Java Screams? Oh, you mean Java Streams!

Java Streams for the advanced

Michael Mirwaldt

TOC

- Who is the presentator?
- Which basics about Java Streams are expected?
- What is recommended for Streams?
- What are side effects and how can they be avoided?
- How can a “Stream Monolith” split up into pieces?
- For what are Streams useful and for what aren’t they?
- How can checked exceptions be handled in streams?
- How can stream expressions be debugged?

Who is the presentator?

- Michael Mirwaldt, 37 years old,
- lives in Munich, Germany
- Senior Java backend developer
  for an insurance company
- CS degree from Munich university LMU
- 16 years experience with Java
- Contributions to JMH and JCStress
- has played improvisation theater for 11 years
- Proud uncle of 2 cute nieces
- Github/Twitter: (@)mmirwaldt

Which basics about Java Streams are expected?

- Java 8
  - Java Lambdas
  - Method references
  - Functional interfaces:
    Supplier, Consumer, Function, Predicate
  - Java Stream:
    map, filter, flatMap, collect, reduce

What is recommended for Streams?

- Max. 5 operations per expression
- One operation per line
- Side effects must be avoided
- Foreach operation must rarely be used
- Stream expressions must be easy to understand
- If you need to think a long time about how to write a stream expression for a problem, then don’t use a stream expression
- Split big “Stream monoliths” into several small stream expressions
- In doubt, try out one solution with a stream and one without it and compare them
- For a check, ask your colleague what a stream expression does

What are side effects
and how can they be avoided? (1)

- Access something outside a stream from inside a stream: “**Reading OK but not writing!**”

```
Set<Integer> acceptables = Set.of(1, 2, 3, 5, 6, 7, 9, 11);
Set<Integer> inputs = Set.of(0, 2, 3, 4, 8, 9);

List<Integer> flawed = new ArrayList<>();
inputs.stream()
    .filter(acceptables::contains) // OK
    .forEach(flawed::add); // NO!

List<Integer> better = inputs.stream()
    .filter(acceptables::contains)
    .collect(toList()); // Choose the right terminal operation!
```

What are side effects and how can they be avoided? (2)

- Stateful predicates:

```
List<Integer> numbers = List.of(1, 2, 3, 5, 6, 8, 9);
List<Integer> flawedThirds = numbers.stream()
    .filter(new Predicate<>() {
        int counter = 1;
        public boolean test(Integer value) { return counter++ % 3 == 0; }
    })
    .toList();
List<Integer> betterThirds = numbers.stream()
    .filter(elem -> (numbers.indexOf(elem) + 1) % 3 == 0)
    .toList(); // result : [3, 8]
```

How can a “Stream Monolith” split up into pieces? (1)

- A “Stream Monolith”:

```
List<String> lines = Files.readAllLines(Path.of("rhyme.txt"));
SortedMap<Long, List<String>> top10words = lines.stream()
    .filter(line -> !line.isEmpty())
    .map(line -> line.replaceAll("[\\!|\\.|\\-|\\,]", ""))
    .flatMap(line -> Arrays.stream(line.split("\\s+")))
    .collect(groupingBy(s -> s, counting()))
    .entrySet().stream()
    .sorted((left, right) -> -Long.compare(left.getValue(), right.getValue()))
    .limit(10)
    .collect(
        groupingBy(Map.Entry::getValue, () -> new TreeMap<>(reverseOrder()),
        mapping(Map.Entry::getKey, toList())));
```

How can a “Stream Monolith” split up into pieces? (2)

- First split:

```
List<String> lines = Files.readAllLines(Path.of("rhyme.txt"));
Map<String, Long> frequenciesByWords = lines.stream()
    .filter(line -> !line.isEmpty())
    .map(line -> line.replaceAll("[\\!|\\.|\\-|\\,]", ""))
    .flatMap(line -> Arrays.stream(line.split("\\s+")))
    .collect(groupingBy(s -> s, counting()));
SortedMap<Long, List<String>> wordsByFrequency = a.entrySet().stream()
    .collect(
        groupingBy(Map.Entry::getValue, () -> new TreeMap<>(reverseOrder()),
        mapping(Map.Entry::getKey, toList())));
```

How can a “Stream Monolith” split up into pieces? (3)

- Second split:

```
SortedMap<Long, List<String>> top10words =
    wordsByFrequencies.entrySet().stream()
    .flatMap(entry -> entry.getValue().stream().map(value -> Map.of(entry.getKey(), value)))
    .flatMap(map -> map.entrySet().stream())
    .limit(10)
    .collect(
        groupingBy(Map.Entry::getKey, () -> new TreeMap<>(reverseOrder()),
        mapping(Map.Entry::getValue, toList())));
```

How can a “Stream Monolith” split up into pieces? (4)

- Records (Java 16+) can help:

```
record WordEntry(long frequency, String word) { }
SortedMap<Long, List<String>> top10words = wordsByFrequency.entrySet().stream()
    .flatMap(entry -> entry.getValue().stream()
        .map(value -> new WordEntry(entry.getKey(), value)))
    .limit(10)
    .collect(
        groupingBy(WordEntry::frequency, () -> new TreeMap<>(reverseOrder()),
        mapping(WordEntry::word, toList())));
```

For what are Streams useful and for what aren’t they? (1)

- Streams
  - are expressions but no programs
  - are pipelines but neither iterators nor loops
  - are onedimensional but not multidimensional
  - always deliver **one** result but never more than one
  - only read from one source but rarely change it
  - can be infinite but must be limited
  - only create overhead if they are empty

For what are Streams useful and for what aren’t they? (2)

- Express a query:

```
var names = List.of("Heinz", "Michael", "Brian", "Marc", "Kurt");
var selectedUpperCaseNamesByFirstLetter = names.stream()
    .filter(name -> 'J' <= name.charAt(0)) // range J-Z
    .map(String::toUpperCase)
    .collect(groupingBy(name -> name.substring(0, 1), toList())); // result : {K=[KURT], M=[…]}
```

- Convert a string to CamelCase:

```
String moduleName = "project-process-create-account";
String camelCaseClassName = Arrays.stream(moduleName.split("-"))
    .skip(2)
    .map(name -> name.substring(0, 1).toUpperCase() + name.substring(1))
    .collect(joining()) + "Process"; // result: CreateAccountProcess
```

For what are Streams useful and for what aren’t they? (3)

- Checks with allMatch() (or anyMatch() or noneMatch()):

```
public static int parseAndSum(List<String> numbersAsStrings) {
    if (numbersAsStrings.stream().allMatch(str -> str.matches("-?\\d+"))) {
        return numbersAsStrings.stream()
            .mapToInt(Integer::parseInt)
            .sum();
    } else {
        String nonInt = numbersAsStrings.stream()
            .filter(str -> !str.matches("-?\\d+"))
            .findFirst().get();
        throw new IllegalArgumentException("’ + nonInt + "’ is not an int.");
    }
}
```

For what are Streams useful and for what aren’t they? (4)

- Merge 2 Maps by preferring the minimum value in case of collisions:

```
SortedMap<Integer, Integer> leftMap = new TreeMap<>(Map.of(1, 3, 2, 1, 3, 4));
SortedMap<Integer, Integer> rightMap = new TreeMap<>(Map.of(1, 2, 2, 3, 3, 4));
SortedMap<Integer, Integer> mergedByMin =
    Stream.of(leftMap, rightMap)
    .flatMap(map -> map.entrySet().stream())
    .collect(toMap(Map.Entry::getKey,
                   Map.Entry::getValue,
                   Math::min,
                   TreeMap::new)
     ); // result : {1=2, 2=1, 3=4}
```

For what are Streams useful and for what aren’t they? (5)

- Infinite Streams:

```
var first100Primes = IntStream.iterate(2, i -> i + 1)
    .filter(i -> isPrime(i))
    .limit(100)
    .boxed()
    .toList(); // result : [2, 3, 5, … , 521, 523, 541]
// ---------------------------------------------------------------
public static boolean isPrime(int n) {
    for (int i = 2; i < n; i++) {
        if(n % i == 0) { return false; }
    }
    return 1 < n;
}
```

For what are Streams useful and for what aren’t they? (6)

- Flatten by mapMulti():

```
List<Object> intTree = List.of(1, List.of(2, 3), List.of(List.of(4, 5)));
List<Integer> intList = intTree.stream()
    .mapMultiToInt((node, downStream) -> visit(node, downStream))
    .limit(4)
    .boxed()
    .toList(); // result : [1, 2, 3, 4]
// ---------------------------------------------------------------
public static void visit(Object node, IntConsumer downStream) {
    if (node instanceof Iterable<?> iterable) {
        for (Object e : iterable) { visit(e, downStream); }
    } else if(node instanceof Integer i) { downStream.accept(i); }
}
```

For what are Streams useful and for what aren’t they? (7)

```
List<Integer> ints = List.of(1, 2, 3, 5, 6, 8, 9);
```

```
System.out.println("-".repeat(120));
ints.stream().forEach(i -> {
    System.out.println(i);
    System.out.println("-".repeat(120));
});
```

☹

```
System.out.println("-".repeat(120));
for (Integer i : ints) {
    System.out.println(i);
    System.out.println("-".repeat(120));
}
```

☺

For what are Streams useful and for what aren’t they? (8)

- Pipeline built despite an empty stream:

```
Stream<String> stream = Stream.<Integer>empty()
.filter(i -> i < 3)
.map(Integer::toBinaryString);
```

- Iterator instead of stream:

```
List<Integer> ints = new ArrayList<>(Arrays.asList(1, 2, 3, 5, 6, 8, 9));
ListIterator<Integer> iterator = ints.listIterator();
int i = 1;
while (iterator.hasNext()) {
    iterator.next();
    if(i % 3 == 0) { iterator.remove(); }
    i++;
} // ints : [1, 2, 5, 6, 9]
```

For what are Streams useful and for what aren’t they? (9)

- Transpose a matrix by two nested for-loops:

```
int[][] matrix = new int[][] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
for (int r = 1; r < matrix.length; r++) {
    for (int c = 0; c < r; c++) {
        int temp = matrix[c][r];
        matrix[c][r] = matrix[r][c];
        matrix[r][c] = temp;
    }
}
// matrix :
// [[1, 4, 7],
//  [2, 5, 8],
//  [3, 6, 9]]
```

For what are Streams useful and for what aren’t they? (10)

- Fibonacci **with** a stream:

```
public static long fibonacciByStream(int n) {
    long[] results = IntStream.rangeClosed(3, n)
                              .boxed()
                              .reduce(new long[] {0, 1, 1},
                                      (fib, i) -> {
                                          fib[i % 3] = fib[(i - 2) % 3] + fib[(i - 1) % 3];
                                          return fib;
                                      },
                                      (a, b) -> null);
    return results[n % 3];
}
```

For what are Streams useful and for what aren’t they? (11)

- Fibonacci **without** a stream:

```
public static long fibonacciByLoop(int n) {
    long[] fib = new long[] {0, 1, 1};
    for (int i = 3; i <= n; i++) {
        fib[i % 3] = fib[(i - 2) % 3] + fib[(i - 1) % 3];
    }
    return fib[n % 3];
}
```

How can checked exceptions be handled in streams? (1)

- “Sneaky throw”-Hack:

```
static <T, R> Function<T, R> sneakyThrow(TFunction<T, R> f) {
    return t -> {
        try { return f.apply(t); }
        catch (Exception ex) { return sneaky(ex); }
    };
}
public interface TFunction<T, R> { R apply(T t) throws Exception; }
static <T extends Exception, R> R sneaky(Exception t) throws T { throw (T) t; }
List<URL> urls = Stream.of("http://www.wikipedia.de", "http://www.mozilla.org/")
    .map(sneakyThrow(URL::new))
    .collect(Collectors.toList());
```

How can checked exceptions be handled in streams? (2)

- Convert a checked exception into a RuntimeException:

```
public interface ExceptionFunction<T, R> { R apply(T t) throws Exception; }
static <T, R> Function<T, R> unchecked(ExceptionFunction<T, R> f) {
    return t -> {
        try { return f.apply(t); }
        catch (RuntimeException ex) { throw ex; }
        catch (Exception ex) { throw new RuntimeException(ex); }
    };
}
List<URL> urls = Stream.of("http://www.wikipedia.de", "http://www.mozilla.org/")
    .map(unchecked(URL::new))
    .collect(Collectors.toList());
```

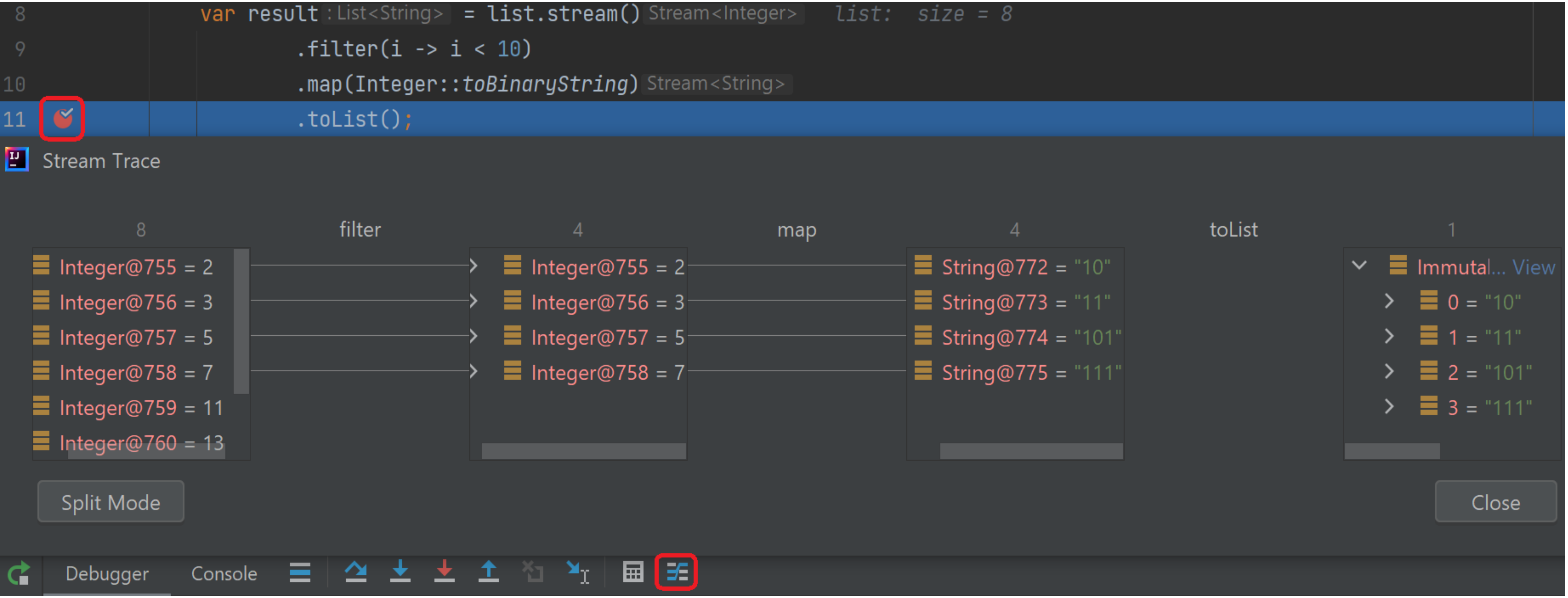
How can stream expressions be debugged? (1)

- Peek-Methode:

```
var list = List.of(2, 3, 5, 7, 11, 13, 17, 19);
var result = list.stream()
    .filter(i -> i < 14)
    .peek(i -> System.out.println("after filter(): " + i))
    .map(Integer::toBinaryString)
    .peek(i -> System.out.println("after map(): " + i))
    .toList();
```

How can stream expressions be debugged? (2)

- IntelliJ IDEA:



```
var result : List<String> = list.stream() Stream<Integer>   list:  size = 8
        .filter(i -> i < 10)
        .map(Integer::toBinaryString) Stream<String>
        .toList();
```

Stream Trace

```
8                    filter   4                    map   4                         toList   1
Integer@755 = 2      ->       Integer@755 = 2      ---   String@772 = "10"                  Immutal... View
Integer@756 = 3      ->       Integer@756 = 3      ---   String@773 = "11"                    0 = "10"
Integer@757 = 5      ->       Integer@757 = 5      ---   String@774 = "101"                   1 = "11"
Integer@758 = 7      ->       Integer@758 = 7      ---   String@775 = "111"                   2 = "101"
Integer@759 = 11                                                                              3 = "111"
Integer@760 = 13
```

Thank you for your attention!

- Any questions?

???

- Slides and examples available at https://github.com/mmirwaldt/JavaStreamsForTheAdvanced