Chapter 17
Java SE 8 Lambdas and Streams

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.
  - Create streams representing ranges of int values and random int values.
  - Implement functional interfaces with lambdas.
  - Perform on IntStreams intermediate operations filter, map, mapToObj and sorted, and terminal operations forEach, count, min, max, sum, average and reduce.
  - Perform on Streams intermediate operations distinct, filter, map, mapToDouble and sorted, and terminal operations collect, forEach, findFirst and reduce.
  - Process infinite streams.
  - Implement event handlers with lambdas.

17.1 Introduction

- Prior to Java SE 8, Java supported three programming paradigms—procedural programming, object-oriented programming and generic programming.
- Java SE 8 added lambdas and streams—key technologies of functional programming.

17.1 Introduction(cont.)

- We can use it to write programs faster, simpler, more concisely（简明地）,with fewer bugs, be easier to parallelize than programs written with previous techniques.

17.2 Streams and Reduction

- In counter-controlled iteration, we typically accomplish what we want using a for loop.
- In this section, we'll use a better way to accomplish the same tasks.

17.2 Streams and Reduction(cont.)

External literation：

specify all the iteration details

```
int total = 0;
for (int number = 1; number <= 10; number++) {
total += number;
}
```

Error Prone

```
IntStream.rangeClosed(1, 10)
         .sum();
```

A **stream** is a sequence of elements on which you perform tasks, and the **stream pipeline** moves the stream's elements through a sequence of tasks.

```
IntStream.rangeClosed(1, 10)
```

create an IntStream containing the ordered sequence of int elements1,2,3,4,5,6,7,8,9 and 10.

```
IntStream.range(1, 10)
```

produces an IntStream containing the ordered sequence of int elements 1, 2, 3, 4, 5, 6, 7, 8 and 9, but not 10.

.sum()

returns the sum of all the ints in the stream.

The processing step performed by method sum is known as a **reduction**—it reduces the stream of values to a single value.

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.

Return a stream with a collection

- Interface Collection<E>

default Stream<E>	**parallelStream()** Returns a possibly parallel Stream with this collection as its source.
default Stream<E>	**stream()** Returns a sequential Stream with this collection as its source.

Return a stream with an array Class Arrays

static **DoubleStream**	**stream**(double[] array) Returns a sequential **DoubleStream** with the specified array as its source.
static **DoubleStream**	**stream**(double[] array, int startInclusive, int endExclusive) Returns a sequential **DoubleStream** with the specified range of the specified array as its source.
static **IntStream**	**stream**(int[] array) Returns a sequential **IntStream** with the specified array as its source.
static **IntStream**	**stream**(int[] array, int startInclusive, int endExclusive) Returns a sequential **IntStream** with the specified range of the specified array as its source.
static **LongStream**	**stream**(long[] array) Returns a sequential **LongStream** with the specified array as its source.
static **LongStream**	**stream**(long[] array, int startInclusive, int endExclusive) Returns a sequential **LongStream** with the specified range of the specified array as its source.
static <T> **Stream**<T>	**stream**(T[] array) Returns a sequential **Stream** with the specified array as its source.
static <T> **Stream**<T>	**stream**(T[] array, int startInclusive, int endExclusive) Returns a sequential **Stream** with the specified range of the specified array as its source.

17.2 Streams and Reduction(cont.)

- **Intermediate operations（中间操作）**
  - **use lazy evaluation.** 把结果给后面的流，进行下一步的传递
  - results in a new stream object
  - map, filter, distinct, limit, sorted
- **Terminal operation（终止操作、终端操作）**
  - use eager evaluation
  - initiates a stream pipeline's processing and produces a result.
  - count, min, max, average, summaryStatistics，reduce

17.3 Mapping and Lambdas

```
int total = 0;
for (int number = 2; number <= 20; number
+= 2) {
  total += number;}
```

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

17.3 Mapping and Lambdas(cont.)

- **lambda expression**
  - represents an *anonymous method—a shorthand notation for implementing a functional interface,* similar to an anonymous inner class. 匿名内部类
  - The type of a lambda expression is the type of the functional interface that the lambda expression implements.
  - Lambda expressions can be used anywhere functional interfaces are expected.

17.3 Mapping and Lambdas(cont.)

- As of Java SE 8, any interface containing only one abstract method is known as a **functional interface**（函数式接口）.

Lambda Syntax

- A lambda consists of a parameter list followed by the arrow token (->) and a body, as in:
  - *(parameterList) -> {statements}*
- The following lambda receives two ints and returns their sum:

对应方法调用

  - (int x, int y) -> {return x + y;}

There are several variations of this syntax

- the parameter types usually may be omitted, as in:
  - (x, y) -> {return x + y;}

只有一个返回值，省略return

- When the body contains only one expression, the return keyword and curly braces may be omitted, as in:
  - (x, y) -> x + y

- When the parameter list contains only one parameter, the parentheses may be omitted, as in:
  - value -> System.out.printf("%d ", value)
  - x -> x * 2

只有一个参数可以省略括号，返回值void

- To define a lambda with an empty parameter list：
  - () -> System.out.println("Welcome to lambdas!")

17.4 Filtering

```
int total = 0;
for (int x = 1; x <= 10; x++) {
  if (x % 2 == 0) { // if x is even
      total += x * 3;
  }
}
```

```
IntStream.rangeClosed(1, 10)
.filter(x -> x % 2 == 0)
.map(x -> x * 3)
.sum();
```

17.5 How Elements Move Through Stream Pipelines

```
IntStream.rangeClosed(1, 10)
.filter(x -> x % 2 == 0)
.map(x -> x * 3)
.sum();
```

the stream pipeline operates as follows:

For each element
    If the element is an even integer
        Multiply the element by 3 and add the result to the total

17.5 How Elements Move Through Stream Pipelines(cont.)

- When you initiate a stream pipeline with a terminal operation, the intermediate operations' processing steps are applied for a given stream element before they are applied to the next stream element.

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

```
filter: 1

filter: 2
map: 2

filter: 3

filter: 4
map: 4

filter: 5

filter: 6
map: 6

filter: 7

filter: 8
map: 8

filter: 9

filter: 10
map: 10
```

17.6 Method References

For a lambda that simply calls another method, you can replace the lambda with that method's name—known as a method reference.

```
randomNumbers.ints(10, 1, 7)
             .forEach(System.out::println);

    // 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);
```

调用静态方法

把其他类型转换为字符串

把串用空格拼到一块

```
Random numbers on separate lines:
4
5
2
2
2
2
5
3
1
1

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

17.6 Method Referennces (cont.)

- IntStream method mapToObj enables you to map from ints to a stream of reference-type elements.

定义泛型的子类

```
<U> Stream<U>          mapToObj(IntFunction<? extends U> mapper)
```

Returns an object-valued Stream consisting of the results of applying the given function to the elements of this stream.

- The Stream terminal operation collect uses a collector to gather the stream's elements into a single object—often a collection.
- static Collectors method joining creates a concatenated String representation of the stream's elements.

17.7 IntStream Operation

```
 9  public class IntStreamOperations {
10     public static void main(String[] args) {
11        int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
12
13        // display original values
14        System.out.print("Original values: ");
15        System.out.println(
16           IntStream.of(values)
17                    .mapToObj(String::valueOf)
18                    .collect(Collectors.joining(" ")));
19
20        // count, min, max, sum and average of the values
21        System.out.printf("%nCount: %d%n", IntStream.of(values).count());
22        System.out.printf("Min: %d%n",
23           IntStream.of(values).min().getAsInt());
24        System.out.printf("Max: %d%n",
25           IntStream.of(values).max().getAsInt());
26        System.out.printf("Sum: %d%n", IntStream.of(values).sum());
27        System.out.printf("Average: %.2f%n",
28           IntStream.of(values).average().getAsDouble());
```

```
30          // sum of values with reduce method
31          System.out.printf("%nSum via reduce method: %d%n",
32             IntStream.of(values)
33                      .reduce(0, (x, y) -> x + y));
34
35          // product of values with reduce method
36          System.out.printf("Product via reduce method: %d%n",
37             IntStream.of(values)
38                      .reduce((x, y) -> x * y).getAsInt());
39
40          // sum of squares of values with map and sum methods
41          System.out.printf("Sum of squares via map and sum: %d%n%n",
42             IntStream.of(values)
43                      .map(x -> x * x)
44                      .sum());
45
46          // displaying the elements in sorted order
47          System.out.printf("Values displayed in sorted order: %s%n",
48             IntStream.of(values)
49                      .sorted()
50                      .mapToObj(String::valueOf)
51                      .collect(Collectors.joining(" ")));
52       }
53    }
```

初始值是0

```
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
```

```
IntStream.of(values)
```

creates an IntStream from the array values

```
System.out.println(IntStream.of(values).summaryStatistics());
```

Produces:

IntSummaryStatistics{count=10, sum=55, min=1, average=5.500000, max=10}

```
IntStream.of(values)
        .reduce(0, (x, y) -> x + y)
```

The first argument to `reduce(0)` is the operation's identity value.

The second argument is a method that receives two int values (representing the left and right operands of a binary operator), performs a calculation with the values and returns the result.

17.8 Functional Interfaces

接口里面还可以有default,private,static

- functional interface

只包含一个抽象方法

  - an interface that contains exactly one abstract method (and may also contain default and static methods)

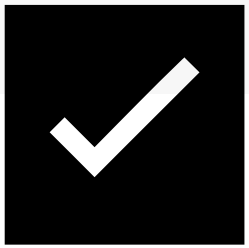
一个表达式可以通过不改变程序的行为而被其结果值替换

- **referential transparency（RT, 引用透明性）**
  - An expression is *referentially transparent* if it can be replaced by its resulting value without changing the behavior of the program. This must be true regardless of where the expression is used in the program.

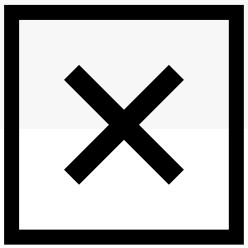
referential transparency?

改变了外部变量，引用不透明

```
List<String> collectNames(String[] names) {

  List<String> result = new ArrayList<String>();

  for (String name:names) {

    result.add(name);

  }
}
```



```
List<String> all  = new ArrayList<String>();
List<String> collectNames(String[] names) {

  for (String name:names) {

    all.add(name);

  }

  return all;

}
```



17.8 Functional Interfaces(cont.)

- **Pure functions**
  - depend only on their parameters
  - have no side-effects and do not maintain any state.
- In Java, pure functions are methods that implement functional interfaces—typically defined as lambdas

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.

The six basic generic functional interfaces in package java.util.function.

Interface	Description
`BinaryOperator<T>`	Contains method `apply` that takes two `T` arguments, performs an operation on them (such as a calculation) and returns a value of type `T`. You'll see several examples of `BinaryOperators` starting in Section 17.3.
`Consumer<T>`	Contains method `accept` that takes a `T` argument and returns `void`. Performs a task with it's `T` argument, such as outputting the object, invoking a method of the object, etc. You'll see several examples of `Consumers` starting in Section 17.3.
`Function<T,R>`	Contains method `apply` that takes a `T` argument and returns a value of type `R`. Calls a method on the `T` argument and returns that method's result. You'll see several examples of `Functions` starting in Section 17.5.
`Predicate<T>`	Contains method `test` that takes a `T` argument and returns a `boolean`. Tests whether the `T` argument satisfies a condition. You'll see several examples of `Predicates` starting in Section 17.3.
`Supplier<T>`	Contains method `get` that takes no arguments and produces a value of type `T`. 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 17.7.
`UnaryOperator<T>`	Contains method `get` that takes no arguments and returns a value of type `T`. You'll see several examples of `UnaryOperators` starting in Section 17.3.

17.9 Lambdas: A Deeper Look

- Lambda expressions can be used anywhere functional interfaces are expected.

```
IntStream.rangeClosed(1, 10)
        .filter(x ->  x % 2 == 0)
        .map(x ->  x * 3)
        .sum()
```

```
IntStream filter(IntPredicate predicate)
```

```
boolean test(int value)
```

x % 2 == 0

x

```
IntStream.rangeClosed(1, 10)
        .filter(x ->  x % 2 == 0)
        .map(x ->  x * 3)
        .sum()
```

```
IntStream map(IntUnaryOperator mapper)
```

```
int applyAsInt(int operand)
```

x * 3

x

17.9 Lambdas: A Deeper Look(cont.)

- Scope and Lambdas
  - lambdas do not have their own scope
- Capturing Lambdas and final Local Variables
  - A lambda that refers to a local variable from the enclosing method (known as the lambda's lexical scope语法范围) is a capturing lambda.
  - Any local variable that a lambda references in its [redacted]ust be final.

17.10 Stream<Integer> Manipulations

- IntStream is simply an int-optimized Stream that provides methods for common int operations.

```java
// Fig. 17.11: ArraysAndStreams.java
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));

      // 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);
```

返回list类型

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

```
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]
```

17.11 Stream<String> Manipulations

```
3  // Fig. 17.12: ArraysAndStreams2.java
5  import java.util.Arrays;
6  import java.util.Comparator;
7  import java.util.stream.Collectors;
8
9  public class ArraysAndStreams2 {
10    public static void main(String[] args) {
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
23       // strings less than "n" (case insensitive) sorted ascending
24       System.out.printf("strings less than n 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 less than n 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  }
38
```

```
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]
```

17.12 Stream<Employee> Manipulations

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

```
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       }
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       }
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       }
```

```
68      // return Employee's first and last name combined
69      public String getName()
70      {
71         return String.format("%s %s", getFirstName(), getLastName());
72      }
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
```

```
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")};
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
```

Filtering Employees with Salaries in a Specified Range

```
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
```

基于什么进行比较

```
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
```

Sorting Employees By Multiple Fields

```
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
```

Mapping Employees to Unique Last Name Strings

```
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
```

Grouping Employees By Department

```
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
```

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.

```
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
```

Counting the Number of Employees in Each Department

```
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
```

根据department做分组，counting得到每组的个数

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

- 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).**

Summing and Averaging Employee Salaries

```
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
```

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

17.12 Stream<Employee> Manipulations(cont.)

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

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

17.12 Stream<Employee> Manipulations(cont.)

- Predicate<Employee>

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method	Description
boolean	**evaluate** (**Object** value, int column)	This method is called by a FilteredRowSet object to check whether the value lies between the filtering criterion (or criteria if multiple constraints exist) set using the setFilter() method.
boolean	**evaluate** (**Object** value, **String** columnName)	This method is called by the FilteredRowSet object to check whether the value lies between the filtering criteria set using the setFilter method.
boolean	**evaluate** (**RowSet** rs)	This method is typically called a FilteredRowSet object internal methods (not public) that control the RowSet object's cursor moving from row to the next.

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.

17.12 Stream<Employee> Manipulations(cont.)

- Aside: Composing Lambda Expressions
  - Many functional interfaces in the package java.util.function package provide default methods that enable you to compose functionality.
  - For example, the interface IntPredicat contains three default methods: and,negate,or.

```
IntPredicate even = value -> value % 2 == 0;

IntPredicate greaterThan5 = value -> value > 5;

even.and(greaterThan5);
```

17.13 Creating a Stream<String> from a File

```
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()));
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
```

```
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

I
       inputs: 1
     instruct: 1
   introduces: 1

J
         java: 1
          jdk: 1

L
         last: 1
        later: 1
        learn: 1

M
         make: 1
     messages: 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
```

- Line 20 uses Files method **lines to create a Stream<String> for reading the** lines of text from a file
- Line 22 uses Stream method **flatMap to break each line of text into its separate** words.
- The lambda in line 22 passes the String representing a line of text to Pattern method **split- AsStream (new in Java SE 8), which uses the regular expression specified in the** Pattern (line 16) to tokenize the String into its individual words.

- Re hihy.iteye.com/blog/1777065

- Collectors method **groupingBy** that receives three arguments— a classifier, a Map factory and a downstream Collector.
- The classifier is a Function that returns objects for use as keys in the resulting Map—the method reference String::toLowerCase converts each word in the Stream<String> to lowercase. The Map factory is an object that implements interface Supplier and returns a

- So, line 27 calls Map method entrySet on wordCounts to get a Set of **Map.Entry objects that each** contain one key–value pair from wordCounts. This produces an object of type Set<Map.Entry<String, Long>>.
- Line 28 calls Set method stream to get a Stream<Map.Entry<String, Long>>.

- Lines 29–31 call Stream method collect with three arguments—a classifier, a Map factory and a downstream Collector. The classifier Function in this case gets the key from the Map.Entry then uses String method charAt to get the key's first character—this becomes a Character key in the resulting Map. Once again, we use the constructor reference TreeMap::new as the Map factory to create a TreeMap that maintains its keys in sorted order. The downstream Collector (Collectors.toList()) places the Map.Entry objects into a List c

17.14 Streams of Random Values

```
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
```

```
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.)

`ints()`—creates an `IntStream` for an *infinite stream* of random `ints`. An infinite stream has an *unknown* number of elements—you use a short-circuiting terminal operation to complete processing on an infinite stream. We'll use an infinite stream in Chapter 23 to find prime numbers with the Sieve of Eratosthenes.

`ints(long)`—creates an `IntStream` with the specified number of random `ints`.

`ints(int, int)`—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 argument.

`ints(long, int, int)`—creates an `IntStream` with the specified number of random `int` values in the range starting with the first argument and up to, but not including, the second argument.

Converting an IntStream to a Stream<Integer>

- We do this by calling IntStream method **boxed.**

- Static method **identity from interface Function creates a Function that simply returns** its argument.

17.15 Infinite Streams

- Lazy evaluation makes it possible to work with infinite streams that represent an unknown, potentially infinite, number of elements.

IntStream method iterate generates an ordered sequence of values starting with the seed value (1) in its first argument.

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

We did not specify how many elements to produce, so this is the equivalent of an infinite loop.

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

limits the total number of elements produced to 10, so it displays the numbers from 1 through 10.

IntStream method generate create unordered infinite streams.

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

This is equivalent to using SecureRandom's no-argument ints method

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

17.16 Lambda Event Handlers

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

tipPercentageLabel.setText(percent.format(tipPercenta
ge));
      }
   }
);
```

can be implemented more concisely with a lambda as:

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

tipPercentageLabel.setText(percent.format(tipPerce
ntage));
   });
```

17.17 Additional Notes on Java SE 8 Interfaces

- Java SE 8 Interfaces Allow Inheritance of Method Implementations
  - When a class implements an interface with default methods and does not override them, the class inherits the default methods' implementations.
    - If a class implements two or more unrelated interfaces that provide a default method with the same signature, the implementing class must override that method; otherwise, a compilation error occurs.

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

- An interface's designer can now evolve an interface by adding new default and static methods without breaking existing code that implements the interface.

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

- Java SE 8: @FunctionalInterface Annotation
  - you can declare that an interface is a functional interface by preceding it with the @FunctionalInterface annotation.
  - The compiler will then ensure that the interface contains only one abstract method; otherwise, it will generate a compilation error.