9b

Streams API

OBJECTIVES

In this lecture you will learn:

- **The advantages of using the Streams API**
- **The difference between external and internal iteration**
- **The basic characteristics of the Stream API**
- **Intermediate and terminal operations to perform IntStream and Stream operations**

9b.1 Introduction

A common pattern for Java developers when working with collections is to **iterate over a collection**, operating on **each element in turn**. For example, if we wanted to add up the number of students who are from Sofia, we would write the code:

```
Iterator<Student> it = studentList.iterator();
int count = 0;
while (it.hasNext()) {
    Student student = it.next();
    if (student.isFrom("Sofia"))  {
        count++;
    }
}
```

9b.1 Introduction

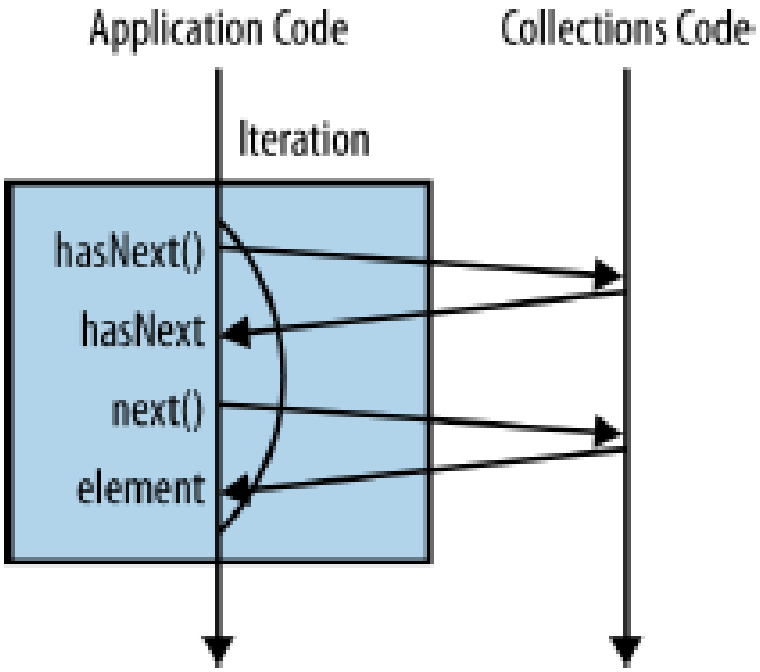
There are three major problems with the above approach:

1.We just want to count the number of students, but we would also have to provide how the iteration will take place. This is also called external iteration because the client program is handling the algorithm to iterate over the list.

2.The program is sequential in nature, there is no way we can do this in parallel easily.

3.There is a lot of boilerplate code that **needs to be written every time you want to iterate** over the collection

External iteration



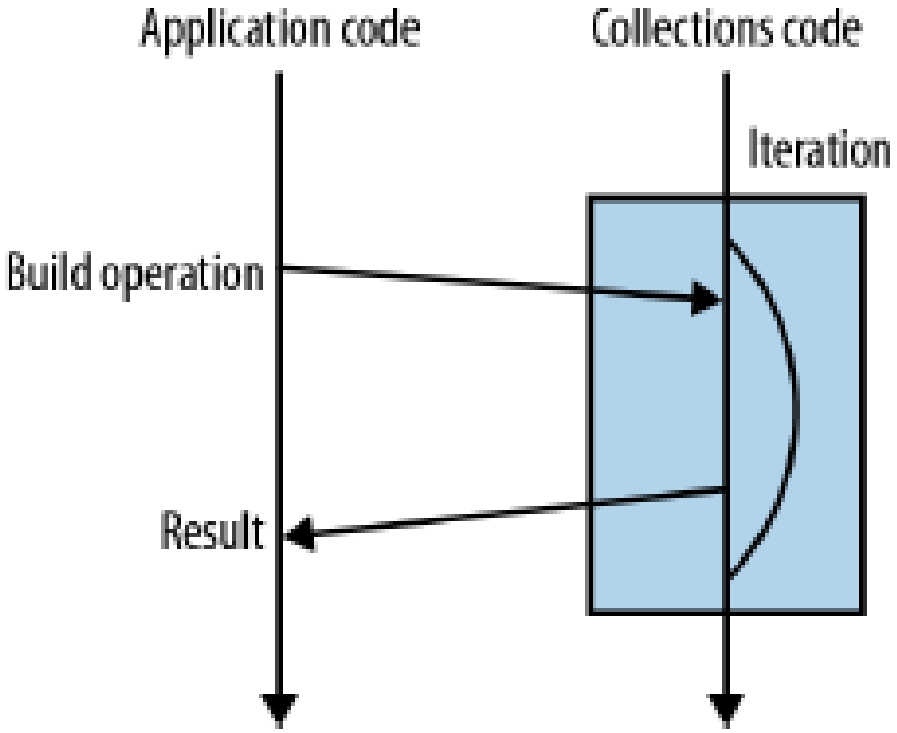
9b.1 Introduction

To overcome all the above shortcomings, Java 8 introduces Stream API. We can use Stream API to implement internal iteration, that is better because the Java framework is in control of the iteration.

Internal iteration provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc.

Most of the Stream API method arguments are functional interfaces, so lambda expressions work very well with them. Let's see how can we write above logic in a single line statement

Internal iteration



9b.1 Introduction

```
long studentsFromSofia =
    studentList.stream()
    .filter(student ->student.isFrom("Sofia"))
    .count();
```

The first thing to notice is the call to `stream()`, which performs a similar role to the call to `iterator()` in the previous example. Instead of returning an `Iterator` to control the iteration, it returns the equivalent `interface` in the internal iteration world: `Stream`

A Stream is a tool for building up complex operations on collections using a functional approach.

9b.1 Introduction

We actually **break this example into two simpler operations**:

- **Finding** all the students from Sofia
- **Counting** a filtered list of students

Both of these **operations correspond** to a **method** on the **`Stream`** interface. In order to **find** students from Sofia, we **filter** the **`Stream`**. **Filtering** in this case means "***keep only objects that pass a test.***" The test is defined by the **`filter`**() function, which returns either **`true`** or **`false`** depending on whether the student is from Sofia. The **`count`**() method **counts** how many objects are in a given **`Stream`**.

9b.1 Introduction

The `Stream` object returned **isn't a new** collection- **it's a recipe for creating a new collection**.

The call to `filter`() builds up a `Stream` **recipe**, but there's nothing to force this recipe to be used.

Methods such as `filter()` that **build up the `Stream` recipe but don't force a new value** to be generated at the end are referred to as ***lazy (*intermediate *operations)***.

Methods such as count that **generate a final value** out of the `Stream` sequence are called ***eager (*terminal *operations)***.

9b.1 Introduction

Streams are not collections- they do not manage their own data. Instead, **they are wrappers around existing data structures**.

- When you **make or transform** a Stream, **it does not copy the underlying data**. Instead, it just **builds a pipeline of operations**. How many times that pipeline will be invoked depends on what you later do with the stream (find the first element, skip some elements, see if all elements match a Predicate, etc.)
- With the `Streams` API, we **aren't changing the contents** of the `Collection`. We're **just declaring** what the contents of the `Stream` will be.

9b.1 Introduction

An intermediate operation **specifies tasks to perform on the stream's elements** and **always** results in a **new Stream**.

Intermediate operations are **lazy**, they **aren't performed until a terminal operation is invoked**.

- Allows library developers to optimize stream-processing performance

Terminal operation are eager, they perform the requested operation **when they are called**

- **initiates processing** of a **Stream pipeline's** intermediate operations
- **produces a result**

9b.1 Introduction

It's very **easy to figure out** whether an operation is **eager** or **lazy- look at what it returns**.

If it **gives you back** a Stream, it's **lazy**; if it **gives you back another value or `void`**, then it's **eager**. This makes sense because **the preferred way of using these methods** is to form **a sequence of lazy operations chained together** and then to **have a single eager operation at the end** that generates your result.

9b.1 Introduction

The call to filter builds up a `Stream` recipe, but there's nothing to force this recipe to be used.

```
studentList.stream()
  .filter(student -> student.isFrom("Sofia"));
```

Not printing out artist names due to lazy evaluation

```
studentList.stream()
.filter(student -> {
      System.out.println(student.getName());
      return student.isFrom("Sofia");
    });
```

9b.1 Introduction

Printing out artist names

```
long studentsFromSofia.stream()
.filter(student -> {
    System.out.println(student.getName());
    return student.isFrom("Sofia");
    })
.count();
```

9b.2 Characteristics of Streams

Streams are not data structures

– Streams have *no* storage. They **carry values from a source through a pipeline of operations** and values **computed on-demand**

– They also **never modify the underlying data structure** (e.g., the List or array that the Stream wraps)

Designed for lambdas

– All Stream operations **use functional interfaces and** lambdas as arguments

Do not support indexed access

– You can ask for the first element, but not the second or third or last element.

Can easily be output as arrays or Lists

– Simple syntax to build an array or List from a Stream

9b.2 Characteristics of Streams

Lazy

– The **chain of intermediate operation is postponed** until the terminal operation operation is executed.

Parallelizable

– Stream **supports sequential as well as parallel processing**, parallel processing can be very helpful in achieving high performance for large collections, without **having to write explicit multi-threading code**

Can be unbounded

– Unlike with collections, you can **designate a generator** function, and clients can consume entries as long as they want. Unlike an iterator, it generates the next value when needed, rather than returning the next item of a pre- generated collection. (http://www.codeproject.com/Articles/793374/Generators-with-Java)

9b.3 `IntStream` Operations

- **Figure 17.5 demonstrates operations on an `IntStream` (package `java.util.stream`)- a specialized stream for manipulating `int` values.**
- **The techniques shown in this example also apply to `LongStreams` and `DoubleStreams` for `long` and `double` values, respectively.**

9b.3.1 Creating an IntStream and Displaying Its Values with the forEach Terminal Operation

- IntStream static method of receives an int array as an argument and returns an IntStream for processing the array's values.
- IntStream method forEach (a terminal-operation) receives as its argument an object that implements the IntConsumer functional interface (package java.util.function). This interface's accept method receives one int value and performs a task with it.

9b.3 `IntStream` Operations

```
1  // Fig. 17.5: IntStreamOperations.java
2  // Demonstrating IntStream operations.
3  import java.util.Arrays;
4  import java.util.stream.IntStream;
5
6  public class IntStreamOperations
7  {
8     public static void main(String[] args)
9     {
10       int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
11
12       // display original values
13       System.out.print("Original values: ");
14       IntStream.of(values)
15                .forEach(value -> System.out.printf("%d ", value));
16       System.out.println();
17
```

Fig. 17.5 | Demonstrating `IntStream` operations. (Part 1 of 5.)

9b.3 `IntStream` Operations

```
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
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          // sum of squares of values with reduce method
34          System.out.printf("Sum of squares via reduce method: %d%n",
35             IntStream.of(values)
36                      .reduce(0, (x, y) -> x + y * y));
```

Fig. 17.5 | Demonstrating `IntStream` operations. (Part 2 of 5.)

9b.3.2 Terminal Operations `count`, `min`, `max`, `sum` and `average`

- Class `IntStream` provides terminal operations for common stream reductions
  - `count` returns the number of elements
  - `min` returns the smallest int
  - `max` returns the largest int
  - `sum` returns the sum of all the ints
  - `average` returns an `OptionalDouble` (package `java.util`) containing the average of the `int`s as a value of type `double`
- Class `OptionalDouble`'s `getAsDouble` method returns the `double` in the object or throws a `NoSuchElementException`.
  - To prevent this exception, you can call method `orElse`, which returns the `OptionalDouble`'s value if there is one, or the value you pass to `orElse`, otherwise.

9b.3.2 Terminal Operations `count`, `min`, `max`, `sum` and `average`

- `IntStream` method `summaryStatistics` 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`).

9b.3.2 Terminal Operations count, min, max, sum and average

```
IntSummaryStatistics stats = IntStream.of(values)
                                      .summaryStatistics();
    // average
    System.out.println(stats.getAverage());
    // count
    System.out.println(stats.getCount());
     // max
    System.out.println(stats.getMax());
    // min
    System.out.println(stats.getMin());
    // sum
    System.out.println(stats.getSum());
```

9b.3.3 Terminal Operation `reduce`

- **You can define your own reductions for an `IntStream` by calling its `reduce` method.**
  - **First argument is a value that helps you begin the reduction operation**
  - **Second argument is an object that implements the `IntBinaryOperator` functional interface**
- **Method `reduce`'s first argument is formally called an identity value- a value that, when combined with any stream element using the `IntBinaryOperator` produces that element's original value.**

9b.3.3 Terminal Operation `reduce`

- **You can define your own reductions for an `IntStream` by calling its `reduce` method.**
  - **First argument is a value that helps you begin the reduction operation**
  - **Second argument is an object that implements the `IntBinaryOperator` functional interface**
- **Method `reduce`'s first argument is formally called an identity value- a value that, when combined with any stream element using the `IntBinaryOperator` produces that element's original value.**

9b.3.4 Intermediate Operations: Filtering and Sorting `IntStream` Values

```
37
38          // product of values with reduce method
39          System.out.printf("Product via reduce method: %d%n",
40             IntStream.of(values)
41                      .reduce(1, (x, y) -> x * y));
42
43          // even values displayed in sorted order
44          System.out.printf("%nEven values displayed in sorted order: ");
45          IntStream.of(values)
46                   .filter(value -> value % 2 == 0)
47                   .sorted()
48                   .forEach(value -> System.out.printf("%d ", value));
49          System.out.println();
50
51          // odd values multiplied by 10 and displayed in sorted order
52          System.out.printf(
53             "Odd values multiplied by 10 displayed in sorted order: ");
54          IntStream.of(values)
55                   .filter(value -> value % 2 != 0)
56                   .map(value -> value * 10)
57                   .sorted()
58                   .forEach(value -> System.out.printf("%d ", value));
59          System.out.println();
60
```

Fig. 17.5 | Demonstrating `IntStream` operations. (Part 3 of 5.)

9b.3.4 Intermediate Operations: Filtering and Sorting `IntStream` Values

- Filter elements to produce a stream of intermediate results that match a predicate.
- `IntStream` method `filter` receives an object that implements the `IntPredicate` functional interface (package `java.util.function`).
- `IntStream` method `sorted` (a lazy operation) orders the elements of the stream into ascending order (by default).
  - All prior intermediate operations in the stream pipeline must be complete so that method `sorted` knows which elements to sort.

9b.3.4 Intermediate Operations: Filtering and Sorting IntStream Values

- Method `filter` a `stateless` intermediate operation- it does not require any information about other elements in the stream in order to test whether the current element satisfies the predicate.
- Method `sorted` is a `stateful` intermediate operation that requires information about all of the other elements in the stream in order to sort them.
- The interface `IntPredicate`'s `default` method `and` performs a logical AND operation with short-circuit evaluation between the `IntPredicate` on which it's called and its `IntPredicate` argument.

9b.3.4 Intermediate Operations: Filtering and Sorting IntStream Values

- Interface `IntPredicate`'s `default` method `negate` reverses the `boolean` value of the `IntPredicate` on which it's called.
- Interface `IntPredicate` `default` method `or` performs a logical `OR` operation with short-circuit evaluation between the `IntPredicate` on which it's called and its `IntPredicate` argument.
- You can use the interface `IntPredicate` `default` methods to compose more complex conditions.

9b.3.5 Intermediate Operation: Mapping

- **Mapping is an intermediate operation that transforms a stream's elements to new values and produces a stream containing the resulting (possibly different type) elements.**
- **`IntStream` method `map` (a stateless intermediate operation) receives an object that implements the `IntUnaryOperator` functional interface (package `java-.util.function`).**

9b.3.6 Creating Streams of `int`s with `IntStream` Methods `range` and `rangeClosed`

- `IntStream` methods `range` and `rangeClosed` each produce an ordered sequence of `int` values.
  - Both methods take two `int` arguments representing the range of values.
  - Method `range` produces a sequence of values from its first argument up to, but not including, its second argument.
  - Method `rangeClosed` produces a sequence of values including both of its arguments.

9b.3.6 Creating Streams of `int`s with `IntStream` Methods `range` and `rangeClosed`

```
61          // sum range of integers from 1 to 10, exlusive
62          System.out.printf("%nSum of integers from 1 to 9: %d%n",
63             IntStream.range(1, 10).sum());
64
65          // sum range of integers from 1 to 10, inclusive
66          System.out.printf("Sum of integers from 1 to 10: %d%n",
67             IntStream.rangeClosed(1, 10).sum());
68       }
69    } // end class IntStreamOperations
```

Fig. 17.5 | Demonstrating `IntStream` operations. (Part 4 of 5.)

9b.3 `IntStream` Operations

```
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
Sum of squares via reduce method: 385
Product via reduce method: 3628800

Even values displayed in sorted order: 2 4 6 8 10
Odd values multiplied by 10 displayed in sorted order: 10 30 50 70 90

Sum of integers from 1 to 9: 45
Sum of integers from 1 to 10: 55
```

9b.3.4 Intermediate Operations: Filtering and Sorting IntStream Values

Fig. 17.5 | Demonstrating `IntStream` operations. (Part 5 of 5.)

9b.4 Stream<Integer> Manipulations

- Class Array's stream method is used to create a Stream from an array of objects.
- Class Arrays provides overloaded stream methods for creating IntStreams, LongStreams and DoubleStreams from int, long and double arrays or from ranges of elements in the arrays.

9b.4 Stream<Integer> Manipulations

```
1  // Fig. 17.6: ArraysAndStreams.java
2  // Demonstrating lambdas and streams with an array of Integers.
3  import java.util.Arrays;
4  import java.util.Comparator;
5  import java.util.List;
6  import java.util.stream.Collectors;
7
8  public class ArraysAndStreams
9  {
10    public static void main(String[] args)
11    {
12       Integer[] values = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};
13
14       // display original values
15       System.out.printf("Original values: %s%n", Arrays.asList(values));
16
17       // sort values in ascending order with streams
18       System.out.printf("Sorted values: %s%n",
19          Arrays.stream(values)
20                .sorted()
21                .collect(Collectors.toList()));
22
```

Fig. 17.6 | Demonstrating lambdas and streams with an array of `Integers`. (Part 1 of 3.)

9b.4.1 Creating a `Stream<Integer>`

- **interface `Stream` (package `java.util.stream`) is a generic interface for performing stream operations on objects.**
- **The types of objects that are processed are determined by the `Stream`'s source**

9b.4.2 Sorting a `Stream` and Collecting the Results

- `Stream` method `sorted` sorts a stream's elements into ascending order by default.
- To create a collection containing a stream pipeline's results, you can use `Stream` method `collect` (a terminal operation).
  - As the stream pipeline is processed, method `collect` performs a mutable reduction operation that places the results into an object, such as a `List`, `Map` or `Set`.
- Method `collect` with one argument receives an object that implements interface `Collector` (package `java.util.stream`), which specifies how to perform the mutable reduction.

9b.4.2 Sorting a `Stream` and Collecting the Results

```
23          // values greater than 4
24          List<Integer> greaterThan4 =
25             Arrays.stream(values)
26                   .filter(value -> value > 4)
27                   .collect(Collectors.toList());
28          System.out.printf("Values greater than 4: %s%n", greaterThan4);
29
30          // filter values greater than 4 then sort the results
31          System.out.printf("Sorted values greater than 4: %s%n",
32             Arrays.stream(values)
33                   .filter(value -> value > 4)
34                   .sorted()
35                   .collect(Collectors.toList()));
36
37          // greaterThan4 List sorted with streams
38          System.out.printf(
39             "Values greater than 4 (ascending with streams): %s%n",
40             greaterThan4.stream()
41                   .sorted()
42                   .collect(Collectors.toList()));
43       }
44    } // end class ArraysAndStreams
```

Fig. 17.6 | Demonstrating lambdas and streams with an array of `Integer`s. (Part 2 of 3.)

9b.4.2 Sorting a `Stream` and Collecting the Results

```
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. 17.6 | Demonstrating lambdas and streams with an array of `Integer`s. (Part 3 of 3.)

9b.4.2 Sorting a `Stream` and Collecting the Results

- Class `Collectors` (package `java.util.stream`) provides `static` methods that return predefined `Collector` implementations.
- `Collectors` method `toList` transforms a `Stream<T>` into a `List<T>` collection.

9b.4.3 Filtering a `Stream` and Storing the Results for Later Use

- `Stream` method `filter` receives a `Predicate` and results in a stream of objects that match the `Predicate`.
- `Predicate` method `test` returns a `boolean` indicating whether the argument satisfies a condition.
- Interface `Predicate` also has methods `and`, `negate` and `or`.

9b.4.4 Sorting Previously Collected Results

- **Once you place the results of a stream pipeline into a collection, you can create a new stream from the collection for performing additional stream operations on the prior results.**

Mistakes with Streams

Streams are consumable, so there **is no way to create a reference to stream for future usage**. Since the data is on-demand, it's **not possible to reuse the same stream multiple times**.

The following code won't work:

```
IntStream stream = IntStream.of(1, 2);
stream.forEach(System.out::println);
// Let's do it again!
stream.forEach(System.out::println);
```

You'll get a

```
java.lang.IllegalStateException:
stream has already been operated upon or
closed
```

9b.5 Stream<String> Manipulations

Figure 17.7 performs some of the same stream operations you learned in using a `Stream<String>`.

9b.5 Stream<String> Manipulations

```
1  // Fig. 17.7: 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  {
9     public static void main(String[] args)
10    {
11       String[] strings =
12          {"Red", "orange", "Yellow", "green", "Blue", "indigo", "Violet"};
13
14       // display original strings
15       System.out.printf("Original strings: %s%n", Arrays.asList(strings));
16
17       // strings in uppercase
18       System.out.printf("strings in uppercase: %s%n",
19          Arrays.stream(strings)
20                .map(String::toUpperCase)
21                .collect(Collectors.toList()));
22
```

Fig. 17.7 | Demonstrating lambdas and streams with an array of `Strings`. (Part 1 of 2.)

9b.5.1 Mapping `Strings` to Uppercase Using a Method Reference

- `Stream` method `map` maps each element to a new value and produces a new stream with the same number of elements as the original stream.
- Line 20 uses method reference to method `toUpperCase` of `class String`.
- `ClassName::instanceMethodName` represents a method reference for an instance method of a class.
  - Creates a one-parameter lambda that invokes the instance method on the lambda's argument and returns the method's result.

9b.5 Stream<String> Manipulations

```
23          // strings less than "n" (case insensitive) sorted ascending
24          System.out.printf("strings greater than m sorted ascending: %s%n",
25             Arrays.stream(strings)
26                   .filter(s -> s.compareToIgnoreCase("n") < 0)
27                   .sorted(String.CASE_INSENSITIVE_ORDER)
28                   .collect(Collectors.toList()));
29
30          // strings less than "n" (case insensitive) sorted descending
31          System.out.printf("strings greater than m sorted descending: %s%n",
32             Arrays.stream(strings)
33                   .filter(s -> s.compareToIgnoreCase("n") < 0)
34                   .sorted(String.CASE_INSENSITIVE_ORDER.reversed())
35                   .collect(Collectors.toList()));
36       }
37    } // end class ArraysAndStreams2
```

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

Fig. 17.7 | Demonstrating lambdas and streams with an array of `Strings`. (Part 2 of 2.)

9b.5.2 Filtering Strings Then Sorting Them in Case-Insensitive Ascending Order

- Stream method sorted can receive a Functional interface Comparator as an argument to specify how to compare stream elements for sorting.
- By default, method sorted uses the natural order for the stream's element type.
- For Strings, 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.

9b.5.3 Filtering `Strings` Then Sorting Them in Case-Insensitive Descending Order

- **Functional interface `Comparator`'s `default` method `reversed` reverses an existing `Comparator`'s ordering.**

9b.6 Stream<Employee> Manipulations

- **The example in Figs. 17.9–17.16 demonstrates various lambda and stream capabilities using a `Stream<Employee>`.**
- **Class `Employee` (Fig. 17.9) represents an employee with a first name, last name, salary and department and provides methods for manipulating these values.**

9b.6 Stream<Employee> Manipulations

```
1  // Fig. 17.9: Employee.java
2  // Employee class.
3  public class Employee
4  {
5     private String firstName;
6     private String lastName;
7     private double salary;
8     private String department;
9
10    // constructor
11    public Employee(String firstName, String lastName,
12       double salary, String department)
13    {
14       this.firstName = firstName;
15       this.lastName = lastName;
16       this.salary = salary;
17       this.department = department;
18    }
19
20    // set firstName
21    public void setFirstName(String firstName)
22    {
23       this.firstName = firstName;
24    }
```

Fig. 17.9 | Employee class for use in Figs. 17.10–17.16. (Part 1 of 4.)

9b.6 Stream<Employee> Manipulations

```
25
26      // get firstName
27      public String getFirstName()
28      {
29         return firstName;
30      }
31
32      // set lastName
33      public void setLastName(String lastName)
34      {
35         this.lastName = lastName;
36      }
37
38      // get lastName
39      public String getLastName()
40      {
41         return lastName;
42      }
43
44      // set salary
45      public void setSalary(double salary)
46      {
47         this.salary = salary;
48      }
```

Fig. 17.9 | Employee class for use in Figs. 17.10–17.16. (Part 2 of 4.)

9b.6 Stream<Employee> Manipulations

```
49
50      // get salary
51      public double getSalary()
52      {
53         return salary;
54      }
55
56      // set department
57      public void setDepartment(String department)
58      {
59         this.department = department;
60      }
61
62      // get department
63      public String getDepartment()
64      {
65         return department;
66      }
67
68      // return Employee's first and last name combined
69      public String getName()
70      {
71         return String.format("%s %s", getFirstName(), getLastName());
72      }
```

Fig. 17.9 | `Employee` class for use in Figs. 17.10–17.16. (Part 3 of 4.)

9b.6 Stream<Employee> Manipulations

```
73
74      // return a String containing the Employee's information
75      @Override
76      public String toString()
77      {
78         return String.format("%-8s %-8s %8.2f   %s",
79            getFirstName(), getLastName(), getSalary(), getDepartment());
80      } // end method toString
81   } // end class Employee
```

Fig. 17.9 | `Employee` class for use in Figs. 17.10–17.16. (Part 4 of 4.)

9b.6.1 Creating and Displaying a `List<Employee>`

- When the instance method reference `System.out::println` is passed to `Stream` method `forEach`, it's converted by the compiler into an object that implements the `Consumer` functional interface.
  - This interface's `accept` method receives one argument and returns `void`. In this case, the `accept` method passes the argument to the `System.out` object's `println` instance method.
- Class `ProcessingEmployees` (Figs. 17.10–17.16) is split into several figures so we can show you the lambda and streams operations with their corresponding outputs.
- Figure 17.10 creates an array of `Employees` and gets its `List` view.

9b.6 Stream<Employee> Manipulations

```
1  // Fig. 17.10: 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
12 public class ProcessingEmployees
13 {
14    public static void main(String[] args)
15    {
16       // initialize array of Employees
17       Employee[] employees = {
18          new Employee("Jason", "Red", 5000, "IT"),
19          new Employee("Ashley", "Green", 7600, "IT"),
20          new Employee("Matthew", "Indigo", 3587.5, "Sales"),
21          new Employee("James", "Indigo", 4700.77, "Marketing"),
22          new Employee("Luke", "Indigo", 6200, "IT"),
23          new Employee("Jason", "Blue", 3200, "Sales"),
24          new Employee("Wendy", "Brown", 4236.4, "Marketing")};
```

Fig. 17.10 | Creating an array of `Employees`, converting it to a `List` and displaying the `List`. (Part 1 of 2.)

9b.6 Stream<Employee> Manipulations

```
25
26         // get List view of the Employees
27         List<Employee> list = Arrays.asList(employees);
28
29         // display all Employees
30         System.out.println("Complete Employee list:");
31         list.stream().forEach(System.out::println);
32
```

```
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. 17.10 | Creating an array of `Employee`s, converting it to a `List` and displaying the `List`. (Part 2 of 2.)

9b.6.2 Filtering Employees with Salaries in a Specified Range

- Figure 17.11 demonstrates filtering `Employees` with an object that implements the functional interface `Predicate<Employee>`, which is defined with a lambda
- To reuse a lambda, you can assign it to a variable of the appropriate functional interface type.
- The `Comparator` interface's `static` method `comparing` receives a `Function` that's used to extract a value from an object in the stream for use in comparisons and returns a `Comparator` object.

9b.6 Stream<Employee> Manipulations

```
33          // Predicate that returns true for salaries in the range $4000-$6000
34          Predicate<Employee> fourToSixThousand =
35             e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);
36
37          // Display Employees with salaries in the range $4000-$6000
38          // sorted into ascending order by salary
39          System.out.printf(
40             "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
41          list.stream()
42              .filter(fourToSixThousand)
43              .sorted(Comparator.comparing(Employee::getSalary))
44              .forEach(System.out::println);
45
46          // Display first Employee with salary in the range $4000-$6000
47          System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
48             list.stream()
49                 .filter(fourToSixThousand)
50                 .findFirst()
51                 .get());
52
```

Fig. 17.11 | Filtering `Employees` with salaries in the range $4000–$6000. (Part 1 of 2.)

9b.6 Stream<Employee> Manipulations

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

9b.6.2 Filtering `Employees` with Salaries in a Specified Range

- A nice performance feature 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.
- `Stream` method `findFirst` is a short-circuiting terminal operation that processes the stream pipeline and terminates processing as soon as the first object from the stream pipeline is found.
  - Returns an `Optional` containing the object that was found, if any.

9b.6.3 Sorting Employees By Multiple Fields

Figure 17.12 shows how to use streams to sort objects by *multiple* fields.

To sort objects by two fields, you create a `Comparator` that uses two `Functions`.

First you call `Comparator` method `comparing` to create a `Comparator` with the first `Function`.

On the resulting `Comparator`, you call method `thenComparing` with the second `Function`.

The resulting `Comparator` compares objects using the first `Function` then, for objects that are equal, compares them by the second `Function`.

9b.6 Stream<Employee> Manipulations

```
53          // Functions for getting first and last names from an Employee
54          Function<Employee, String> byFirstName = Employee::getFirstName;
55          Function<Employee, String> byLastName = Employee::getLastName;
56
57          // Comparator for comparing Employees by first name then last name
58          Comparator<Employee> lastThenFirst =
59             Comparator.comparing(byLastName).thenComparing(byFirstName);
60
61          // sort employees by last name, then first name
62          System.out.printf(
63             "%nEmployees in ascending order by last name then first:%n");
64          list.stream()
65              .sorted(lastThenFirst)
66              .forEach(System.out::println);
67
68          // sort employees in descending order by last name, then first name
69          System.out.printf(
70             "%nEmployees in descending order by last name then first:%n");
71          list.stream()
72              .sorted(lastThenFirst.reversed())
73              .forEach(System.out::println);
74
```

Fig. 17.12 | Sorting `Employees` by last name then first name. (Part 1 of 2.)

9b.6 Stream<Employee> Manipulations

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

9b.6.4 Mapping Employees to Unique Last Name Strings

Figure 17.13 shows how to map objects of one type (`Employee`) to objects of a different type (`String`).

You can map objects in a stream to different types to produce another stream with the same number of elements as the original stream.

`Stream` method `distinct` eliminates duplicate objects in a stream.

9b.6 Stream<Employee> Manipulations

```
75          // display unique employee last names sorted
76          System.out.printf("%nUnique employee last names:%n");
77          list.stream()
78              .map(Employee::getLastName)
79              .distinct()
80              .sorted()
81              .forEach(System.out::println);
82
83          // display only first and last names
84          System.out.printf(
85             "%nEmployee names in order by last name then first name:%n");
86          list.stream()
87              .sorted(lastThenFirst)
88              .map(Employee::getName)
89              .forEach(System.out::println);
90
```

Fig. 17.13 | Mapping `Employee` objects to last names and whole names. (Part 1 of 2.)

9b.6 Stream<Employee> Manipulations

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

9b.6.5 Grouping Employees By Department

- Figure 17.14 uses `Stream` method `collect` to group `Employees` by department.
- `Collectors` `static` method `groupingBy` with one argument receives a `Function` that classifies objects in the stream—the values returned by this function are used as the keys in a `Map`.
  - The corresponding values, by default, are Lists containing the stream elements in a given category.
- `Map` method `forEach` performs an operation on each key–value pair.
  - Receives an object that implements functional interface `BiConsumer`.
  - `BiConsumer`'s `accept` method has two parameters.
  - For `Map`s, the `first` represents the key and the `second` the corresponding value.

9b.6 Stream<Employee> Manipulations

```
91          // group Employees by department
92          System.out.printf("%nEmployees by department:%n");
93          Map<String, List<Employee>> groupedByDepartment =
94             list.stream()
95                .collect(Collectors.groupingBy(Employee::getDepartment));
96          groupedByDepartment.forEach(
97             (department, employeesInDepartment) ->
98             {
99                System.out.println(department);
100               employeesInDepartment.forEach(
101                  employee -> System.out.printf("   %s%n", employee));
102            }
103         );
104
```

Fig. 17.14 | Grouping `Employees` by department. (Part 1 of 2.)

9b.6 Stream<Employee> Manipulations

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

9b.6.6 Counting the Number of Employees in Each Department

- Figure 17.15 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.
- `Collectors` `static` method `groupingBy` with two arguments receives a `Function` that classifies the objects in the stream and another `Collector` (known as the downstream `Collector`).
- `Collectors` `static` method `counting` returns a `Collector` that counts the number of objects in a given classification, rather than collecting them into a `List`.

9b.6 Stream<Employee> Manipulations

```
105         // count number of Employees in each department
106         System.out.printf("%nCount of Employees by department:%n");
107         Map<String, Long> employeeCountByDepartment =
108            list.stream()
109                .collect(Collectors.groupingBy(Employee::getDepartment,
110                   Collectors.counting()));
111         employeeCountByDepartment.forEach(
112            (department, count) -> System.out.printf(
113               "%s has %d employee(s)%n", department, count));
114
```

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

Fig. 17.15 | Counting the number of `Employee`s in each department.

9b.6.7 Summing and Averaging Employee Salaries

- Figure 17.16 demonstrates `Stream` method `mapToDouble`, which maps objects to `double` values and returns a `DoubleStream`.
- `Stream` method `mapToDouble` maps objects to `double` values and returns a `DoubleStream`. The method receives an object that implements the functional interface `ToDoubleFunction` (package `java.util.function`).
  - This interface's `applyAsDouble` method invokes an instance method on an object and returns a `double` value.

9b.6 Stream<Employee> Manipulations

```
115         // sum of Employee salaries with DoubleStream sum method
116         System.out.printf(
117            "%nSum of Employees' salaries (via sum method): %.2f%n",
118            list.stream()
119                .mapToDouble(Employee::getSalary)
120                .sum());
121
122         // calculate sum of Employee salaries with Stream reduce method
123         System.out.printf(
124            "Sum of Employees' salaries (via reduce method): %.2f%n",
125            list.stream()
126                .mapToDouble(Employee::getSalary)
127                .reduce(0, (value1, value2) -> value1 + value2));
128
129         // average of Employee salaries with DoubleStream average method
130         System.out.printf("Average of Employees' salaries: %.2f%n",
131            list.stream()
132                .mapToDouble(Employee::getSalary)
133                .average()
134                .getAsDouble());
135      } // end main
136   } // end class ProcessingEmployees
```

Fig. 17.16 | Summing and averaging `Employee` salaries. (Part 1 of 2.)

9b.6 Stream<Employee> Manipulations

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

9b.7 Creating a `Stream<String>` from a File

- **Figure 17.17 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.**
- **Figure 17.18 shows the program's output.**
- **`Files` method `lines` creates a `Stream<String>` for reading the lines of text from a file.**
- **`Stream` method `flatMap` receives a `Function` that maps an object into a stream—e.g., a line of text into words.**
- **`Pattern` method `splitAsStream` uses a regular expression to tokenize a `String`.**

9b. 7 Creating a Stream<String> from a File

```
1  // Fig. 17.17: 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 {
13    public static void main(String[] args) throws IOException
14    {
15       // Regex that matches one or more consecutive whitespace characters
16       Pattern pattern = Pattern.compile("\\s+");
17
18       // count occurrences of each word in a Stream<String> sorted by word
19       Map<String, Long> wordCounts =
20          Files.lines(Paths.get("Chapter2Paragraph.txt"))
21               .map(line -> line.replaceAll("(?!')\\p{P}", ""))
22               .flatMap(line -> pattern.splitAsStream(line))
23               .collect(Collectors.groupingBy(String::toLowerCase,
24                  TreeMap::new, Collectors.counting()));
```

Fig. 17.17 | Counting word occurrences in a text file. (Part 1 of 2.)

9b. 7 Creating a Stream<String> from a File

```
25
26          // display the words grouped by starting letter
27          wordCounts.entrySet()
28             .stream()
29             .collect(
30                Collectors.groupingBy(entry -> entry.getKey().charAt(0),
31                   TreeMap::new, Collectors.toList()))
32             .forEach((letter, wordList) ->
33                {
34                   System.out.printf("%n%C%n", letter);
35                   wordList.stream().forEach(word -> System.out.printf(
36                      "%13s: %d%n", word.getKey(), word.getValue()));
37                });
38       }
39    } // end class StreamOfLines
```

Fig. 17.17 | Counting word occurrences in a text file. (Part 2 of 2.)

9b. 7 Creating a Stream<String> from a File

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

Fig. 17.18 | Output for the program of Fig. 17.17 arranged in three columns.

9b. 7 Creating a Stream<String> from a File

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

Fig. 17.18 | Output for the program of Fig. 17.17 arranged in three columns.

9b.7 Creating a Stream<String> from a File

- Collectors method groupingBy with three arguments receives a classifier, a Map factory and a downstream Collector.
  - The classifier is a Function that returns objects which are used as keys in the resulting Map.
  - The Map factory is an object that implements interface Supplier and returns a new Map collection.
  - The downstream Collector determines how to collect each group's elements.
- Map method entrySet returns a Set of Map.Entry objects containing the Map's key–value pairs.
- Set method stream returns a stream for processing the Set's elements.

9b.8 Generating Streams of Random Values

- **In Fig. 6.7, we demonstrated rolling a six-sided die 6,000,000 times and summarizing the frequencies of each face 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.**

9b.8 Generating Streams of Random Values

- **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 6,000,000 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.**

9b.8 Generating Streams of Random Values

- **Figure 17.19 reimplements those programs with a *single statement* that does it all, using lambdas, streams, internal iteration and no mutable variables to roll the die 6,000,000 times, calculate the frequencies and display the results.**

9b.8 Generating Streams of Random Values

```
1  // Fig. 17.19: RandomIntStream.java
2  // Rolling a die 6,000,000 times with streams
3  import java.security.SecureRandom;
4  import java.util.Map;
5  import java.util.function.Function;
6  import java.util.stream.IntStream;
7  import java.util.stream.Collectors;
8
9  public class RandomIntStream
10 {
11    public static void main(String[] args)
12    {
13       SecureRandom random = new SecureRandom();
14
15       // roll a die 6,000,000 times and summarize the results
16       System.out.printf("%-6s%s%n", "Face", "Frequency");
17       random.ints(6_000_000, 1, 7)
18             .boxed()
19             .collect(Collectors.groupingBy(Function.identity(),
20                Collectors.counting()))
21             .forEach((face, frequency) ->
22                System.out.printf("%-6d%d%n", face, frequency));
23    }
24 } // end class RandomIntStream
```

Fig. 17.19 | Rolling a die 6,000,000 times with streams. (Part 1 of 2.)

9b.8 Generating Streams of Random Values

```
Face   Frequency
1      999339
2      999937
3      1000302
4      999323
5      1000183
6      1000916
```

Fig. 17.19 | Rolling a die 6,000,000 times with streams. (Part 2 of 2.)

9b.8 Generating Streams of Random Values

Class `SecureRandom`'s methods `ints`, `longs` and `doubles` (inherited from class `Random`) return `IntStream`, `LongStream` and `DoubleStream`, respectively, for streams of random numbers.

Method `ints` with no arguments creates an `IntStream` for an infinite stream of random `int` values.

An infinite- stream is a stream with an unknown number of elements—you use a short-circuiting terminal operation to complete processing on an infinite stream.

9b.8 Generating Streams of Random Values

Method `ints` with a `long` argument creates an `IntStream` with the specified number of random `int` values.

Method `ints` with two `int` arguments creates an `IntStream` for an infinite stream of random `int` values in the range starting with the first argument and up to, but not including, the second.

Method `ints` with a `long` and two `int` arguments 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.

9b.8 Generating Streams of Random Values

- To convert an `IntStream` to a `Stream<Integer>` call `IntStream` method `boxed`.
- `Function` `static` method `identity` creates a `Function` that simply returns its argument.

Questions