines 39–42 performs the following tasks:
  - Line 39 creates a `Stream<Employee>`.
  - Line 40 filters the stream using the `Predicate` named `fourToSixThousand`.
  - Line 41 sorts *by salary* the `Employees` that remain in the stream. To create a salary `Comparator`, we use the `Comparator` interface's `static` method `comparing`, which receives a `Function` that performs a task on its argument and returns the result. The unbound instance method reference `Employee::getSalary` is converted by the compiler into a one-parameter lambda that calls `getSalary` on its `Employee` argument. The `Comparator` returned by method `comparing` calls its `Function` argument on each of two `Employee` objects, then returns a negative value if the first `Employee`'s salary is less than the second, 0 if they're equal and a positive value if the first `Employee`'s salary is greater than the second. `Stream` method `sorted` uses these values to order the `Employee`s.
  - Finally, line 42 performs the terminal `forEach` operation that processes the stream pipeline and outputs the `Employee`s sorted by salary.

17.12.2 Filtering `Employees` with Salaries in a Specified Range (cont.)

Short-Circuit Stream Pipeline Processing

- In Section 5.9, you studied short-circuit evaluation with the logical AND (&&) and logical OR (||) operators
- One of the nice performance features of lazy evaluation is the ability to perform *short-circuit evaluation*—that is, to stop processing the stream pipeline as soon as the desired result is available
- Line 48 of Fig. 17.15 demonstrates `Stream` method `findFirst`—a *short-circuiting terminal operation* that processes the stream pipeline and terminates processing as soon as the *first* object from the stream's intermediate operation(s) is found

17.12.2 Filtering `Employees` with Salaries in a Specified Range (cont.)

Short-Circuit Stream Pipeline Processing

- Based on the original list of `Employee`s, the stream pipeline in lines 46–49
  - `list.stream()`
  - `    .filter(fourToSixThousand)`
  - `    .findFirst()`
  - `    .get()`
- which filters `Employee`s with salaries in the range $4000–$6000—proceeds as follows:
  - The `Predicate` `fourToSixThousand` is applied to the first `Employee` (Jason- Red). His salary ($5000.00) is in the range $4000–$6000, so the `Predicate` returns `true` and processing of the stream terminates *immediately*, having processed only one of the eight objects in the stream.
  - Method `findFirst` then returns an `Optional` (in this case, an `Optional<Employee>`) containing the object that was found, if any. The call to `Optional` method `get` (line 49) returns the matching `Employee` object in this example. Even if the stream contained millions of `Employee` objects, the `filter` operation would be performed only until a match was found.

17.12.2 Filtering `Employees` with Salaries in a Specified Range (cont.)

Short-Circuit Stream Pipeline Processing

- We knew from this example's `Employees` that this pipeline would find at least one `Employee` with a salary in the range 4000–6000
- So, we called `Optional` method `get` without first checking whether the `Optional` contained a result
- If `findFirst` yields an empty `Optional`, this would cause a `NoSuchElementException`
- Method `findFirst` is one of several search-related terminal operations
- Figure 17.16 shows several similar `Stream` methods

Error-Prevention Tip 17.3

For a stream operation that returns an `Optional<T>`, store the result in a variable of that type, then use the object's `isPresent` method to confirm that there is a result, before calling the `Optional`'s `get` method. This prevents `NoSuchElementExceptions`.

17.12.3 Sorting `Employees` By Multiple Fields

- Figure 17.17 shows how to use streams to sort objects by *multiple* fields
- In this example, we sort `Employee`s by last name, then, for `Employee`s with the same last name, we also sort them by first name
- To do so, we begin by creating two `Function`s that each receive an `Employee` and return a `String`:
  - `byFirstName` (line 52) is assigned a method reference for `Employee` instance method `getFirstName`
  - `byLastName` (line 53) is assigned a method reference for `Employee` instance method `getLastName`

Search-related terminal stream operations	
`findAny`	Similar to `findFirst`, but finds and returns *any* stream element based on the prior intermediate operations. Immediately terminates processing of the stream pipeline once such an element is found. Typically, `findFirst` is used with sequential streams and `findAny` is used with parallel streams (Section 23.13).
`anyMatch`	Determines whether *any* stream elements match a specified condition. Returns `true` if at least one stream element matches and `false` otherwise. Immediately terminates processing of the stream pipeline if an element matches.
`allMatch`	Determines whether *all* of the elements in the stream match a specified condition. Returns `true` if so and `false` otherwise. Immediately terminates processing of the stream pipeline if any element does not match.

Fig. 17.16 | Search-related terminal stream operations.

17.12.3 Sorting `Employees` By Multiple Fields (cont.)

- Next, we use these `Function`s to create a `Comparator` (`lastThenFirst`; lines 56–57) that first compares two `Employee`s by last name, then compares them by first name
- We use `Comparator` method `comparing` to create a `Comparator` that calls `Function` `byLastName` on an `Employee` to get its last name
- On the resulting `Comparator`, we call `Comparator` method `thenComparing` to create a *composed* `Comparator` that first compares `Employee`s by last name and, *if the last names are equal*, then compares them by first name

17.12.3 Sorting `Employees` By Multiple Fields (cont.)

- Lines 62–64 use this new `lastThenFirst` `Comparator` to sort the `Employee`s in *ascending* order, then display the results
- We reuse the `Comparator` in lines 69–71, but call its `reversed` method to indicate that the `Employee`s should be sorted in *descending* order by last name, then first name
- Lines 52–57 may be expressed more concisely as:
  - `Comparator<Employee> lastThenFirst =`
  - `   Comparator.comparing(Employee::getLastName)`
  `.thenComparing(Employee::getFirstName);`

```
51          // Functions for getting first and last names from an Employee
52          Function<Employee, String> byFirstName = Employee::getFirstName;
53          Function<Employee, String> byLastName = Employee::getLastName;
54
55          // Comparator for comparing Employees by first name then last name
56          Comparator<Employee> lastThenFirst =
57             Comparator.comparing(byLastName).thenComparing(byFirstName);
58
59          // sort employees by last name, then first name
60          System.out.printf(
61             "%nEmployees in ascending order by last name then first:%n");
62          list.stream()
63              .sorted(lastThenFirst)
64              .forEach(System.out::println);
65
66          // sort employees in descending order by last name, then first name
67          System.out.printf(
68             "%nEmployees in descending order by last name then first:%n");
69          list.stream()
70              .sorted(lastThenFirst.reversed())
71              .forEach(System.out::println);
72
```

Fig. 17.17 | Sorting `Employees` by last name then first name. (Part 1 of 2.)

```
Employees in ascending order by last name then first:
Jason    Blue       3200.00   Sales
Wendy    Brown      4236.40   Marketing
Ashley   Green      7600.00   IT
James    Indigo     4700.77   Marketing
Luke     Indigo     6200.00   IT
Matthew  Indigo     3587.50   Sales
Jason    Red        5000.00   IT

Employees in descending order by last name then first:
Jason    Red        5000.00   IT
Matthew  Indigo     3587.50   Sales
Luke     Indigo     6200.00   IT
James    Indigo     4700.77   Marketing
Ashley   Green      7600.00   IT
Wendy    Brown      4236.40   Marketing
Jason    Blue       3200.00   Sales
```

Fig. 17.17 | Sorting `Employees` by last name then first name. (Part 2 of 2.)

17.12.3 Sorting `Employees` By Multiple Fields (cont.)

Aside: Composing Lambda Expressions

- Many functional interfaces in the package `java.util.function` package provide `default` methods that enable you to compose functionality
- For example, consider the interface `IntPredicate`, which contains three `default` methods:
  - `and`—performs a *logical AND* with *short-circuit evaluation* between the `IntPredicate` on which it's called and the `IntPredicate` it receives as an argument.
  - `negate`—*reverses* the `boolean` value of the `IntPredicate` on which it's called.
  - `or`—performs a *logical OR* with *short-circuit evaluation* between the `IntPredicate` on which it's called and the `IntPredicate` it receives as an argument.

17.12.3 Sorting Employees By Multiple Fields (cont.)

- You can use these methods and `IntPredicate` objects to compose more complex conditions
- For example, consider the following two `IntPredicate`s that are each initialized with lambdas:
  - `IntPredicate even = value -> value % 2 == 0;`
  - `IntPredicate greaterThan5 = value -> value > 5;`
- To locate all the even integers greater than 5 in an `IntStream`, you could pass to `IntStream` method `filter` the following composed `IntPredicate`:
  - `even.and(greaterThan5)`
- Like `IntPredicate`, functional interface `Predicate` represents a method that returns a `boolean` indicating whether its argument satisfies a condition
- `Predicate` also contains methods `and` and `or` for combining predicates, and `negate` for reversing a predicate's `boolean` value.

17.12.4 Mapping `Employees` to Unique-Last-Name `Strings`

- You previously used `map` operations to perform calculations on `int` values, to convert `int`s to `String`s and to convert `String`s to uppercase letters
- Figure 17.18 maps objects of one type (`Employee`) to objects of a different type (`String`)
- The stream pipeline in lines 75–79 performs the following tasks:
  - Line 75 creates a `Stream<Employee>`.
  - Line 76 maps the `Employees` to their last names using the unbound instance-method reference `Employee::getName` as method `map`'s `Function` argument. The result is a `Stream<String>` containing only the `Employee`s' last names.
  - Line 77 calls `Stream` method `distinct` on the `Stream<String>` to eliminate any duplicate `String`s—the resulting stream contains only unique last names.
  - Line 78 sorts the unique last names.
  - Finally, line 79 performs a terminal `forEach` operation that processes the stream pipeline and outputs the unique last names in sorted order.
- Lines 84–87 sort the `Employees` by last name then, first name, then `map` the `Employee`s to `String`s with `Employee` instance method `getName` (line 86) and display the sorted names in a terminal `forEach` operation

```
73          // display unique employee last names sorted
74          System.out.printf("%nUnique employee last names:%n");
75          list.stream()
76              .map(Employee::getLastName)
77              .distinct()
78              .sorted()
79              .forEach(System.out::println);
80
81          // display only first and last names
82          System.out.printf(
83             "%nEmployee names in order by last name then first name:%n");
84          list.stream()
85              .sorted(lastThenFirst)
86              .map(Employee::getName)
87              .forEach(System.out::println);
88
```

Fig. 17.18 | Mapping `Employee` objects to last names and whole names. (Part 1 of 2.)

```
Unique employee last names:
Blue
Brown
Green
Indigo
Red

Employee names in order by last name then first name:
Jason Blue
Wendy Brown
Ashley Green
James Indigo
Luke Indigo
Matthew Indigo
Jason Red
```

Fig. 17.18 | Mapping `Employee` objects to last names and whole names. (Part 2 of 2.)

17.12.5 Grouping `Employees` By Department

- Previously, we've used the terminal stream operation `collect` to concatenate stream elements into a `String` representation and to place stream elements into `List` collections
- Figure 17.19 uses `Stream` method `collect` (line 93) to group `Employees` by department

```
89          // group Employees by department
90          System.out.printf("%nEmployees by department:%n");
91          Map<String, List<Employee>> groupedByDepartment =
92             list.stream()
93                .collect(Collectors.groupingBy(Employee::getDepartment));
94          groupedByDepartment.forEach(
95             (department, employeesInDepartment) -> {
96                System.out.printf("%n%s%n", department);
97                employeesInDepartment.forEach(
98                   employee -> System.out.printf("   %s%n", employee));
99             }
100         );
101
```

Fig. 17.19 | Grouping `Employees` by department. (Part 1 of 2.)

```
Employees by department:

Sales
   Matthew  Indigo     3587.50   Sales
   Jason    Blue       3200.00   Sales

IT
   Jason    Red        5000.00   IT
   Ashley   Green      7600.00   IT
   Luke     Indigo     6200.00   IT

Marketing
   James    Indigo     4700.77   Marketing
   Wendy    Brown      4236.40   Marketing
```

Fig. 17.19 | Grouping `Employees` by department. (Part 2 of 2.)

17.12.5 Grouping Employees By Department (cont.)

- Recall that `collect`'s argument is a `Collector` that specifies how to summarize the data into a useful form
- In this case, we use the `Collector` returned by `Collectors` `static` method `groupingBy`, which receives a `Function` that classifies the objects in the stream
- The values returned by this `Function` are used as the keys in a `Map` collection
- The corresponding values, by default, are `List`s containing the stream elements in a given category
- When method `collect` is used with this `Collector`, the result is a `Map<String, List<Employee>>` in which each `String` key is a department and each `List<Employee>` contains the `Employee`s in that department

17.12.5 Grouping `Employees` By Department (cont.)

- We assign this `Map` to variable `groupedByDepartment`, which we use in lines 94–100 to display the `Employees` grouped by department
- `Map` method `forEach` performs an operation on each of the `Map`'s key–value pairs—in this case, the keys are departments and the values are collections of the `Employee`s in a given department
- The argument to this method is an object that implements functional interface `BiConsumer`, which represents a two-parameter method that does not return a result
- For a `Map`, the first parameter represents the key and the second represents the corresponding value

17.12.6 Counting the Number of `Employees` in Each Department

- Figure 17.20 once again demonstrates `Stream` method `collect` and `Collectors` `static` method `groupingBy`, but in this case we count the number of `Employee`s in each department
- The technique shown here enables us to combine grouping and reduction into a single operation

```
102         // count number of Employees in each department
103         System.out.printf("%nCount of Employees by department:%n");
104         Map<String, Long> employeeCountByDepartment =
105            list.stream()
106               .collect(Collectors.groupingBy(Employee::getDepartment,
107                  Collectors.counting()));
108         employeeCountByDepartment.forEach(
109            (department, count) -> System.out.printf(
110               "%s has %d employee(s)%n", department, count));
111
```

```
Count of Employees by department:
Sales has 2 employee(s)
IT has 3 employee(s)
Marketing has 2 employee(s)
```

Fig. 17.20 | Counting the number of `Employees` in each department.

17.12.6 Counting the Number of `Employee`s in Each Department (cont.)

- The stream pipeline in lines 104–107 produces a `Map<String, Long>` in which each `String` key is a department name and the corresponding `Long` value is the number of `Employee`s in that department
- In this case, we use a version of `Collectors` `static` method `groupingBy` that receives two arguments:
  - the first is a `Function` that classifies the objects in the stream and
  - the second is another `Collector` (known as the downstream `Collector`) that's used to collect the objects classified by the `Function`
- We use a call to `Collectors` `static` method `counting` as the second argument
- This resulting `Collector` reduces the elements in a given classification to a count of those elements, rather than collecting them into a `List`
- Lines 108–110 then output the key–value pairs from the resulting `Map<String, Long>`

17.12.7 Summing and Averaging `Employee` Salaries

- Previously, we showed that streams of primitive-type elements can be mapped to streams of objects with method `mapToObj` (found in classes `IntStream`, `LongStream` and `DoubleStream`)
- Similarly, a `Stream` of objects may be mapped to an `IntStream`, `LongStream` or `DoubleStream`
- Figure 17.21 demonstrates `Stream` method `mapToDouble` (lines 116, 123 and 129), which maps objects to `double` values and returns a `DoubleStream`
- In this case, we map `Employee` objects to their salaries so that we can calculate the *sum* and *average*

17.12.7 Counting the Number of `Employees` in Each Department (cont.)

- Method `mapToDouble` receives an object that implements the functional interface `ToDoubleFunction` (package `java.util.function`), which represents a one-parameter method that returns a `double` value
- Lines 116, 123 and 129 each pass to `mapToDouble`– the unbound instance-method reference `Employee::getSalary`, which returns the current `Employee`'s salary as a `double`
- The compiler converts this method reference into a one-parameter lambda that calls `getSalary` on its `Employee` argument

17.12.7 Counting the Number of `Employee`s in Each Department (cont.)

- Lines 115–117 create a `Stream<Employee>`, map it to a `DoubleStream`, then invoke `DoubleStream` method `sum` to total the `Employee`s' salaries
- Lines 122–124 also sum the `Employee`s' salaries, but do so using `DoubleStream` method `reduce` rather than `sum`—note that the lambda in line 124 could be replaced with the `static` method reference
  - `Double::sum`
- Class `Double`'s `sum` method receives two `double`s and returns their sum

17.12.7 Counting the Number of `Employees` in Each Department (cont.)

- Finally, lines 128–131 calculate the average of the `Employee`s' salaries using `DoubleStream` method `average`, which returns an `OptionalDouble` in case the `DoubleStream` does not contain any elements
- Here, we know the stream has elements, so we simply call `OptionalDouble` method `getAsDouble` to get the result

```
112         // sum of Employee salaries with DoubleStream sum method
113         System.out.printf(
114            "%nSum of Employees' salaries (via sum method): %.2f%n",
115            list.stream()
116                .mapToDouble(Employee::getSalary)
117                .sum());
118
119         // calculate sum of Employee salaries with Stream reduce method
120         System.out.printf(
121            "Sum of Employees' salaries (via reduce method): %.2f%n",
122            list.stream()
123                .mapToDouble(Employee::getSalary)
124                .reduce(0, (value1, value2) -> value1 + value2));
125
```

Fig. 17.21 | Summing and averaging `Employee` salaries. (Part 1 of 2.)

```
126         // average of Employee salaries with DoubleStream average method
127         System.out.printf("Average of Employees' salaries: %.2f%n",
128            list.stream()
129                .mapToDouble(Employee::getSalary)
130                .average()
131                .getAsDouble());
132     }
133  }
```

```
Sum of Employees' salaries (via sum method): 34524.67
Sum of Employees' salaries (via reduce method): 34525.67
Average of Employees' salaries: 4932.10
```

Fig. 17.21 | Summing and averaging `Employee` salaries. (Part 2 of 2.)

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

17.13 Creating a `Stream<String>` from a File (cont.)

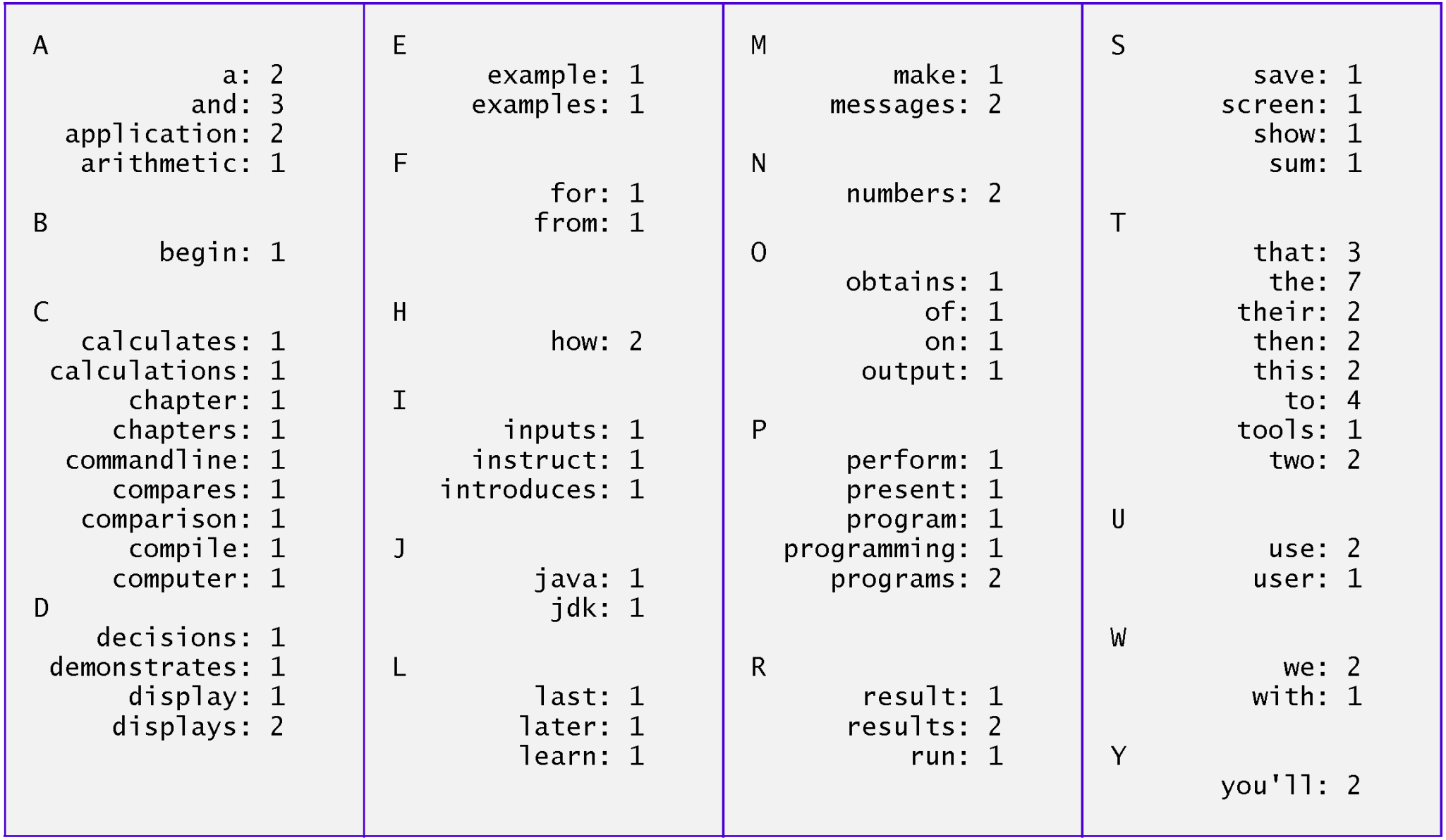
- Line 14 of Fig. 17.22 creates a regular expression `Pattern` that we'll use to split lines of text into their individual words
- The `Pattern` `\s+` represents one or more consecutive white-space characters—recall that because `\` indicates an escape sequence in a `String`, we must specify each `\` 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

```
1  // Fig. 17.22: StreamOfLines.java
2  // Counting word occurrences in a text file.
3  import java.io.IOException;
4  import java.nio.file.Files;
5  import java.nio.file.Paths;
6  import java.util.Map;
7  import java.util.TreeMap;
8  import java.util.regex.Pattern;
9  import java.util.stream.Collectors;
10
11 public class StreamOfLines {
12    public static void main(String[] args) throws IOException {
13       // Regex that matches one or more consecutive whitespace characters
14       Pattern pattern = Pattern.compile("\\s+");
15
16       // count occurrences of each word in a Stream<String> sorted by word
17       Map<String, Long> wordCounts =
18          Files.lines(Paths.get("Chapter2Paragraph.txt"))
19               .flatMap(line -> pattern.splitAsStream(line))
20               .collect(Collectors.groupingBy(String::toLowerCase,
21                  TreeMap::new, Collectors.counting()));
```

Fig. 17.22 | Counting word occurrences in a text file. (Part 1 of 2.)

```
22
23          // display the words grouped by starting letter
24          wordCounts.entrySet()
25             .stream()
26             .collect(
27                Collectors.groupingBy(entry -> entry.getKey().charAt(0),
28                   TreeMap::new, Collectors.toList()))
29             .forEach((letter, wordList) -> {
30                System.out.printf("%n%C%n", letter);
31                wordList.stream().forEach(word -> System.out.printf(
32                   "%13s: %d%n", word.getKey(), word.getValue()));
33             });
34       }
35    }
```

Fig. 17.22 | Counting word occurrences in a text file. (Part 2 of 2.)



```
A                   E                   M                   S
          a: 2            example: 1          make: 1            save: 1
        and: 3           examples: 1      messages: 2           screen: 1
application: 2                                                    show: 1
 arithmetic: 1      F                   N                         sum: 1
                          for: 1             numbers: 2
B                        from: 1                              T
       begin: 1                          O                         that: 3
                                           obtains: 1              the: 7
C                   H                         of: 1            their: 2
 calculates: 1              how: 2              on: 1             then: 2
calculations: 1                            output: 1              this: 2
     chapter: 1     I                                              to: 4
    chapters: 1           inputs: 1        P                      tools: 1
commandline: 1         instruct: 1           perform: 1            two: 2
   compares: 1      introduces: 1           present: 1
  comparison: 1                             program: 1       U
     compile: 1     J                   programming: 1            use: 2
    computer: 1           java: 1          programs: 2           user: 1
D                          jdk: 1
  decisions: 1                                                W
demonstrates: 1     L                   R                         we: 2
     display: 1           last: 1           result: 1           with: 1
    displays: 2          later: 1          results: 2
                         learn: 1              run: 1       Y
                                                          you'll: 2
```

Fig. 17.23 | Output of Fig. 17.22 arranged in three columns.

17.13 Creating a Stream<String> from a File (cont.)

Summarizing the Occurrences of Each Word in the File

- The stream pipeline in lines 17–21 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

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

17.13 Creating a `Stream<String>` from a File (cont.)

- The pipeline performs the following tasks:
  - 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.
  - 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`.)

17.13 Creating a `Stream<String>` from a File (cont.)

- 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`.

17.13 Creating a `Stream<String>` from a File (cont.)

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.getKey().charAt(0),
               TreeMap::new, Collectors.toList()))
          .forEach((letter, wordList) -> {
             System.out.printf("%n%C%n", letter);
             wordList.stream().forEach(word -> System.out.printf(
                "%13s: %d%n", word.getKey(), word.getValue()));
          });
```

- 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`

17.13 Creating a `Stream<String>` from a File (cont.)

- 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>>`.

17.13 Creating a `Stream<String>` from a File (cont.)

- 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>>>`.

17.13 Creating a `Stream<String>` from a File (cont.)

- 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.

17.14 Streams of Random Values

- Figure 6.7 summarized 60,000,000 rolls of a six-sided die using *external iteration* (a `for` loop) and a `switch` statement that determined which counter to increment
- We then displayed the results using separate statements that performed external iteration
- In Fig. 7.7, we reimplemented Fig. 6.7, replacing the entire `switch` statement with a single statement that incremented counters in an array—that version of rolling the die still used external iteration to produce and summarize 60,000,000 random rolls and to display the final results
- Both prior versions of this example used mutable variables to control the external iteration and to summarize the results
- Figure 17.24 reimplements those programs with a *single statement* that does it all, using lambdas, streams, internal iteration and *no mutable variables* to roll the die 60,000,000 times, calculate the frequencies and display the results

Performance Tip 17.3

The techniques that `SecureRandom` uses to produce secure random numbers are significantly slower than those used by `Random` (package `java.util`). For this reason, Fig. 17.24 may appear to freeze when you run it—on our computers, it took over one minute to complete. To save time, you can speed this example's execution by using class `Random`. However, industrial-strength applications should use secure random numbers. Exercise 17.25 asks you to time Fig. 17.24's stream pipeline, then Exercise 23.18 asks you time the pipeline using parallel streams to see if the performance improvems on a multicore system.

```
1  // Fig. 17.24: RandomIntStream.java
2  // Rolling a die 60,000,000 times with streams
3  import java.security.SecureRandom;
4  import java.util.function.Function;
5  import java.util.stream.Collectors;
6
7  public class RandomIntStream {
8     public static void main(String[] args) {
9        SecureRandom random = new SecureRandom();
10
11       // roll a die 60,000,000 times and summarize the results
12       System.out.printf("%-6s%s%n", "Face", "Frequency");
13       random.ints(60_000_000, 1, 7)
14             .boxed()
15             .collect(Collectors.groupingBy(Function.identity(),
16                Collectors.counting()))
17             .forEach((face, frequency) ->
18                System.out.printf("%-6d%d%n", face, frequency));
19    }
20 }
```

Fig. 17.24 | Rolling a die 60,000,000 times with streams. (Part 1 of 2.)

```
Face   Frequency
1      9992993
2      10000363
3      10002272
4      10003810
5      10000321
6      10000241
```

Fig. 17.24 | Rolling a die 60,000,000 times with streams. (Part 2 of 2.)

17.14 Streams of Random Values (cont.)

- Class `SecureRandom` has overloaded methods `ints`, `longs` and `doubles`, which it inherits from class `Random` (package `java.util`)
- These methods return an `IntStream`, a `LongStream` or a `DoubleStream`, respectively, that represent streams of random numbers
- Each method has four overloads

17.14 Streams of Random Values (cont.)

- We describe the `ints` overloads here—methods `longs` and `doubles` perform the same tasks for streams of `long` and `double` values, respectively:
  - `ints()`—creates an `IntStream` for an *infinite stream* (Section 17.15) of random `int` values.
  - `ints(long)`—creates an `IntStream` with the specified number of random `int`s.
  - `ints(int, int)`—creates an `IntStream` for an *infinite stream* of random `int` values in the half-open range starting with the first argument and up to, but not including, the second argument.
  - `ints(long, int, int)`—creates an `IntStream` with the specified number of random `int` values in the range starting with the first argument and up to, but not including, the second argument.
- Line 13 uses the last overloaded version of `ints` (which we introduced in Section 17.6) to create an `IntStream` of 60,000,000 random integer values in the range 1–6.

17.14 Streams of Random Values (cont.)

Converting an `IntStream` to a `Stream<Integer>`

- We summarize the roll frequencies in this example by collecting them into a `Map<Integer, Long>` in which each `Integer` key is a side of the die and each `Long` value is the frequency of that side
- Unfortunately, Java does not allow primitive values in collections, so to summarize the results in a `Map`, we must first convert the `IntStream` to a `Stream<Integer>`
- We do this by calling `IntStream` method `boxed`.

17.14 Streams of Random Values (cont.)

Summarizing the Die Frequencies

- Lines 15–16 call `Stream` method `collect` to summarize the results into a `Map<Integer, Long>`
- The first argument to `Collectors` method `groupingBy` (line 15) calls `static` method `identity` from interface `Function`, which creates a `Function` that simply returns its argument
- This allows the actual random values to be used as the `Map`'s keys
- The second argument to method `groupingBy` counts the number of occurrences of each key.

17.14 Streams of Random Values (cont.)

Displaying the Results

- Lines 17–18 call the resulting `Map`'s `forEach` method to display the summary of the results
- This method receives an object that implements the `BiConsumer` functional interface as an argument
- Recall that for `Map`s, the first parameter represents the key and the second represents the corresponding value
- The lambda in lines 17–18 uses parameter `face` as the key and `frequency` as the value, and displays the face and frequency.

17.15 Infinite Streams

- A data structure, such as an array or a collection, always represents a finite number of elements—all the elements are stored in memory, and memory is finite
- Of course, any stream created from a finite data structure will have a finite number of elements, as has been the case in this chapter's prior examples
- Lazy evaluation makes it possible to work with infinite streams that represent an unknown, potentially infinite, number of elements
- For example, you could define a method `nextPrime` that produces the next prime number in sequence every time you call it
- You could then use this to define an infinite stream that *conceptually* represents all prime numbers
- However, because streams are lazy until you perform a terminal operation, you can use intermediate operations to restrict the total number of elements that are actually calculated when a terminal operation is performed

17.15 Streams of Random Values (cont.)

- Consider the following pseudocode stream pipeline:
  - *Create an infinite stream representing all prime numbers*
  - *If the prime number is less than 10,000*
  - *Display the prime number*
- Even though we begin with an infinite stream, only the finite set of primes less than 10,000 would be displayed
- You create infinite streams with the stream-interfaces methods `iterate` and `generate`
- For the purpose of this discussion, we'll use the `IntStream` version of these methods

17.15 Streams of Random Values (cont.)

IntStream Method *iterate*

- Consider the following infinite stream pipeline:
  - ```
    IntStream.iterate(1, x -> x + 1)
             .forEach(System.out::println);
    ```
- `IntStream` method `iterate` generates an ordered sequence of values starting with the seed value (`1`) in its first argument
- Each subsequent element is produced by applying to the preceding value in the sequence the `-IntUnaryOperator` specified as `iterate`'s second argument
- The preceding pipeline generates the infinite sequence 1, 2, 3, 4, 5, …, but this pipeline has a problem
- We did not specify how many elements to produce, so this is the equivalent of an infinite loop.
    ```

17.15 Streams of Random Values (cont.)

Limiting an Infinite Stream's Number of Elements

- One way to limit the total number of elements that an infinite stream produces is the short-circuiting terminal operation `limit`, which specifies the maximum number of elements to process from a stream
- In the case of an infinite stream, `limit` terminates the infinite generation of elements
- So, the following stream pipeline begins with an infinite stream, but limits the total number of elements produced to 10, so it displays the numbers from 1 through 10

```
IntStream.iterate(1, x -> x + 1)
         .limit(10)
         .forEach(System.out::println);
```

- Similarly, the pipeline starts with an infinite stream, but sums only the squares of the integers from 1 through 10

```
IntStream.iterate(1, x -> x + 1)
         .map(x -> x * x)
         .limit(10)
         .sum()
```

Error-Prevention Tip 17.4

Ensure that stream pipelines using methods that produce infinite streams `limit` the number of elements to produce.

17.15 Streams of Random Values (cont.)

IntStream Method *generate*

- You also may create unordered infinite streams using method `generate`, which receives an `IntSupplier` representing a method that takes no arguments and returns an `int`
- If you have a `SecureRandom` object named `random`, the following stream pipeline generates and displays 10 random integers:

```
IntStream.generate(() -> random.nextInt())
         .limit(10)
         .forEach(System.out::println);
```

- This is equivalent to

```
SecureRandom.ints()
         .limit(10)
         .forEach(System.out::println);
```

17.16 Lambda Event Handlers

- Event-listener interfaces with one `abstract` method—like `ChangeListener`—are functional interfaces
- For such interfaces, you can implement event handlers with lambdas
- Consider the following `Slider` event handler from Fig. 12.23:

```
tipPercentageSlider.valueProperty().addListener(
   new ChangeListener<Number>() {
      @Override
      public void changed(ObservableValue<? extends Number> ov,
         Number oldValue, Number newValue) {
         tipPercentage =
            BigDecimal.valueOf(newValue.intValue() / 100.0);
         tipPercentageLabel.setText(percent.format(tipPercentage));
      }
   }
);
```

17.16 Lambda Event Handlers (cont.)

- The preceding handler be implemented more concisely with a lambda as

```
tipPercentageSlider.valueProperty().addListener(
   (ov, oldValue, newValue) -> {
      tipPercentage =
         BigDecimal.valueOf(newValue.intValue() / 100.0);
      tipPercentageLabel.setText(percent.format(tipPercentage));
   });
```

- For a simple event handler, a lambda significantly reduces the amount of code you need to write

17.17 Additional Notes on Java SE 8 Interfaces

Java SE 8 Interfaces Allow Inheritance of Method Implementations

- Functional interfaces *must* contain only one `abstract` method, but may also contain `default`- methods and `static` methods that are fully implemented in the interface declarations
- For example, the `Function` interface—which is used extensively in functional programming—has methods `apply` (`abstract`), `compose` (`default`), `andThen` (`default`) and `identity` (`static`)
- When a class implements an interface with `default` methods and does *not* override them, the class inherits the `default` methods' implementations

17.17 Additional Notes on Java SE 8 Interfaces (cont.)

- An interface's designer can now evolve an interface by adding new `default` and `static` methods without breaking existing code that implements the interface
- For example, interface `Comparator` (Section 16.7.1) now contains many `default` and `static` methods, but older classes that implement this interface will still compile and operate properly in Java SE 8.
- Recall that one class can implement many interfaces
- If a class implements two or more unrelated interfaces that provide a `default` method with the same signature, the implementing class *must* override that method; otherwise, a compilation error occurs

17.17 Additional Notes on Java SE 8 Interfaces (cont.)

Java SE 8: `@FunctionalInterface` Annotation

- You can create your own functional interfaces by ensuring that each contains only one `abstract`- method and zero or more `default` and/or `static` methods
- Though not required, you can declare that an interface is a functional interface by preceding it with the `@FunctionalInterface` annotation
- The compiler will then ensure that the interface contains only one `abstract`- method; otherwise, it will generate a compilation error.