Java Review Session 5

CS 5004

Topics Covered

Java Syntax

- Stream Processing
  - Creating Streams
  - Transforming Streams
  - Collecting Streams
- Lambda Expressions
- The Optional Type

Stream processing

- Streams let you process data by specifying what you want to have done
- Sometimes can be easier to understand since they express the intent of the programmer more clearly than explicit loops
- Useful for processing big/large data
  - Can distribute work over multiple processors
- Provide a way to process elements from a source
  - Processes the elements from a source on demand without storing them in memory
  - Operate on elements as needed rather than all at once (more efficient)
- Stream workflow:
  - 1. Stream Creation, Stream Transformation, Stream Collection

Streams vs Collections?

- Unlike a collection:
  - Streams do not store data (it comes from elsewhere - a collection, file, database, etc.)
  - Streams are immutable (we can’t insert, add, or remove items from them)
    - We can create a new stream from an existing stream though
- Stream processing is lazy
  - They build up a chain of operations (a pipeline) to describe how the elements should be processed
  - Waits for a terminal operation to be called (collect, toList, forEach)
- Streams have short circuiting

1. Creating Streams

Static `Stream.of()` method

If you have a Java Collections can call the `.stream()` method

This obtains a stream for the elements in a collection

You can turn any collection into a stream

Some utility methods yield streams

Table 1 Producing Streams

Example	Result
`Stream.of(1, 2, 3)`	A stream containing the given elements. You can also pass an array.
`Collection<String> coll = . . .; coll.stream()`	A stream containing the elements of a collection.
`Files.lines(path)`	A stream of the lines in the file with the given path. Use a try-with-resources statement to ensure that the underlying file is closed.
`Stream<String> stream = . . .; stream.parallel()`	Turns a stream into a parallel stream.
`Stream.generate(() -> 1)`	An infinite stream of ones (see Special Topic 19.1).
`Stream.iterate(0, n -> n + 1)`	An infinite stream of `Integer` values (see Special Topic 19.1).
`IntStream.range(0, 100)`	An `IntStream` of `int` values between 0 (inclusive) and 100 (exclusive)—see Section 19.8.
`Random generator = new Random(); generator.ints(0, 100)`	An infinite stream of random `int` values drawn from a random generator—see Section 19.8.
`"Hello".codePoints()`	An `IntStream` of code points of a string—see Section 19.8.

Image: Cay Horstmann

2. Transforming Streams

You can string together as many map and filter operations as you like

They are evaluated lazily (without creating intermediate results)

The map method applies a function to all elements of a stream, yielding another stream with the results

The filter method yields a stream of all the elements fulfilling a condition

Table 3 Stream Transformations

Example	Comments
`stream.filter(`*condition*`)`	A stream with the elements matching the condition.
`stream.map(`*function*`)`	A stream with the results of applying the function to each element.

`stream.mapToInt(`*function*`)`
`stream.mapToDouble(`*function*`)`
`stream.mapToLong(`*function*`)`
	A primitive-type stream with the results of applying a function with a return value of a primitive type—see Section 19.8.

`stream.limit(n)` `stream.skip(n)`	A stream consisting of the first `n`, or all but the first `n` elements.

`stream.distinct()`
`stream.sorted()`
`stream.sorted(`*comparator*`)`
	A stream of the distinct or sorted elements from the original stream.

Image: Cay Horstmann

More Stream Transformations

- `stream.limit(n)`
- `stream.skip(n)`
- `stream.distinct()`
- `stream.sorted()`

3. Collecting results

Often we want to put the final stream values back into an array or collection

Can use the `collect()` method

Now we can change collections into streams and then streams back into collections

Table 2 Collecting Results from a `Stream<T>`

Example	Comments
`stream.toArray(T[]::new)`	Yields a `T[]` array.
`stream.collect(Collectors.toList())` `stream.collect(Collectors.toSet())`	Yields a `List<T>` or `Set<T>`.
`stream.collect(Collectors.joining(", ")`	Yields a string, joining the elements by the given separator. Only for `Stream<String>`.
`stream.collect(Collectors.groupingBy(` `  keyFunction, collector)`	Yields a map that associates group keys with collected group values—see Section 19.9.

Image: Cay Horstmann

Lambda Expressions

- A lambda expression consists of one or more parameter variables, an arrow, and and expression or block yielding the result
  - Has a parameter variable that is mapped to a result
  - Sometimes the compiler can figure out the type of the parameter (other times you need to specify)
  - Can also have multiple parameters: `(v, w) -> v.lenth() - w.length()`
- When the result is simple, you can use a single expression
  - You can also distribute code over multiple lines enclosing the statements in brackets
  - Then use the return keyword for the result (just like in methods)
- When a lambda expression consists of just one method call you can use a shorter syntax called a method reference
  - A class name :: method name (is equivalent to the lambda expression for this)
  - E.g. `String::toUpperCase()` is equivalent to `(String w) -> w.toUpperCase()`

Lambda Expressions Cont.

Syntax 19.1 Lambda Expressions

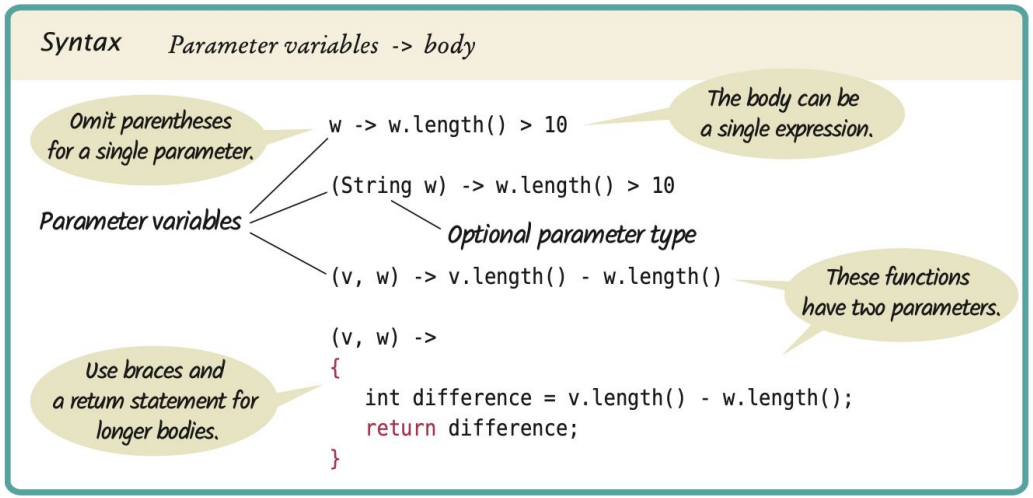

Image: Cay Horstmann

The Optional Type, Primitive-Type Streams, and Grouping

- The Optional class is a wrapper for objects that may or may not be present
  - It holds a value or indication that no value is present
- Helps us reduce the occurrence of NullPointerExceptions

- It can be inefficient to use wrappers on streams since you need a separate wrapper for each element in the stream
- There are a few primitive type streams (specialized stream interfaces):
  - IntStream stream = IntStream.of(1, 2, 3, 4, 5);
  - Have some specialized operations like sum(), min(), max(), etc.
  - Have for IntStream, DoubleStream, and LongStream

Some Primitive-type Stream Methods

Table 5 Computing Results from a `Stream<T>`

Example	Comments
`stream.count()`	Yields the number of elements as a `long` value.
`stream.findFirst()` `stream.findAny()`	Yields the first, or an arbitrary element as an `Optional<T>`—see Section 19.6.
`stream.max(`*comparator*`)` `stream.min(`*comparator*`)`	Yields the largest or smallest element as an `Optional<T>`—see Section 19.7.

`pstream.sum()`
`pstream.average()`
`pstream.max()`
`pstream.min()`
	The sum, average, maximum, or minimum of a primitive-type stream—see Section 19.8.

`stream.allMatch(`*condition*`)`
`stream.anyMatch(`*condition*`)`
`stream.noneMatch(`*condition*`)`
	Yields a `boolean` variable indicating whether all, any, or no elements match the condition—see Section 19.7.

`stream.forEach(`*action*`)`	Carries out the action on all stream elements—see Section 19.7.

Image: Cay Horstmann