9

Lambda Operations

ORACLE

Objectives

After completing this lesson, you should be able to:

- Extract data from an object by using map
- Describe the types of stream operations
- Describe the Optional class
- Describe lazy processing
- Sort a stream
- Save results to a collection by using the `collect` method
- Group and partition data by using the `Collectors` class

Streams API

- Streams
  - `java.util.stream`
  - A sequence of elements on which various methods can be chained:
- The Stream class has these properties:
  - Immutable data
  - Can only be used once
  - Encourages fluent programming through method chaining
- The Java API doc gives details of all `Stream` methods.
- Classes
  - `Stream<T>` handles non-numerical objects.
  - `DoubleStream`, `IntStream`, `LongStream` handle primitive `int`, `long`, and `double` types.

A stream pipeline consists of a source, zero, or more intermediate operations (which transform a stream into another stream) and a terminal operation that ends the use of a stream.

To perform a computation, stream operations are composed into a stream pipeline.

A stream pipeline consists of:

- **A source:** An array, a collection, a generator function, an I/O channel
- Zero or more intermediate operations, which transform a stream into another stream, e.g. **`filter`**
- A terminal operation, which produces a result or side effect, e.g. **`count`** or **`forEach`**

Streams may be lazy. Computation on the source data is performed only when the terminal operation is initiated, and source elements are consumed only if needed.

Types of Operations

- Intermediate
  - `filter()` `map()` `peek()` `dropWhile()`
- Intermediate short-circuit
  - `limit()` `takeWhile()`
- Terminal
  - `forEach()` `count()` `sum()` `average()` `min()` `max()` `collect()`
- Terminal short-circuit
  - `findFirst()` `findAny()` `anyMatch()` `allMatch()` `noneMatch()`
  - `takeWhile()`

The above is a list of stream methods by their operation type.

Extracting Data with Map

`map(Function<? super T,? extends R> mapper)`

- A map takes one `Function` as an argument.
  - A `Function` takes one generic type and returns the same type or something else.
- Primitive versions of `map` method
  - `mapToInt` `mapToLong` `mapToDouble`

The map method is typically used to extract data from a field and perform a calculation or operation. The results of the mapping operation are returned as a stream.

Taking a Peek

`peek(Consumer<? super T> action)`

- The peek method performs the operation specified by the lambda expression and returns the elements to the stream.
- Useful for printing intermediate results

The `peek` method of the `Stream` class allows you to look at element data in the stream. After `peek` is called, all elements in the current stream are returned to the stream and are available to the next stream in the pipeline.

Caution: With the `peek` method, you can change element data in the stream. Any changes will be made to the underlying collection. However, this would not be a best practice as the data would not be accessed in a thread-safe manner. Manipulating the data in this way is strongly discouraged.

Search Methods: Overview

- `findFirst()`
  - Returns the first element that meets the specified criteria
- `allMatch()`
  - Returns `true` if all the elements meet the criteria
- `noneMatch()`
  - Returns `true` if none of the elements meet the criteria
- All of the above are short-circuit terminal operations.

The `findFirst` method of the `Stream` class finds the first element in the stream specified by the filters in the pipeline. The `findFirst` method is a terminal short-circuit operation. A terminal operation ends the processing of a pipeline.

The `allMatch` method returns whether all elements of this stream match the provided `predicate`. The method may not evaluate the `predicate` on all elements if not necessary for determining the result. If the stream is empty, `true` is returned and the `predicate` is not evaluated.

The `noneMatch` method returns whether no elements of this stream match the provided `predicate`. It will not evaluate the `predicate` on all elements if this is not necessary for determining the result. If the stream is empty, `true` is returned and the `predicate` is not evaluated.

Search Methods

- Nondeterministic search methods
  - Used for nondeterministic cases, in effect, situations where parallel is more effective
  - Results may vary between invocations.
- `findAny()`
  - Returns the first element found that meets the specified criteria
  - Results may vary when performed in parallel.
- `anyMatch()`
  - Returns true if any elements meet the criteria
  - Results may vary when performed in parallel.

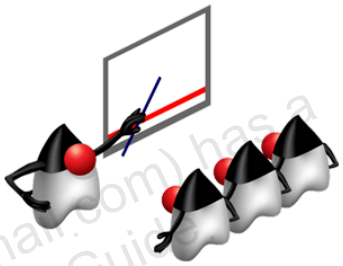
Nondeterministic means that the search may return a different result on each invocation, but any of these are correct and usable.

The `findAny` method returns an `Optional<T>` describing some element of the stream or an empty `Optional<T>` if the stream is empty. The behavior of this operation is explicitly nondeterministic; it is free to select any element in the stream. This is to allow for maximal performance in parallel operations; the cost is that multiple invocations on the same source may not return the same result. (If a stable result is desired, use `findFirst()` instead.) This is a short-circuiting terminal operation.

The `anyMatch` method returns whether any elements of this stream match the provided `predicate`. The method may not evaluate the `predicate` on all elements if it is not necessary for determining the result. If the stream is empty, `false` is returned and the `predicate` is not evaluated. This is a short-circuiting terminal operation.

Optional Class

- `Optional<T>`
  - A container object that may or may not contain a non-null value
  - If a value is present, `isPresent()` returns true.
  - `get()` returns the value.
  - Many other methods available including `stream` to return a new `Stream` object if necessary
  - In `java.util` package
- Optional primitives
  - `OptionalDouble OptionalInt OptionalLong`



An `Optional<T>` is a container object that may or may not contain a non-null value. If a value is present, `isPresent()` returns `true` and `get()` returns the value. There are a number of additional methods that can be used with this class. See the API documentation for further details.

Short-Circuiting Example

Performs only required operations

Using `findFirst` short-circuiting terminal operation.

```
== First CO Bonus ==
Stream start
Stream start
Stream start
Stream start
Stream start
Stream start
Executives
CO Executives
```

```
== CO Bonuses ==
Stream start
Stream start
Stream start
Stream start
Stream start
Stream start
Executives
CO Executives
  Bonus paid: $7,200.00
Stream start
Executives
CO Executives
  Bonus paid: $6,600.00
Stream start
Executives
CO Executives
  Bonus paid: $8,400.00
```

Using `forEach` terminal operation.

The slide shows two lists of operations on a list of Employees. The list on the right must go through all the employee elements as it uses the `forEach` terminal operation. The list on the left uses the `findFirst` method and, thus, when the first element is found, stream processing terminates.

Stream Data Methods

`count()`

- Returns the count of elements in this stream

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

- Returns the maximum element of this stream according to the provided `Comparator`

`min(Comparator<? super T> comparator)`

- Returns the minimum element of this stream according to the provided `Comparator`

The `count` method returns the number of elements in the current stream. This is a terminal operation.

The `max` method returns the highest matching value given a `Comparator` to rank elements. The `max` method is a terminal operation.

The `min` method returns the lowest matching value given a `Comparator` to rank elements. The `min` method is a terminal operation.

Performing Calculations

Primitive streams have `average` and `sum` methods:

- `DoubleStream`, `IntStream`, `LongStream`

`average()`

- Returns an `OptionalDouble` describing the arithmetic mean of elements of this stream
- Returns an empty `Optional` if this stream is empty

`sum()`

- Returns the sum of elements in this stream

The `average` method returns the average of a list of values passed from a stream. The `average` method is a terminal operation.

The `sum` method calculates a sum based on the stream passed to it. Notice that the `mapToDouble` method is called before the stream is passed to sum. If you look at the `Stream` class, no `sum` method is included. Instead, a `sum` method is included in the primitive version of the `Stream` class, `IntStream`, `DoubleStream`, and `LongStream`. The `sum` method is a terminal operation.

Sorting

`sorted()`

- Returns a stream consisting of the elements sorted according to natural order

`sorted(Comparator<? super T> comparator)`

- Returns a stream consisting of the elements sorted according to the `Comparator`

The `sorted` method can be used to sort stream elements based on their natural order. This is an intermediate operation.

Comparator Updates

`comparing(Function<? super T,? extends U> keyExtractor)`

- Allows you to specify any field to sort on based on a method reference or lambda
- Primitive versions of the Function also supported

`thenComparing(Comparator<? super T> other)`

- Specify additional fields for sorting.

`reversed()`

- Reverse the sort order by appending to the method chain.

The `sorted` method can also take a `Comparator` as a parameter. Combined with the `comparing` method, the `Comparator` class provides a great deal of flexibility when sorting a stream.

The `thenComparing` method can be added to the `comparing` method to do a multilevel sort on the elements in the stream. The `thenComparing` method takes a `Comparator` as a parameter just like the `comparing` method.

The `reversed` method can be appended to a pipeline, thus reversing the sort order of the elements in the stream.

Saving Data from a Stream

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

- Allows you to save the result of a stream to a new data structure
- A number of useful collectors are available from the `Collectors` class
  - Examples
  - `stream().collect(Collectors.toList());`
  - `stream().collect(Collectors.toMap() );`
- If a static import of the `Collectors` class is used in the source file, the code can be simplified for readability to just the method call:
  - `stream().collect(toList());`

toList and toMap are just two static methods of the Collectors class that return a Collector.

The `collect` method allows you to save the results of all the filtering, mapping, and sorting that takes place in a pipeline. Notice how the `collect` method is called. It takes a `Collector` as a parameter. The `Collectors` class provides a number of collectors that can be combined in many ways to return the elements remaining in a pipeline after intermediate operations.

The Collectors class and the many collectors that it provides is covered in much more detail in the lesson titled “Terminal Operations: Collectors.”

Collectors Class

- **`averagingDouble(ToDoubleFunction<? super T> mapper)`**
  - Produces the arithmetic mean of a double-valued function applied to the input elements
- **`groupingBy(Function<? super T,? extends K> classifier)`**
  - A "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a map
- **`joining()`**
  - Concatenates the input elements into a String, in encounter order
- **`partitioningBy(Predicate<? super T> predicate)`**
  - Partitions the input elements according to a Predicate

The `groupingBy` method of the `Collectors` class allows you to generate a `Map` based on the elements contained in a stream. The keys are based off a selected field in a class. Matching objects are placed into an `ArrayList` that becomes the value for the key.

The `joining` method of the `Collectors` class allows you to join together elements returned from a stream.

The `partitioningBy` method offers an interesting way to create a `Map`. The method takes a `Predicate` as an argument and creates a `Map` with two `boolean` keys. One key is `true` and includes all the elements that meet the true criteria of the `Predicate`. The other key, `false`, contains all the elements that resulted in `false` values as determined by the `Predicate`.

Quick Streams with `Stream.of`

- The `Stream.of` method allows you to easily create a stream.

```
public static void main(String[] args) {

   Stream.of("Monday", "Tuesday","Wednesday", "Thursday")
     .filter(s -> s.startsWith("T"))
     .forEach(s -> System.out.println("Matching Days: " + s));
 }
```

The `Stream.of` method can be used to create a stream out of an array of elements. The elements can be listed as shown in the slide or from the results of method calls.

Flatten Data with `flatMap`

- Use the `flatMap` method to flatten data in a stream.

```
Path file = new File("tempest.txt").toPath();

try{

    long matches = Files.lines(file)
        .flatMap(line -> Stream.of(line.split(" ")))
        .filter(word -> word.contains("my"))
        .peek(s -> System.out.println("Match: " + s))
        .count();

    System.out.println("# of Matches: " + matches);
```

Because flatMap returns a stream, the lambda function must produce a stream.

The `flatMap` method can be used to convert data into a stream. When called on a stream, it has the effect of turning each element of the stream into a new stream. It therefore is used to “flatten” the data structure.

In order to search for the occurrence of a particular word, you need a stream of type String, where each word is a String element. Then filter to only include the word of your choice in the stream, peek, to show a match being made, and finally count, to count the matches.

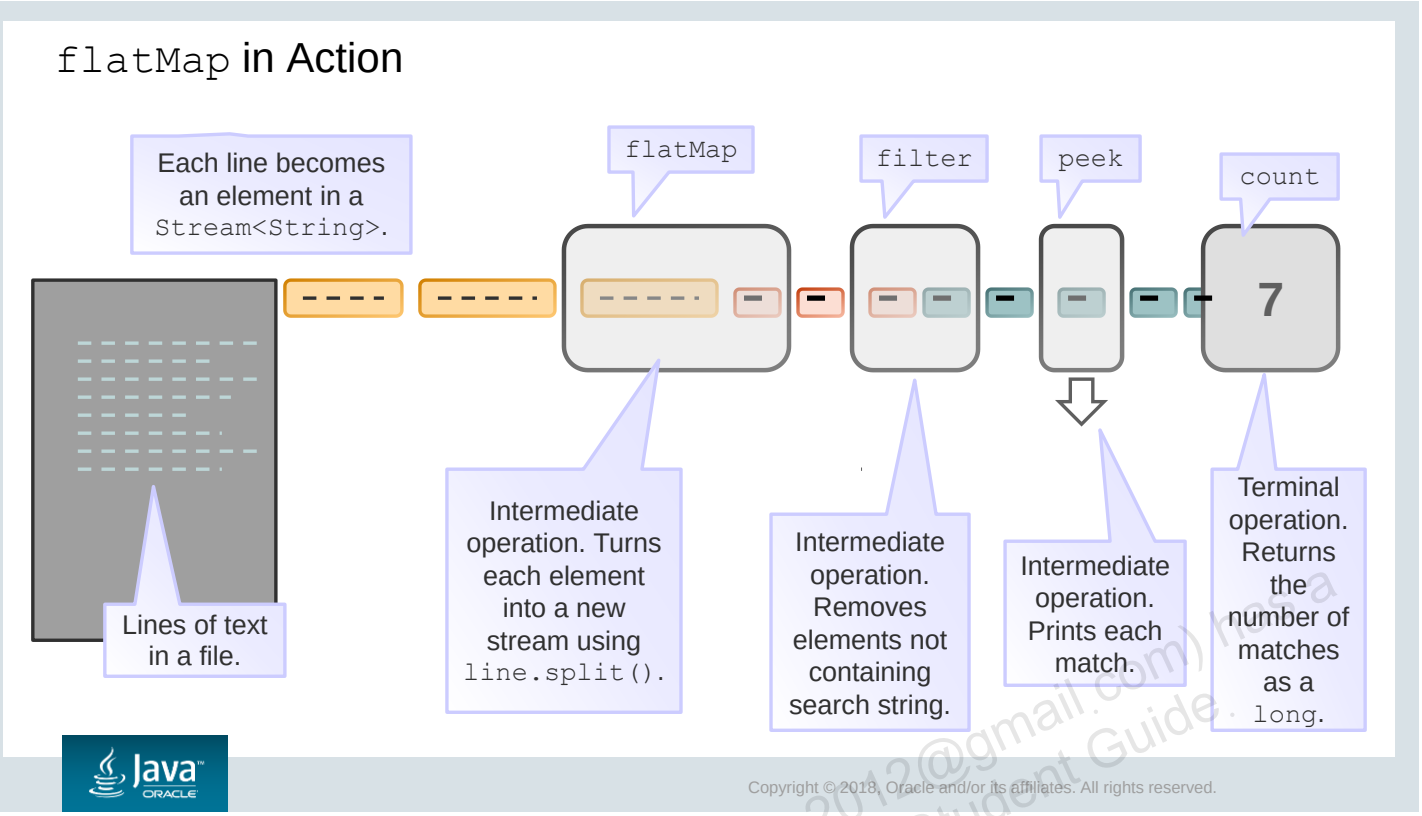
Output for peek is:

- **Match:** your
- **Match:** yourself
- **Match:** your

Output for matches (i.e. count): 3

But why is flatMap needed?

Because `File.lines(file)` returns a stream of type `String` of entire lines. To instead have a stream of every word, each line is used to generate a new stream of single words.



In order to search for the occurrence of a particular word, you need a stream of type String, where each word is a String element. Then filter to only include the word of your choice in the stream, peek, to show a match being made, and finally count, to count the matches.

Output for peek is:

- **Match:** my
- **Match:** rummy
- **Match:** myself
- **Match:** …

Output for matches (i.e. count): 7

But why is flatMap needed?

Because `File.lines(file)` returns a stream of type `String` of entire lines. To instead have a stream of individual words, each line is used to generate a new stream of single words.

Summary

In this lesson, you should have learned how to:

- Extract data from an object using map
- Describe the types of stream operations
- Describe the Optional class
- Describe lazy processing
- Sort a stream
- Save results to a collection by using the `collect` method
- Group and partition data by using the `Collectors` class

Practice 9: Overview

This practice covers the following topics:

- Practice 9-1: Using Map and Peek
- Practice 9-2: FindFirst and Lazy Operations
- Practice 9-3: Analyzing Transactions with Stream Methods
- Practice 9-4: Performing Calculations with Primitive Streams
- Practice 9-5: Sorting Transactions with Comparator
- Practice 9-6: Collecting Results with Streams
- Practice 9-7: Joining Data with Streams
- Practice 9-8: Grouping Data with Streams