



Lesson 3:

Advanced Lambda and Stream Concepts

Simon Ritter
Java Technology Evangelist

Lesson Agenda

- Understanding and using reductions
- Finite and infinite streams
- Avoiding the use of the `forEach` method
- Using collectors
- Parallel streams (and when not to use them)
- Debugging streams and lambdas
- Course conclusions





Lesson 3-1: Understanding and Using Reductions

A Simple Problem

- Find the length of the longest line in a file

```
Path input = Paths.get("lines.txt");  
  
int longestLineLength = Files.lines(input)  
    .mapToInt(String::length)  
    .max()  
    .getAsInt();
```

Another Simple Problem

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

Naïve Stream Solution

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

- This solves the problem
- 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
- Not functional

Recursive Approach: The Method

```
String findLongestString(String s, int index, List<String> l) {  
    if (index >= l.size())  
        return s;  
  
    if (index == l.size() - 1) {  
        if (s.length() > l.get(index).length())  
            return s;  
        return l.get(index);  
    }  
  
    String s2 = findLongestString(l.get(index), index + 1, l);  
  
    if (s.length() > s2.length())  
        return s;  
    return s2;  
}
```


Recursive Approach: Solving The Problem

```
List<String> lines = new ArrayList<>();  
  
while ((String s = reader.readLine()) != null)  
    lines.add(s);  
  
String longest = findLongestString("", 0, lines);
```

- No explicit loop, no mutable state, so we now have a functional solution
- Unfortunately not a usable one
 - larger data sets will generate an OOM exception

A Better Stream Solution

- The stream API uses the well known filter-map-reduce pattern
- For this problem we do not need to `filter()` or `map()`, just `reduce()`
- Recall the reduce method definition

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

- The key is to find the right accumulator
 - Again, recall 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
 - Without all the stack frames

A Better Stream Solution

- Use the recursive approach as an accumulator for a reduction


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

A Better Stream Solution

- Use the recursive approach as an accumulator for a reduction

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

x in effect maintains state for us, by always holding the longest string found so far



The Simplest Stream Solution

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

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

- `comparingInt()` is a static method on `Comparator`
 - `Comparator<T> comparingInt(ToIntFunction<? extends T> keyExtractor)`

Section 1

Summary

- Reduction take a stream and reduces it to a single value
- The way the reduction works is defined by the accumulator
 - Which is a `BinaryOperator`
 - The accumulator is applied successively to the stream elements
 - The `reduce()` method maintains a partial result state
 - Like a recursive approach, but without the resource overhead
- Requires you to think differently to an imperative, loop based approach

ORACLE®