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. We walk through the rest of the code in Sections 17.12.1–17.12.7

```
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 last
11          double salary, String department) {
12          this.firstName = firstName;
13          this.lastName = lastName;
14          this.salary = salary;
15          this.department = department;
16      }
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
33      // get department
34      public String getDepartment() {
35          return department;
36      }
37
38      // return Employee's first and last name comb
39      public String getName() {
40          return String.format("%s %s", getFirstName
41      }
42
```

```
43      // return a String containing the Employee's
                44      @Override
           45      public String toString() {
46             return String.format("%-8s %-8s %8.2f  %s"
47                getFirstName(), getLastName(), getSalar
                      48      }
                        49   }
```

Fig. 17.13

`Employee` class for use in Figs. 17.14–17.21.

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). 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` instance method, which implicitly calls class `Employee`'s `toString` method to get the `String` representation. 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
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, "
18          new Employee("Matthew", "Indigo", 3587.
```

```
19            new Employee("James", "Indigo", 4700.77
20            new Employee("Luke", "Indigo", 6200, "I
21            new Employee("Jason", "Blue", 3200, "Sa
22            new Employee("Wendy", "Brown", 4236.4,
23
24          // get List view of the Employees
25          List<Employee> list = Arrays.asList(employ
26
27          // display all Employees
28          System.out.println("Complete Employee list
29          list.stream().forEach(System.out::println)
30
```

```
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.

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

9

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, "Marketin
   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 `Employee`s.

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

```
Employees earning $4000-$6000 per month sorted by sal
```

```
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 `Employee`s with salaries in the range $4000–$6000.

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.

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. 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.

We knew from this example's `Employee`s 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`.

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* `NoSuchElementException`*s.*

Method `findFirst` is one of several search-related terminal operations. Figure 17.16 shows several similar `Stream` methods.

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

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`

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. 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 na
52          Function<Employee, String> byFirstName = E
53          Function<Employee, String> byLastName = Em
                        54
```

```
55          // Comparator for comparing Employees by f
 56          Comparator<Employee> lastThenFirst =
57             Comparator.comparing(byLastName).thenCo
                        58
59          // sort employees by last name, then first
           60          System.out.printf(
61             "%nEmployees in ascending order by last
              62          list.stream()
       63                .sorted(lastThenFirst)
  64               .forEach(System.out::println);
                        65
66          // sort employees in descending order by l
           67          System.out.printf(
68             "%nEmployees in descending order by las
              69          list.stream()
 70               .sorted(lastThenFirst.reversed())
  71               .forEach(System.out::println);
                        72
```

```
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 `Employee`s by last name then first name.

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.

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 sort
74         System.out.printf("%nUnique employee last
  75         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
   list.stream()
   .sorted(lastThenFirst)
   .map(Employee::getName)
   .forEach(System.out::println);
```

```
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.

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 departme
91          Map<String, List<Employee>> groupedByDepar
            92             list.stream()
93                 .collect(Collectors.groupingBy(Empl
     94           groupedByDepartment.forEach(
95              (department, employeesInDepartment) ->
96                 System.out.printf("%n%s%n", departme
 97                  employeesInDepartment.forEach(
98                    employee -> System.out.printf("
                   99             }
                    100         );
                         101
```

```
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 `Employee`s by department.

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. 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 depa
103         System.out.printf("%nCount of Employees b
104         Map<String, Long> employeeCountByDepartme
105            list.stream()
106                .collect(Collectors.groupingBy(Emp
107                   Collectors.counting()));
108         employeeCountByDepartment.forEach(
109            (department, count) -> System.out.prin
110               "%s has %d employee(s)%n", departme
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.

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*.

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.

```
112         // sum of Employee salaries with DoubleSt
113         System.out.printf(
114            "%nSum of Employees' salaries (via sum
115            list.stream()
116               .mapToDouble(Employee::getSalary)
117               .sum());
118
119         // calculate sum of Employee salaries wit
120         System.out.printf(
121            "Sum of Employees' salaries (via reduc
122            list.stream()
123               .mapToDouble(Employee::getSalary)
124               .reduce(0, (value1, value2) -> val
125
126         // average of Employee salaries with Doub
127         System.out.printf("Average of Employees'
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
```

```
Average of Employees' salaries: 4932.10
```

Fig. 17.21

Summing and averaging `Employee` salaries.

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.

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.