Lambdas & Streams In JDK 8: Beyond The Basics

Simon Ritter

Deputy CTO, Azul Systems

@speakjava | azul.com

A clever man learns from his mistakes...

...a wise man learns from other people's

Agenda

- Lambdas and Streams Primer
- Delaying Execution
- Avoiding Loops In Streams
- The Art Of Reduction
- Lambdas and Streams and JDK 9
- Conclusions

Lambdas And Streams Primer

Lambda Expressions In JDK 8

Simplified Parameterised Behaviour

- Old style, anonymous inner classes

```
new Thread(new Runnable {
  public void run() {
    doSomeStuff();
  }
}).start();
```

- New style, using a Lambda expression

```
new Thread(() -> doSomeStuff()).start();
```

Type Inference

- Compiler can often infer parameter types in a lambda expression
  - Inferrence based on target functional interface’s method signature

```
static T void sort(List<T> l, Comparator<? super T> c);

List<String> list = getList();
Collections.sort(list, (String x, String y) -> x.length() > y.length());
```

↓

```
Collections.sort(list, (x, y) -> x.length() - y.length());
```

- Fully statically typed (no dynamic typing sneaking in)
  - More typing with less typing

Functional Interface Definition

- Is an interface
- Must have only one abstract method
  - In JDK 7 this would mean only one method (like `ActionListener`)
- JDK 8 introduced default methods
  - Adding multiple inheritance of types to Java
  - These are, by definition, not abstract
- JDK 8 also now allows interfaces to have static methods
- `@FunctionalInterface` to have the compiler check

Is This A Functional Interface?

```
@FunctionalInterface
public interface Runnable {
  public abstract void run();
}
```

Yes. There is only one abstract method

Is This A Functional Interface?

```
@FunctionalInterface
public interface Predicate<T> {
  default Predicate<T> and(Predicate<? super T> p) {…};
  default Predicate<T> negate() {…};
  default Predicate<T> or(Predicate<? super T> p) {…};
  static <T> Predicate<T> isEqual(Object target) {…};
  boolean test(T t);
}
```

Yes. There is still only one abstract method

Is This A Functional Interface?

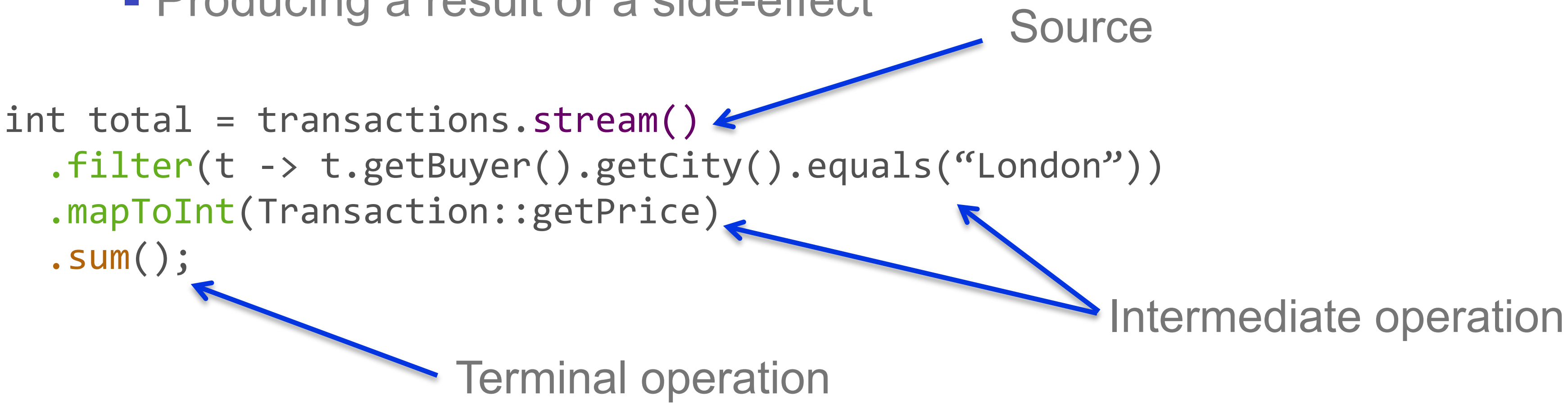
```
@FunctionalInterface
public interface Comparator {
  // Static and default methods elided
  int compare(T o1, T o2);
  ~~boolean equals(Object obj)~~;
}
```

Therefore only one abstract method

The `equals(Object)` method is implicit from the Object class

Stream Overview

- A stream pipeline consists of three types of things
  - A source
  - Zero or more intermediate operations
  - A terminal operation
    - Producing a result or a side-effect



```
int total = transactions.stream()
  .filter(t -> t.getBuyer().getCity().equals(“London”))
  .mapToInt(Transaction::getPrice)
  .sum();
```

Source

Intermediate operation

Terminal operation

Stream Sources

Many Ways To Create

- From collections and arrays
  - `Collection.stream()`
  - `Collection.parallelStream()`
  - `Arrays.stream(T array)` or `Stream.of()`
- Static factories
  - `IntStream.range()`
  - `Files.walk()`

Stream Terminal Operations

- The pipeline is only evaluated when the terminal operation is called
  - All operations can execute sequentially or in parallel
  - Intermediate operations can be merged
    - Avoiding multiple redundant passes on data
    - Short-circuit operations (e.g. `findFirst`)
    - Lazy evaluation
  - Stream characteristics help identify optimisations
    - `DISTINT` stream passed to `distinct()` is a no-op

Optional Class

- Terminal operations like `min()`, `max()`, etc do not return a direct result
- Suppose the input Stream is empty?
- `Optional<T>`
  - Container for an object reference (null, or real object)
  - Think of it like a Stream of 0 or 1 elements
  - use `get()`, `ifPresent()` and `orElse()` to access the stored reference
  - Can use in more complex ways: `filter()`, `map()`, etc
    - `gpsMaybe.filter(r -> r.lastReading() < 2).ifPresent(GPSData::display);`

LambdaExpressions And Delayed Execution

Performance Impact For Logging

- Heisenberg’s uncertainty principle

Always executed

```
logger.finest(getSomeStatusData());
```

- Setting log level to `INFO` still has a performance impact
- Since `Logger` determines whether to log the message the parameter must be evaluated even when not used

Supplier<T>

- Represents a supplier of results
- All relevant logging methods now have a version that takes a Supplier

```
logger.finest(getSomeStatusData());
logger.finest(() -> getSomeStatusData());
```

- Pass a description of how to create the log message
  - Not the message
- If the `Logger` doesn't need the value it doesn't invoke the Lambda
- Can be used for other conditional activities

Avoiding Loops In Streams

Functional v. Imperative

- For functional programming you should not modify state
- Java supports closures over values, not closures over variables
- But state is really useful…

Counting Methods That Return Streams

Still Thinking Imperatively

```
Set<String> sourceKeySet =
  streamReturningMethodMap.keySet();

LongAdder sourceCount = new LongAdder();

sourceKeySet.stream()
  .forEach(c -> sourceCount
    .add(streamReturningMethodMap.get(c).size()));
```

Counting Methods That Return Streams

Functional Way

```
sourceKeySet.stream()
  .mapToInt(c -> streamReturningMethodMap.get(c).size())
  .sum();
```

Printing And Counting Functional Interfaces

Still Thinking Imperatively

```
LongAdder newMethodCount = new LongAdder();

functionalParameterMethodMap.get(c).stream()
  .forEach(m -> {
    output.println(m);

    if (isNewMethod(c, m))
      newMethodCount.increment();
  });

  return newMethodCount.intValue();
```

Printing And Counting Functional Interfaces

More Functional, But Not Pure Functional

```
int count = functionalParameterMethodMap.get(c).stream()
  .mapToInt(m -> {
    int newMethod = 0;
    output.println(m);

    if (isNewMethod(c, m))
      newMethod = 1;

    return newMethod
  })
  .sum();
```

There is still state being modified in the Lambda

Printing And Counting Functional Interfaces

Even More Functional, But Still Not Pure Functional

```
int count = functionalParameterMethodMap.get(nameOfClass)
  .stream()
  .peek(method -> output.println(method))
  .mapToInt(m -> isNewMethod(nameOfClass, m) ? 1 : 0)
  .sum();
```

Strictly speaking printing is a side effect, which is not purely functional

The Art Of Reduction
(Or The Need to Think Differently)

A Simple Problem

- Find the length of the longest line in a file
- Hint: `BufferedReader` has a new method, `lines()`, that returns a Stream

```
BufferedReader reader = ...

reader.lines()
  .mapToInt(String::length)
  .max()
  .getAsInt();
```

Another Simple Problem

- Find the ~~length of the~~ longest line in a file

Naïve Stream Solution

```
String longest = reader.lines().
  sort((x, y) -> y.length() - x.length()).
  findFirst().
  get();
```

- That works, so job done, right?
- Not really. Big files will take a long time and a lot of resources
- Must be a better approach

External Iteration Solution

```
String longest = "";

while ((String s = reader.readLine()) != null)
  if (s.length() > longest.length())
    longest = s;
```

- Simple, but inherently serial
- Not thread safe due to mutable state

Functional Approach: Recursion

```
String findLongestString(String longest, List<String> l, int i) {
  if (l.get(i).length() > longest.length())
    longest = l.get(i);

  if (i < l.length() - 1)
    longest = findLongestString(longest, l, i + 1);

  if (longest.length() > l.get(i).length())
    return longest;
  return l.get(i);
}
```

Recursion: Solving The Problem

```
List<String> lines = new ArrayList<>();

while ((String s = reader.readLine()) != null)
  lines.add(s);

String longest = findLongestString("", lines, 0);
```

- No explicit loop, no mutable state, we’re all good now, right?
- Unfortunately not - larger data sets will generate an OOM exception

A Better Stream Solution

- Stream API uses the well known filter-map-reduce pattern
- For this problem we do not need to filter or map, just reduce

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

- `BinaryOperator` is a subclass of `BiFunction`
  - `R apply(T t, U u)`
- For `BinaryOperator` all types are the same
  - `T apply(T x, T y)`

A Better Stream Solution

- The key is to find the right accumulator
  - The accumulator takes a partial result and the next element, and returns a new partial result
  - In essence it does the same as our recursive solution
  - But without all the stack frames or `List` overhead

A Better Stream Solution

- Use the recursive approach as an acculular for a reduction

```
String longestLine = reader.lines()
  .reduce((x, y) -> {
    if (x.length() > y.length())
      return x;
    return y;
  })
  .get();
```

A Better Stream Solution

- Use the recursive approach as an acculilator for a reduction

```
String longestLine = reader.lines()
  .reduce((x, y) -> {
    if (x.length() > y.length())
      return x;
    return y;
  })
  .get();
```

x in effect maintains state for us, by providing the partial result, which is the longest string found so far

The Simplest Stream Solution

- Use a specialised form of `max()`
- One that takes a Comparator as a parameter

```
reader.lines()
  .max(comparingInt(String::length))
  .get();
```

- `comparingInt()` is a static method on Comparator

```
Comparator<T> comparingInt(
      ToIntFunction<? extends T> keyExtractor)
```

Lambdas And Streams And JDK 9

Additional APIs

- `Optional` now has a `stream()` method
  - Returns a stream of one element or an empty stream
- `Collectors.flatMapping()`
  - Returns a `Collector` that converts a stream from one type to another by applying a flat mapping function

Additional APIs

- Matcher stream support
  - `Stream<MatchResult> results()`
- Scanner stream support
  - `Stream<MatchResult> findAll(String pattern)`
  - `Stream<MatchResult> findAll(Pattern pattern)`
  - `Stream<String> tokens()`

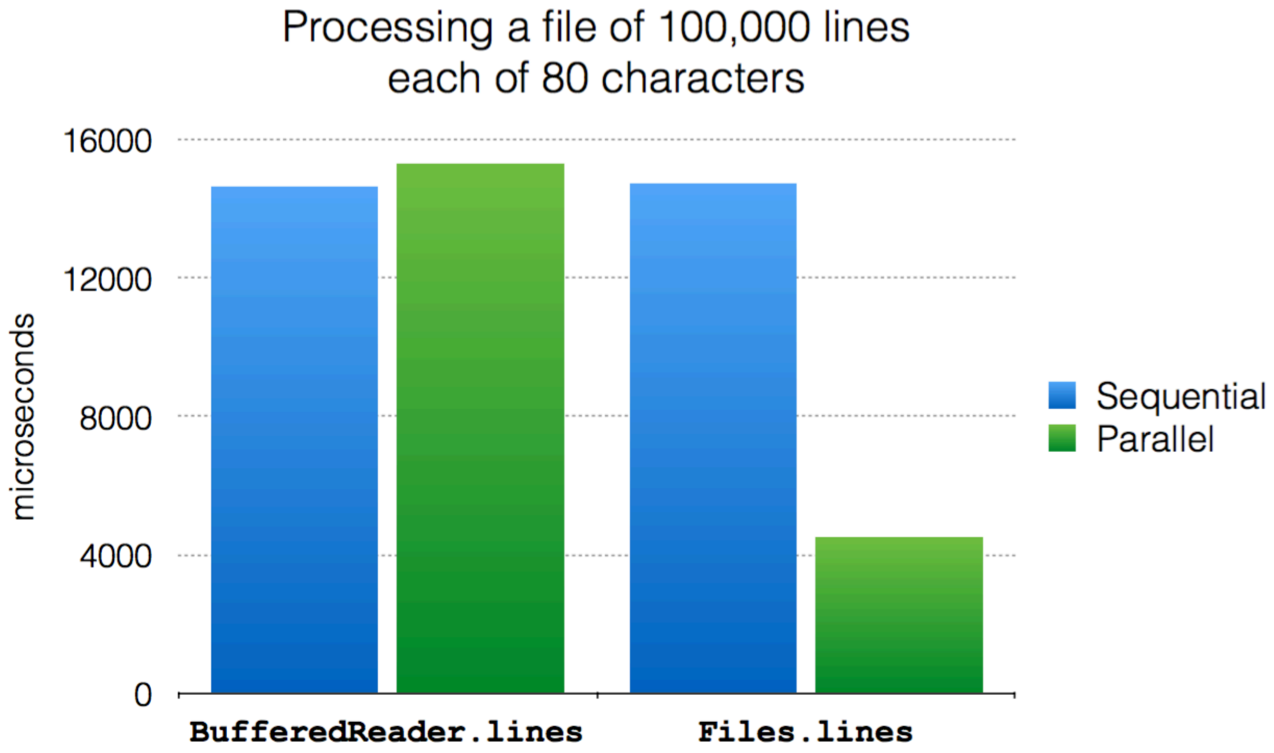
Additional Stream Sources

- `java.net.NetworkInterface`
  - `Stream<InetAddress> inetAddresses()`
  - `Stream<NetworkInterface> subInterfaces()`
  - `Stream<NetworkInterface> networkInterfaces()`
    - `static`
- `java.security.PermissionCollection`
  - `Stream<Permission> elementsAsStream()`

Parallel Support For Files.lines()

- Memory map file for UTF-8, ISO 8859-1, US-ASCII
  - Character sets where line feeds easily identifiable
- Efficient splitting of mapped memory region
- Divides approximately in half
  - To nearest line feed

Parallel Lines Performance



Results produced using **`jmh`** on a MacBook Pro (2012 model)

Stream takeWhile

- `Stream<T> takeWhile(Predicate<? super T> p)`
- Select elements from stream until `Predicate` matches
- Unordered stream needs consideration

```
thermalReader.lines()
  .mapToInt(i -> Integer.parseInt(i))
  .takeWhile(i -> i < 56)
  .forEach(System.out::println);
```

Stream dropWhile

- `Stream<T> dropWhile(Predicate<? super T> p)`
- Ignore elements from stream until `Predicate` matches
- Unordered stream still needs consideration

```
thermalReader.lines()
  .mapToInt(i -> Integer.parseInt(i))
  .dropWhile(i -> i < 56)
  .forEach(System.out::println);
```

Conclusions

Conclusions

- Lambdas provide a simple way to parameterise behaviour
- The Stream API provides a functional style of programming
- Very powerful combination
- Does require developers to think differently
  - Avoid loops, even non-obvious ones!
  - Reductions
- More to come in JDK 9 (and 10)

- Join the Zulu.org community
  - www.zulu.org

Q & A

Simon Ritter

Deputy CTO, Azul Systems

@speakjava | azul.com