Chapter 9

Streams

Business Training

1

1Z0-809 Chapter 9

Content

- Collection Streams and Filters
- How to create a Stream ?
- The Stream Pipeline
- Search Data from a Stream
- The Optional class
- Stream Data Methods and Calculation Methods
- Sort a Collection Using Stream API
- Save Results to a Collection
- Using flatMap Method in Stream
- Map-Filter-Reduce with average() and Optionals

2

Collection Streams and Filters

3

What Is a Stream?

- **A stream is a sequence of elements (you can think of these elements as data) that can be processed with operations.**
  - That's plenty vague as a definition, so to get a better handle on what a stream is, let's make one:

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Arrays.stream(myNums);
```

  - The resulting stream, myStream, is a stream of Integers. What does the stream look like?
    - System.out.println(myStream);
  - This results in the output: `java.util.stream.ReferencePipeline$Head@14ae5a5`
    - Initially, you might think that a stream is a bit like a data structure, like a List or an array.
    - But streams are not a data structure to organize data like a List is or an array is; rather, they are a way to process data that you can think of as flowing through the stream, much like water flows through a stream in the real world.

4

Why Streams ?

- **The main reason to use streams is when you start doing multiple intermediate operations.**
  - So far, we've been performing only one intermediate operation: a filter, using a variety of different Predicates to filter the data we get from the stream before we count it.
  - However, when we use multiple intermediate operations, we start seeing the benefits of streams.
- **You're probably saying to yourself, what's the big deal? We could do the same thing using iteration, right?**
  - The big deal is that streams use something called a "pipeline," which can, in some circumstances, **dramatically improve the efficiency of data processing.**

5

How to create a Stream ?

6

How to Create a Stream ?

- **Create a Stream from a Collection**

```
List<Double> tempsInPhoenix = Arrays.asList(123.6, 118.0, 113.0, 112.5,
     115.8, 117.0, 110.2, 110.1, 106.0, 106.4);
System.out.println("Number of days over 110 in 10 day period: " +
tempsInPhoenix
    .stream()                  // stream the List of Doubles
    .filter(t -> t > 110.0)    // filter the stream
    .count());                 // count the Doubles that pass the filter test
```

- **Build a Stream with Stream.of()**

```
Integer[] myNums = { 1, 2, 3 };
Stream<Integer> myStream = Stream.of(myNums);
```

- **Create a Stream from an Array**

```
String[] dogs = { "Boi", "Zooey", "Charis" };        // make an array
Stream<String> dogStream = Arrays.stream(dogs);      // stream it
System.out.println("Number of dogs in array: " + dogStream.count()); // count it
```

- **Create a Stream from a File**

```
Stream<String> stream = Files.lines(Paths.get(filename)));
```

- **Primitive Value Streams**

```
DoubleStream s3 = DoubleStream.of(406.13, 406.42, 407.18, 409.01);
```


7

Summary of Methods to Create Streams

Interface/Class	Creates a...	With method...
`Collection`	`Stream<E>`	`stream()`
`Arrays`	`Stream<T>`	`stream(T[] array)` `stream(T[] array, int startInclusive, int endExclusive)`
`Arrays`	`IntStream`, `DoubleStream`, `LongStream`	`stream(int[] array)`, `stream(double[] array)`, `stream(long[] array)` (and versions with start/end like `Stream`)
`Files`	`Stream<String>`	`lines(Path path)`, `lines(Path path, Charset cs)`
`Stream`	`Stream<T>`	`of(T... values)`, `of(T t)`
`DoubleStream`	`DoubleStream`	`of(double... values)`, `of(double t)`
`IntStream`	`IntStream`	`of(int... values)`, `of(int t)`
`LongStream`	`LongStream`	`of(long... values)`, `of(long t)`

8

The Stream Pipeline

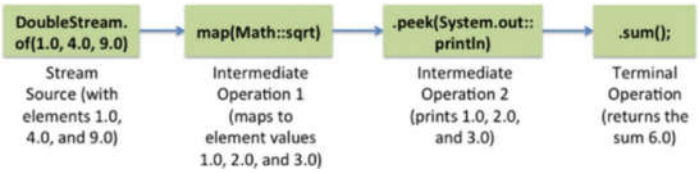
9

An example of Stream pipeline

- **A stream pipeline with source, intermediate operations and terminal operation**
  - Example : You can use map() and peek() methods in primitive versions of Stream<T>; then following code snippet uses a DoubleStream:

```
DoubleStream.of(1.0, 4.0, 9.0)
        .map(Math::sqrt)
        .peek(System.out::println)
        .sum();
```

  - This code prints 1.0, 2.0, and 3.0 in separate lines on the console. Figure 6-1 visually shows the source, intermediate operations and the terminal operations in this stream pipeline.



  - the peek() method is meant primarily for debugging purposes. It helps us understand how the elements are transformed in the pipeline. Do not use it in production code.

10

Search Data from a Stream

11

Search Data from a Stream

- **Methods ending with the word "Match" and methods starting with the word "find" in the Stream interface are useful for searching data from the stream**
- **Important Match and Find Methods in the Stream Interface**

Method Name	Short Description
`boolean anyMatch(Predicate<? super T> check)`	Returns true if there is any elements in the stream that matches the given predicate. Returns false if the stream is empty or if there are no matching elements.
`boolean allMatch(Predicate<? super T> check)`	Returns true only if *all* elements in the stream matches the given predicate. Returns true if the stream is empty without evaluating the predicate!
`boolean noneMatch(Predicate<? super T> check)`	Returns true only if *none* of the elements in the stream matches the given predicate. Returns true if the stream is empty without evaluating the predicate!
`Optional<T> findFirst()`	Returns the first element from the stream; if there is no element present in the stream, it returns an empty `Optional<T>` object.
`Optional<T> findAny()`	Returns one of the elements from the stream; if there is no element present in the stream, it returns an empty `Optional<T>` object.

12

Search Data from a Stream

- **Here is a simple program that illustrates how to use anyMatch(), allMatch(), and noneMatch() methods**

```
// Average temperatures in Concordia, Antarctica in a week in October 2015
boolean anyMatch
        = IntStream.of(-56, -57, -55, -52, -48, -51, -49).anyMatch(temp -> temp > 0);
System.out.println("anyMatch(temp -> temp > 0): " + anyMatch);

boolean allMatch
        = IntStream.of(-56, -57, -55, -52, -48, -51, -49).allMatch(temp -> temp > 0);
System.out.println("allMatch(temp -> temp > 0): " + allMatch);

boolean noneMatch
        = IntStream.of(-56, -57, -55, -52, -48, -51, -49).noneMatch(temp -> temp > 0);
System.out.println("noneMatch(temp -> temp > 0): " + noneMatch);
```

  - This program prints:

    anyMatch(temp -> temp > 0): false

    allMatch(temp -> temp > 0): false

    noneMatch(temp -> temp > 0): true

    - Unlike the anymatch() method that returns false when the stream is empty, the allmatch() and nonematch() methods return true if the stream is empty!

13

The Optional class

14

Stream Methods That Return Optionals

Interface	Method	Returns...
Stream	findAny()	Optional<T>
Stream	findFirst()	Optional<T>
Stream	max(Comparator<? super T> comparator)	Optional<T>
Stream	min(Comparator<? super T> comparator)	Optional<T>
Stream	reduce(BinaryOperator<T> accumulator)	Optional<T>
DoubleStream	average()	OptionalDouble
DoubleStream	findAny(), findFirst()	OptionalDouble
DoubleStream	max(), min()	OptionalDouble
DoubleStream	reduce(DoubleBinaryOperator op)	OptionalDouble
IntStream, LongStream	Similar methods to DoubleStream	OptionalInt, OptionalLong

The Optional class

- **The class java.util.Optional is a holder for value that can be null. There are numerous methods in classes in java.util.stream package that return Optional values. Let us see an example now.**

```
public static void selectHighestTemperature(Stream<Double> temperatures) {
    System.out.println(temperatures.max(Double::compareTo));
}
```

  - Here is a call to this method:

```
selectHighestTemperature(Stream.of(24.5, 23.6, 27.9, 21.1, 23.5, 25.5, 28.3));
```

    - This code prints: Optional[28.3]
  - The max() method in Stream takes a Comparator as an argument and returns an Optional<T>:

```
Optional<T> max(Comparator<? super T> comparator);
```

  - Why Optional<T> instead of return type T?
    - It is because max() method may fail to find the maximum value – think about an empty stream, for example: `selectHighestTemperature(Stream.of());`
    - Now, this code prints: Optional.empty
  - To get the value from Optional, you can use isPresent() and get() methods, as in:

```
public static void selectHighestTemperature(Stream<Double> temperatures) {
    Optional<Double> max = temperatures.max(Double::compareTo);
    if(max.isPresent()) {
        System.out.println(max.get());
    }
}
```

Creating Optional Objects

- **There are many ways to create Optional objects. One way to create Optional objects is to use factory methods in Optional class, as in:**

```
Optional<String> empty = Optional.empty();
```

  - You can also use of() in Optional class:

```
Optional<String> nonEmptyOptional = Optional.of("abracadabra");
```

  - However, you cannot pass null to Optional.of() method, as in:

```
Optional<String> nullStr = Optional.of(null);
System.out.println(nullStr);
// crashes with a NullPointerException
```

- **This will result in throwing a NullPointerException. If you want to create an Optional object that has null value, then you can instead use ofNullable() method:**

```
Optional<String> nullableStr = Optional.ofNullable(null);
System.out.println(nullableStr);
// prints: Optional.empty
```


17

Stream Data Methods and Calculation Methods

18

Important Data and Calculation Methods in IntStream Interface

- **The Stream<T> interface has data and calculation methods count(), min() and max().**
  - Example:

```
String[] string = "you never know what you have until you clean your room".split(" ");
System.out.println(Arrays.stream(string).min(String::compareTo).get());
```

  - This program prints: clean

- **Important Data and Calculation Methods in IntStream Interface**

Method	Short Description
`int sum()`	Returns the sum of elements in the stream; 0 in case the stream is empty.
`long count()`	Returns the number of elements in the stream; 0 if the stream is empty.
`OptionalDouble average()`	Returns the average value of the elements in the stream; an empty `OptionalDouble` value in case the stream is empty.
`OptionalInt min()`	Returns the minimum integer value in the stream; an empty `OptionalInt` value in case the stream is empty.
`OptionalInt max()`	Returns the maximum integer value in the stream; an empty `OptionalInt` value in case the stream is empty.
`IntSummaryStatistics summaryStatistics()`	Returns an `IntSummaryStatistics` object that has `sum`, `count`, `average`, `min`, and `max` values.

19

Sort a Collection Using Stream API

20

Sort a Collection Using Stream API

- **We discussed how to sort a collection using Comparator and Comparable interfaces.**
  - Streams simplify the task of sorting a collection.
- **Here is a program that sorts strings with lexicographical comparison**

```
List words =
        Arrays.asList("follow your heart but take your brain with you".split(" "));
words.stream().distinct().sorted().forEach(System.out::println);
```

  - This program prints: **brain but follow heart take with you your**
  - Here is the modified version of the earlier program that sorts the elements based on the length of the strings.

```
List words =
        Arrays.asList("follow your heart but take your brain with you".split(" "));
Comparator<String> lengthCompare = (str1, str2) -> str1.length() - str2.length();
words.stream().distinct().sorted(lengthCompare).forEach(System.out::println);
```

  - This program prints: **but you take with your brain heart follow**

21

Save Results to a Collection

22

Save Results to a Collection

- **The Collectors class has methods that support the task of collecting elements to a collection.**
  - You can use methods such as **toList(), toSet(), toMap(),** and toCollection() to create a collection from a stream.

```
String frenchCounting = "un:deux:trois:quatre";
List gmailList = Pattern.compile(":")
        .splitAsStream(frenchCounting)
        .collect(Collectors.toList());
gmailList.forEach(System.out::println);
```

    - The collect() method in Stream takes a Collector as an argument:

    <R, A> R collect(Collector<? super T, A, R> collector);

  - Just like Lists and Sets, you can also create Maps from a stream.

```
Map<String, Integer> nameLength = Stream.of("Arnold", "Alois", "Schwarzenegger")
        .collect(Collectors.toMap(name -> name, name -> name.length()));
nameLength.forEach((name, len) -> System.out.printf("%s - %d \n", name, len));
```

    - This program prints: Alois - 5
      Schwarzenegger - 14
      Arnold - 6

23

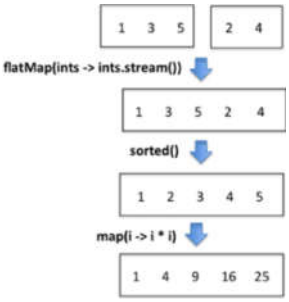
Using flatMap Method in Stream

24

Using FlatMap

- **Let's suppose we have a List of List<Integer>**

```
List<List<Integer>> intsOfInts = Arrays.asList(
                                 Arrays.asList(1, 3, 5),
                                 Arrays.asList(2, 4));
intsOfInts.stream()
        .flatMap(ints -> ints.stream())
            .sorted()
            .map(i -> i * i)
            .forEach(System.out::println);
```



  - This prints the squares of the values 1 to 5.
- To convert those streams into Integer elements, we call the **flatMap()** method. After the call to flatMap(), we have a stream of Integers. We can now perform operations such as **sorted(),** and **map()** to process or transform those elements.

25

Map-Filter-Reduce with average() and Optionals

26

Using reduce()

- **The methods count(), average(), min(), max(), and sum() are all reduction methods already defined on streams.**
  - You can also define your own reductions using the reduce() method.
  - If we look at the type signature of DoubleStream.reduce(), we see:

```
OptionalDouble reduce(DoubleBinaryOperator op)
```

  - To write our own method to sum all the values in the stream using reduce(), we pass a **DoubleBinaryOperator** to reduce(), which adds the two arguments together and returns the sum:

```
OptionalDouble sum =
    readings.stream()
        .mapToDouble(r -> r.value)
        .reduce((v1, v2) -> v1 + v2);
if (sum.isPresent()) {
    System.out.println("Sum of all readings: " + sum.getAsDouble());
}
```

    - And we see the output: `Sum of all readings: 2843.8600000000006`
  - There's another reduce() method in DoubleStream that takes an ***identity*** along with an accumulator function and returns a double:

```
double reduce(double identity, DoubleBinaryOperator op)
```

    - The identity argument serves two roles: it provides an initial value and a value to return if the stream is empty.

```
double sum = readings.stream()
   .mapToDouble(r -> r.value)
   .reduce(0.0, (v1, v2) -> v1 + v2);              // provide an identity
value
System.out.println("Sum of all readings: " + sum); // print 0.0 if stream
                                                   // is empty
```