Lessons Learnt With Lambdas and Streams in JDK 8

Simon Ritter
Deputy CTO, Azul Systems
azul.com

@speakjava

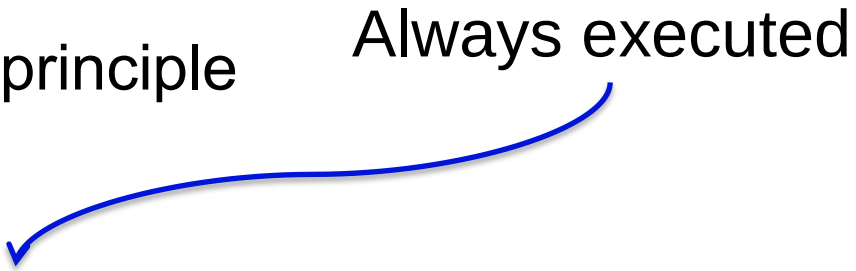
A clever man learns from his mistakes...

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

Lambda Expressions And Delayed Execution

Performance Impact For Logging

- Heisenberg's uncertainty principle




```
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(() -> getSomeStuffToLog());
```

- 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

Still Thinking Imperatively

```
LongAdder newMethodCount = new LongAdder();

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

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

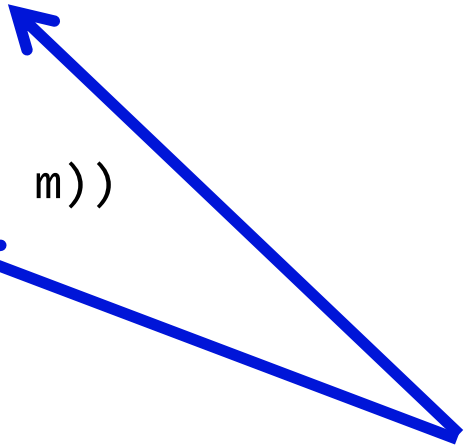
Printing And Counting

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();
```

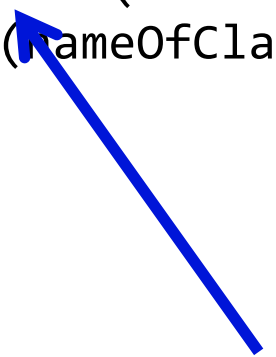


There is still state being modified in the Lambda

Printing And Counting

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

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 = ...

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

Another Simple Problem

- Find the length of the longest line in a file

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

Recursive Approach

```
String findLongestString(String longest, BufferedReader reader) {
  String next = reader.readLine();

  if (next == null)
    return longest;

  if (next.length() > longest.length())
    longest = next;

  return findLongestString(longest, reader);
}
```

Recursion: Solving The Problem

```
String longest = findLongestString("", reader);
```

- No explicit loop, no mutable state, we're all good now, right?
- Unfortunately not:
  - larger data sets will generate an OOM exception
  - Too many stack frames

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

A Better Stream Solution

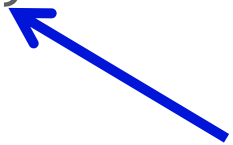
- Use the recursive approach as an acculator for a reduction

```
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 acculator 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)
```

Conclusions

Conclusions

- Lambdas and Stream are a very powerful combination
- Does require developers to think differently
  - Avoid loops, even non-obvious ones!
  - Reductions
- Be careful with parallel streams
- More to come in JDK 9 (and 10)

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

Q & A

Simon Ritter
Deputy CTO, Azul Systems
azul.com

@speakjava