Streams in Java 8

Start programming in a more functional style

Background

Who am I?

- Tobias Coetzee
- I'm a Technical Lead at BBD
- I present the Java Expert Level Certifications at BBD (EJB, JPA, etc.)
- I'm currently a banker
- Part of the JoziJug organising committee
- I like to run
- This partially presentation is based on a Simon Ritter course
- @tobiascode
- https://github.com/Jozi-JUG/streamtraining

Outcomes

What are the take away's?

- Know the differences between functional and imperative programming
- Be able to define the elements of a stream
- Use intermediate and terminal operations
- Create streams from different data types
- Avoid NullPointerExceptions with the Optional Class
- Debug streams

What Did We Do Last Time?

Lambdas

Summary

- CPU's are getting much faster, but we are getting more cores to use.
- Inject functionality into methods, the same way we inject values into methods.
- Lambda syntax and rules, `(parameters) -> {lambda body}`
- Functional interfaces are interfaces that have only one abstract method.
- Lambdas can be used where the type is a functional interface.
- Use lambdas as method parameters or assign them to variables.
- The java.util.function package already gives us a few functional interfaces out of the box.
- Method and constructor references as short hands notation, **::**.
- There are a few new methods that can use lambdas in Java 8, e.g. `forEach(Consumer c)`.

What Is Functional Programming?

Functional Programming

Remember this game?

Functional Programming

Imperative

- Values associated with names can be changed, i.e. mutable state.
- The order of execution is defined as a contract.
- Repetition is external and explicit, we are responsible for the “how”.

Functional (Declarative)

- Values associated with names are set once and cannot be changed, i.e. immutable state.
- Order of execution is not defined
- Repetition is through the use of recursion or internal repetition, the “how” is hidden away.

Thinking about a problem in terms of immutable values and functions that translate between them.

"Multithreaded programming, requiring synchronised access to shared, mutable state, is the assembly language of concurrency"

The Elements of a Stream

Elements of a Stream

What are streams?

- Streams give us functional blocks to better process collections of data with, monads.
- We can chain these blocks together to process collections of data.
- Streams aren't another data structure.
- Streams can process an infinite list of data.
- Streams use internal iteration meaning we don't have to code external iteration, the "how".
- Streams support functional programming as suppose to imperative programming.

"The purpose of streams isn't just to convert from one collection to another; it's to be able to provide a common set of operations over data"

Elements of a Stream

Structure of a Stream

- A stream consists of 3 types of things
  1. A source
  2. Zero or more intermediate operations
  3. A terminal operation


```
result = albums.stream()
         .filter(track -> track.getLength() > 300)
         .map(Track::getName)
         .collect(Collectors.toSet());
```

Demo: Refactor Code

Elements of a Stream

How they work

- The pipeline is only evaluated when the terminal operation is called.
- The terminal operations pulls the data, the source doesn't push it.
- Uses the stream characteristics to help identify optimisations.
- This allows intermediate operations to be merged:
  - Avoiding multiple redundant passes on data
  - Short-circuit operations
  - Laze evaluation
- The stream takes care of the "how".

Intermediate Operations

Intermediate Operations

Intermediate

- A stream provides a sequence of elements.
- Supports either sequential or parallel aggregate operations.
- Most operations take a parameter that describes its behaviour, the “what”.
  - Typically using lambda expressions.
  - Must be non-interfering
  - Stateless
- Streams can switch between sequential and parallel, but all processing is either done sequential or parallel, last call wins.

Intermediate Operations

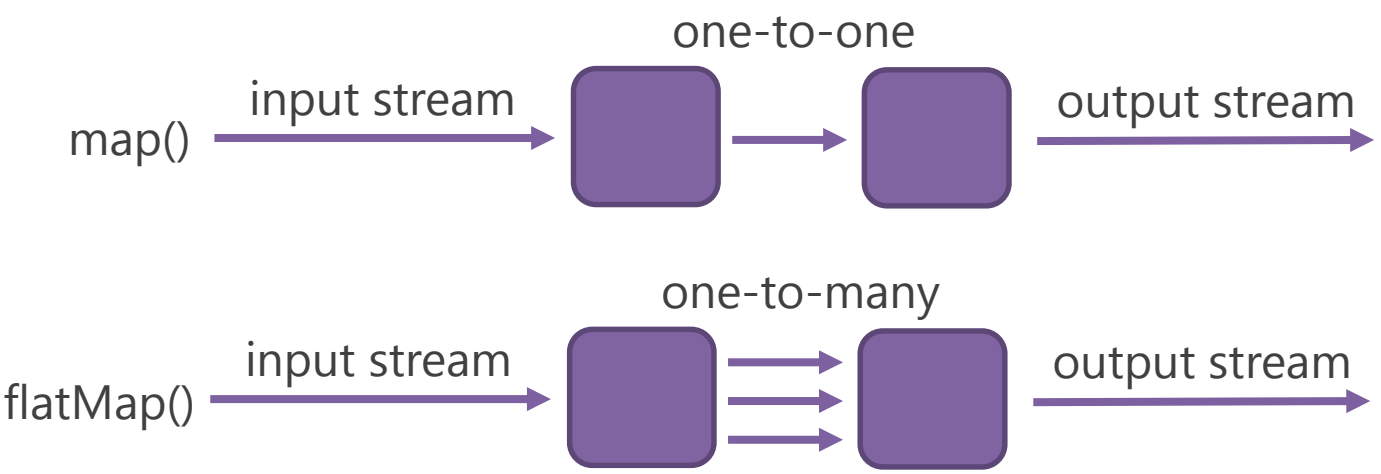
Filtering and Mapping

Interface	Description
`filter(Predicate)`	Returns a stream with only those elements that return true for the Predicate
`map(Function)`	Return a stream where the given Function is applied to each element on the input stream
`mapToInt, mapToDouble, mapToLong`	Like map(), but producing streams of primitives rather than objects.
`distinct()`	Returns a stream with no duplicate elements

Intermediate Operations

Map() and FlatMap()

- Map values from a stream either as one-to-one or one-to-many, but still only produce one stream.



Intermediate Operations

Sizing and Sorting

Interface	Description
`skip(long)`	Returns a stream that skips the first *n* elements of the input stream
`limit(long)`	Return a stream that only contains the first *n* elements of the input stream
`sorted(Comparator)`	Returns a stream that is sorted with the order determined by the Comparator. With no arguments sorts by natural order.
`unordered()`	Returns a stream that is unordered (used internally). Can improve efficiency of operations like `distinct()` and `groupingBy()`

Terminal Operations

Terminal Operations

Terminal

- Terminates the pipeline of the operations on the stream.
- Only at this point is any processing performed.
- This allows for optimisation of the pipeline:
  - Lazy evaluation
  - Merged/fused operations
  - Elimination of redundant operations
  - Parallel execution
- Generates an explicit result of a side effect.

Intermediate Operations

Matching Elements and Iteration

Interface	Description
`findFirst(Predicate)`	The first element that matches predicate
`findAny(Predicate)`	Like `findFirst()`, but for a parallel stream
`allMatch(Predicate)`	All elements in the stream match predicate
`anyMatch(Predicate)`	Any element in the stream matches predicate
`noneMatch(Predicate)`	No elements match the predicate
* `forEach(Consumer)`	Performs an action on each element
* `forEachOrdered(Consumer)`	Like above, but ensures order is respected when used for parallel stream

* Encourages non-functional programming style

Intermediate Operations

Collecting Results and Numerical Results

Interface	Description
* `collect(Collector)`	Performs a mutable reduction on a stream
`toArray()`	Returns an array containing the elements of the stream
`count()`	Returns how many elements are in the stream
`max(Comparator)`	The maximum value element of the stream, returns an `Optional`
`min(Comparator)`	The minimum value element of the stream, returns an `Optional`
`average()`	Return the arithmetic mean of the stream, returns an `Optional`
`sum()`	Returns the sum of the stream elements

* There are a lot of built-in Collectors

Terminal Operations

Creating a Single Result

- The `collect` function isn't the only option.
- `reduce(BinaryOperator accumulator)`
- Also called folding in FP.
- Performs a reduction on the stream using the `BinaryOperator`.
- The `accumulator` takes a partial result and the next element and returns a new partial result as an `Optional`.
- Two other versions
  - One that takes an initial value.
  - One that takes an initial value and `BiFunction`.

Exercise: Refactor Code

Stream Sources

Stream Sources

JDK 8 Libraries

- There are 95 methods in 23 classes that return a stream, most are intermediate operations though.
- Leaves 71 methods in 15 classes that can be used as practical sources.
- There are numerous places to get stream sources.
  - Streams from values
  - Empty streams
  - Streams from functions
  - Streams from arrays
  - Streams from collections
  - Streams from files
  - Streams from other sources

Stream Sources

Collection Interface

- `stream()`
  - Provides a sequential stream of elements in the collection.
- `parallelStream()`
  - Provides a parallel stream of elements in the collection.
  - Uses the fork-join framework for implementation.
  - Only Collection can provide a parallel stream directly.

Arrays Class

- `stream()`
  - Array is a collection of data, so logical to be able to create a stream.
  - Provides a sequential stream.
  - Overloaded methods for different types, int, double, long, Object

Stream Sources

Some other Classes

- Files Class: `find, list, lines, walk`.
- Random numbers to produce finite or infinite streams with or without seeds: `Random, ThreadLocalRandom, SplittableRandom`.
- BufferReader, JarFile, ZipFile, Pattern, Charsequence, etc.

Stream Sources

Primitive Streams

- IntStream, DoubleStream, LongStream are primitive specialisations of the Stream interface.
- `range(int, int)`, `rangeClosed(int, int)`
  - A stream from a start to an end value (exclusive or inclusive)
- `generate(IntSupplier)`, `iterate(int, IntUnaryOperator)`
  - An infinite stream created by a given Supplier.
  - `iterate` uses a seed to start the stream.

Optional

Optional

Problems with `null`

- Certain situations in Java return a result which is a null, i.e. the reference to an object isn't set.
- It tries to help eliminate `NullPointerException`'s.
- Terminal operations like min(), max(), may not return a direct result, suppose the input stream is empty?
- Introducing `Optional<T>`:
  - It is a container for an object reference (`null`, or real object).
  - It is a stream with either 0 or 1 elements, but never `null`.
- Can be used in powerful ways to provide complex conditional handling.
- Doesn't stop developers from returning `null`, but an Optional tells you do maybe rather check.
- Was added for the Stream API, but you can also use it.

Optional

Creating an Optional

- `<T> Optional<T> empty()`: Returns an empty `Optional`.
- `<T> Optional<T> of(T value)`: Returns an `Optional` containing the specified `value`. If the specified value is `null`, it throws a `NullPointerException`.
- `<T> Optional<T> ofNullable(T value)`: Same as above, but if the specified `value` is `null`, it returns and empty `Optional`.

Optional

Getting Data from an Optional

- `<T> get()`: Returns the value, but will throw NoSuchElementException is value is empty.
- `<T> orElse(T defaultValue)`: Returns the default value if value is empty.
- `<T> orElseGet(Supplier<? Extends T> defaultSupplier)`: Same as above, but supplier gets the value.
- `<X extends Throwable> T orElseThrow(Supplier<? Extends T> exceptionSupplier)`: If empty throw the exception from the supplier.

Demo: Refactor to Optional

Exercise: Optional

Debugging Streams

Debugging Streams

Solution

- Most IDE's will support debugging Streams and breakpoints on streams.
- As you can pass in the behaviour for each operation via a lambda you can always add a `System.out.println()` or logging as part of the behaviour.
- If you don't want to mess around with your current behaviour you can use the peek operation.
- The `peek(Consumer <? Super T> action)` methods was added for debugging and should only be used for debugging.
- Each element will pass through it and the `action` will be applied to it.

Homework

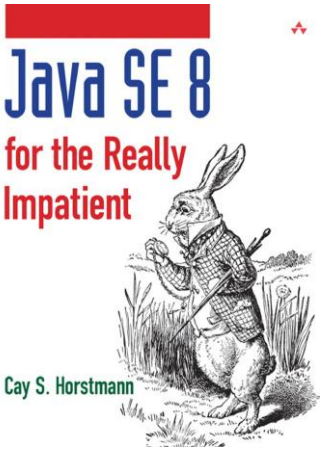
Homework Exercises

How does it work?

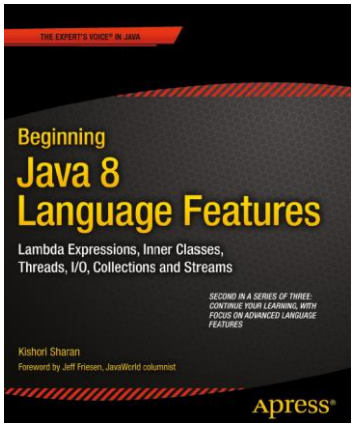
- The exercises for this month are all small exercises that you will complete using the things you have learned during this presentation.
- There are 21 exercises to complete.
- For first month focus on using lambda expressions, method references and some of the new methods that have been added to existing classes.
- There is a template source file that you should use to create the answers to the exercises called Test.java under exercises package the test folder.
- To simplify things the lists and maps you need for the exercises have already been created.
- You just need to focus on the code to solve the exercise.
- The solutions are in the Test.java file in the solutions package, but try all the exercises first before taking a peak.

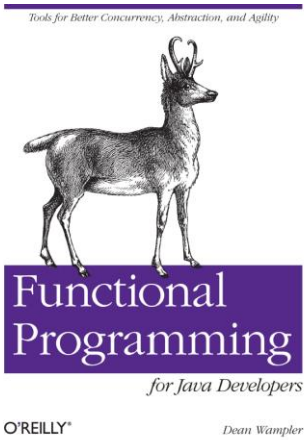
Books

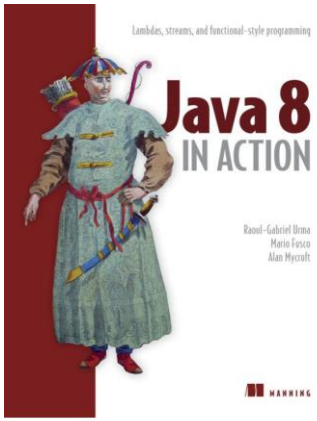
Books on Lambdas and Streams

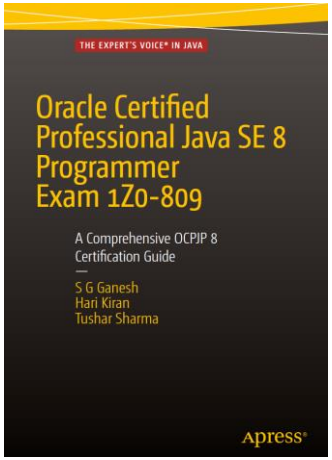


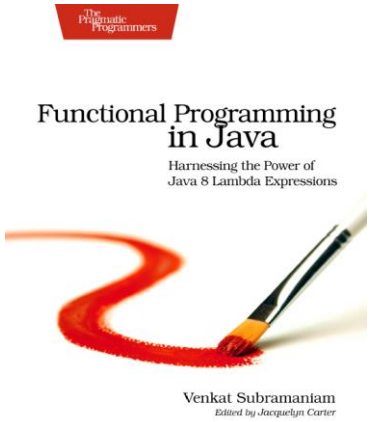
Richard Warburton











Thank you!