Chapter 17
Lambdas and Streams

Java How to Program, 11/e, Global Edition

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

17.12 Stream<Employee> Manipulations

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.

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.


```
// Fig. 17.3: StreamReduce.java
// Sum the integers from 1 through 10 with IntStream.
import java.util.stream.IntStream;

public class StreamReduce {
   public static void main(String[] args) {
      // sum the integers from 1 through 10
      System.out.printf("Sum of 1 through 10 is: %d%n",
         IntStream.rangeClosed(1, 10)
                  .sum());
   }
}
```

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

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

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.

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.


```java
// Fig. 17.4: StreamMapReduce.java
// Sum the even integers from 2 through 20 with IntStream.
import java.util.stream.IntStream;

public class StreamMapReduce {
   public static void main(String[] args) {
      // sum the even integers from 2 through 20
      System.out.printf("Sum of the even ints from 2 through 20 is: %d%n",
         IntStream.rangeClosed(1, 10)            // 1...10
                  .map((int x) -> {return x * 2;}) // multiply by 2
                  .sum());                        // sum
   }
}
```

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

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

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.

Performance Tip 17.1

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

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.

```java
// Fig. 17.7: StreamFilterMapReduce.java
// Triple the even ints from 2 through 10 then sum them with IntStream.
import java.util.stream.IntStream;

public class StreamFilterMapReduce {
   public static void main(String[] args) {
      // sum the triples of the even integers from 2 through 10
      System.out.printf(
         "Sum of the triples of the even ints from 2 through 10 is: %d%n",
         IntStream.rangeClosed(1, 10)
                  .filter(x -> x % 2 == 0)
                  .map(x -> x * 3)
                  .sum());
   }
}
```

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

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

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.

```java
// Fig. 17.8: RandomIntegers.java
// Shifted and scaled random integers.
import java.security.SecureRandom;
import java.util.stream.Collectors;

public class RandomIntegers {
   public static void main(String[] args) {
      SecureRandom randomNumbers = new SecureRandom();

      // display 10 random integers on separate lines
      System.out.println("Random numbers on separate lines:");
      randomNumbers.ints(10, 1, 7)
                   .forEach(System.out::println);
```

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


```java

      // display 10 random integers on the same line
      String numbers =
         randomNumbers.ints(10, 1, 7)
                      .mapToObj(String::valueOf)
                      .collect(Collectors.joining(" "));
      System.out.printf("%nRandom numbers on one line: %s%n", numbers);

   }
}
```

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.)


```java
// Fig. 17.9: IntStreamOperations.java
// Demonstrating IntStream operations.
import java.util.Arrays;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class IntStreamOperations {
   public static void main(String[] args) {
      int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

      // display original values
      System.out.print("Original values: ");
      System.out.println(
         IntStream.of(values)
                  .mapToObj(String::valueOf)
                  .collect(Collectors.joining(" ")));
```

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

```java

      // count, min, max, sum and average of the values
      System.out.printf("%nCount: %d%n", IntStream.of(values).count());
      System.out.printf("Min: %d%n",
         IntStream.of(values).min().getAsInt());
      System.out.printf("Max: %d%n",
         IntStream.of(values).max().getAsInt());
      System.out.printf("Sum: %d%n", IntStream.of(values).sum());
      System.out.printf("Average: %.2f%n",
         IntStream.of(values).average().getAsDouble());

```

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


```java
      // sum of values with reduce method
      System.out.printf("%nSum via reduce method: %d%n",
         IntStream.of(values)
                  .reduce(0, (x, y) -> x + y));

      // product of values with reduce method
      System.out.printf("Product via reduce method: %d%n",
         IntStream.of(values)
                  .reduce((x, y) -> x * y).getAsInt());

      // sum of squares of values with map and sum methods
      System.out.printf("Sum of squares via map and sum: %d%n%n",
         IntStream.of(values)
                  .map(x -> x * x)
                  .sum());
```

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

```java

        // displaying the elements in sorted order
        System.out.printf("Values displayed in sorted order: %s%n",
           IntStream.of(values)
                    .sorted()
                    .mapToObj(String::valueOf)
                    .collect(Collectors.joining(" ")));
    }
 }
```

```
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.)

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.

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.

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`.

```java
// Fig. 17.11: ArraysAndStreams.java
// Demonstrating lambdas and streams with an array of Integers.
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class ArraysAndStreams {
   public static void main(String[] args) {
      Integer[] values = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};

      // display original values
      System.out.printf("Original values: %s%n", Arrays.asList(values));

```

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


```java
      // sort values in ascending order with streams
      System.out.printf("Sorted values: %s%n",
         Arrays.stream(values)
               .sorted()
               .collect(Collectors.toList()));

      // values greater than 4
      List<Integer> greaterThan4 =
         Arrays.stream(values)
               .filter(value -> value > 4)
               .collect(Collectors.toList());
      System.out.printf("Values greater than 4: %s%n", greaterThan4);

```

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

```java
        // filter values greater than 4 then sort the results
        System.out.printf("Sorted values greater than 4: %s%n",
           Arrays.stream(values)
                 .filter(value -> value > 4)
                 .sorted()
                 .collect(Collectors.toList()));

        // greaterThan4 List sorted with streams
        System.out.printf(
           "Values greater than 4 (ascending with streams): %s%n",
           greaterThan4.stream()
                 .sorted()
                 .collect(Collectors.toList()));
     }
  }
```

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.)

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.


```java
// Fig. 17.12: ArraysAndStreams2.java
// Demonstrating lambdas and streams with an array of Strings.
import java.util.Arrays;
import java.util.Comparator;
import java.util.stream.Collectors;

public class ArraysAndStreams2 {
   public static void main(String[] args) {
      String[] strings =
         {"Red", "orange", "Yellow", "green", "Blue", "indigo", "Violet"};

      // display original strings
      System.out.printf("Original strings: %s%n", Arrays.asList(strings));

```

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

```java
        // strings in uppercase
        System.out.printf("strings in uppercase: %s%n",
           Arrays.stream(strings)
                 .map(String::toUpperCase)
                 .collect(Collectors.toList()));

        // strings less than "n" (case insensitive) sorted ascending
        System.out.printf("strings less than n sorted ascending: %s%n",
           Arrays.stream(strings)
                 .filter(s -> s.compareToIgnoreCase("n") < 0)
                 .sorted(String.CASE_INSENSITIVE_ORDER)
                 .collect(Collectors.toList()));

```

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


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

```
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.)

```java
// Fig. 17.13: Employee.java
// Employee class.
public class Employee {
   private String firstName;
   private String lastName;
   private double salary;
   private String department;

   // constructor
   public Employee(String firstName, String lastName,
      double salary, String department) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.salary = salary;
      this.department = department;
   }
```

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


```java

   // get firstName
   public String getFirstName() {
      return firstName;
   }

   // get lastName
   public String getLastName() {
      return lastName;
   }

   // get salary
   public double getSalary() {
      return salary;
   }

```

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

```java
    // get department
    public String getDepartment() {
       return department;
    }

    // return Employee's first and last name combined
    public String getName() {
       return String.format("%s %s", getFirstName(), getLastName());
    }

    // return a String containing the Employee's information
    @Override
    public String toString() {
       return String.format("%-8s %-8s %8.2f   %s",
          getFirstName(), getLastName(), getSalary(), getDepartment());
    }
}
```

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


```java
// Fig. 17.14: ProcessingEmployees.java
// Processing streams of Employee objects.
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;

```

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

```java
public class ProcessingEmployees {
   public static void main(String[] args) {
      // initialize array of Employees
      Employee[] employees = {
         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")};

      // get List view of the Employees
      List<Employee> list = Arrays.asList(employees);

      // display all Employees
      System.out.println("Complete Employee list:");
      list.stream().forEach(System.out::println);

```

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.)

```java
        // Predicate that returns true for salaries in the range $4000-$6000
        Predicate<Employee> fourToSixThousand =
           e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);

        // Display Employees with salaries in the range $4000-$6000
        // sorted into ascending order by salary
        System.out.printf(
           "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
        list.stream()
            .filter(fourToSixThousand)
            .sorted(Comparator.comparing(Employee::getSalary))
            .forEach(System.out::println);

        // Display first Employee with salary in the range $4000-$6000
        System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
           list.stream()
               .filter(fourToSixThousand)
               .findFirst()
               .get());

```

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.)

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.

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.

```java
        // Functions for getting first and last names from an Employee
        Function<Employee, String> byFirstName = Employee::getFirstName;
        Function<Employee, String> byLastName = Employee::getLastName;

        // Comparator for comparing Employees by first name then last name
        Comparator<Employee> lastThenFirst =
           Comparator.comparing(byLastName).thenComparing(byFirstName);

        // sort employees by last name, then first name
        System.out.printf(
           "%nEmployees in ascending order by last name then first:%n");
        list.stream()
            .sorted(lastThenFirst)
            .forEach(System.out::println);

        // sort employees in descending order by last name, then first name
        System.out.printf(
           "%nEmployees in descending order by last name then first:%n");
        list.stream()
            .sorted(lastThenFirst.reversed())
            .forEach(System.out::println);

```

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.)

```java
        // display unique employee last names sorted
        System.out.printf("%nUnique employee last names:%n");
        list.stream()
            .map(Employee::getLastName)
            .distinct()
            .sorted()
            .forEach(System.out::println);

        // display only first and last names
        System.out.printf(
           "%nEmployee names in order by last name then first name:%n");
        list.stream()
            .sorted(lastThenFirst)
            .map(Employee::getName)
            .forEach(System.out::println);

```

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.)

```
      // group Employees by department
      System.out.printf("%nEmployees by department:%n");
      Map<String, List<Employee>> groupedByDepartment =
         list.stream()
             .collect(Collectors.groupingBy(Employee::getDepartment));
      groupedByDepartment.forEach(
         (department, employeesInDepartment) -> {
            System.out.printf("%n%s%n", department);
            employeesInDepartment.forEach(
               employee -> System.out.printf("   %s%n", employee));
         }
      );

```

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.)

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

```

```
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.


```java
        // sum of Employee salaries with DoubleStream sum method
        System.out.printf(
           "%nSum of Employees' salaries (via sum method): %.2f%n",
           list.stream()
               .mapToDouble(Employee::getSalary)
               .sum());

        // calculate sum of Employee salaries with Stream reduce method
        System.out.printf(
           "Sum of Employees' salaries (via reduce method): %.2f%n",
           list.stream()
               .mapToDouble(Employee::getSalary)
               .reduce(0, (value1, value2) -> value1 + value2));

```

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

```java
        // average of Employee salaries with DoubleStream average method
        System.out.printf("Average of Employees' salaries: %.2f%n",
           list.stream()
               .mapToDouble(Employee::getSalary)
               .average()
               .getAsDouble());
    }
}
```

```
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.)


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

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

      // count occurrences of each word in a Stream<String> sorted by 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()));
```

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

```java

      // display the words grouped by starting 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()));
         });
   }
}
```

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


```
A
            a: 2
          and: 3
  application: 2
   arithmetic: 1

B
        begin: 1

C
   calculates: 1
 calculations: 1
      chapter: 1
     chapters: 1
  commandline: 1
     compares: 1
   comparison: 1
      compile: 1
     computer: 1

D
    decisions: 1
 demonstrates: 1
      display: 1
     displays: 2

E
      example: 1
     examples: 1

F
          for: 1
         from: 1

H
          how: 2

I
       inputs: 1
     instruct: 1
   introduces: 1

J
         java: 1
          jdk: 1

L
         last: 1
        later: 1
        learn: 1

M
         make: 1
     messages: 2

N
      numbers: 2

O
      obtains: 1
           of: 1
           on: 1
       output: 1

P
      perform: 1
      present: 1
      program: 1
  programming: 1
     programs: 2

R
       result: 1
      results: 2
          run: 1

S
         save: 1
       screen: 1
         show: 1
          sum: 1

T
         that: 3
          the: 7
        their: 2
         then: 2
         this: 2
           to: 4
        tools: 1
          two: 2

U
          use: 2
         user: 1

W
           we: 2
         with: 1

Y
       you'll: 2
```

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

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.


```java
// Fig. 17.24: RandomIntStream.java
// Rolling a die 60,000,000 times with streams
import java.security.SecureRandom;
import java.util.function.Function;
import java.util.stream.Collectors;

public class RandomIntStream {
   public static void main(String[] args) {
      SecureRandom random = new SecureRandom();

      // roll a die 60,000,000 times and summarize the results
      System.out.printf("%-6s%s%n", "Face", "Frequency");
      random.ints(60_000_000, 1, 7)
            .boxed()
            .collect(Collectors.groupingBy(Function.identity(),
               Collectors.counting()))
            .forEach((face, frequency) ->
               System.out.printf("%-6d%d%n", face, frequency));
   }
}
```

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.)

Error-Prevention Tip 17.4

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