10

Lambda Operations

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 converts collection to a pipeline.
  - Immutable data
  - Can only be used once
  - Method chaining
- Java API doc *is your friend*
- Classes
  - `DoubleStream`, `IntStream`, `LongStream`

Types of Operations

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

Extracting Data with Map

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

- A map takes one `Function` as an argument.
  - A `Function` takes one generic and returns something else.
- Primitive versions of `map`
  - `mapToInt()` `mapToLong()` `mapToDouble()`

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.
- Great for printing intermediate results

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.

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.

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.
  - Found in `java.util`.
- Optional primitives
  - `OptionalDouble OptionalInt OptionalLong`

Lazy Operations

- Lazy operations:
  - Can be optimized
  - Perform only required operations

```
== 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
```

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`

Performing Calculations

`average()`

- Returns an optional describing the arithmetic mean of elements of this stream
- Returns an empty optional if this stream is empty
- Type returned depends on primitive class.

`sum()`

- Returns the sum of elements in this stream
- Methods are found in primitive streams:
  - `DoubleStream`, `IntStream`, `LongStream`

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`

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.

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
- Relies on the `Collectors` class
- Examples
  - `stream().collect(Collectors.toList());`
  - `stream().collect(Collectors.toMap());`

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

Quick Streams with `Stream.of`

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

```
11  public static void main(String[] args) {
12
13    Stream.of("Monday", "Tuesday","Wedensday", "Thursday")
14      .filter(s -> s.startsWith("T"))
15      .forEach(s -> System.out.println("Matching Days: " +
    s));
16  }
```

Flatten Data with `flatMap`

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

```
17          Path file = new File("tempest.txt").toPath();
18
19          try{
20
21              long matches = Files.lines(file)
22                  .flatMap(line -> Stream.of(line.split(" ")))
23                  .filter(word -> word.contains("my"))
24                  .peek(s -> System.out.println("Match: " + s))
25                  .count();
26
27              System.out.println("# of Matches: " + matches);
```

Summary

After completing this lesson, you should be able 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 Overview

- Practice 10-1: Using Map and Peek
- Practice 10-2: FindFirst and Lazy Operations
- Practice 10-3: Analyze Transactions with Stream Methods
- Practice 10-4: Perform Calculations with Primitive Streams
- Practice 10-5: Sort Transactions with Comparator
- Practice 10-6: Collect Results with Streams
- Practice 10-7: Join Data with Streams
- Practice 10-8: Group Data with Streams