Alles im Fluss?
Java Streams für Fortgeschrittene

Michael Mirwaldt

Was erwartet euch?

- Wer ist der Präsentator?
- Welche Grundkenntnisse werden vorausgesetzt?
- Was kann euch der Präsentator empfehlen?
- Was sind Seiteneffekte und wie können sie vermieden werden?
- Wie kann ein “Stream Monolith” zerlegt werden?
- Wie kann ein Stream-Ausdruck weiterentwickelt werden?
- Was und wie sind Streams?
- Wofür sind Streams geeignet?
- Wofür sind Streams ungeeignet?
- Zusammenfassung

Wer ist der Präsentator?

- Michael Mirwaldt, 37 Jahre alt, in München
- Senior Java Backend Entwickler bei einer Versicherung
- Informatik-Studium an der LMU
- 16 Jahre Erfahrung mit Java
- Github-Projekt: Lazy build streams
- Spielt selber Improvisationstheater seit 11 Jahren
- Stolzer Onkel von süßen 2 Nichten
- Github/Twitter: (@)mmirwaldt

Welche Grundkenntnisse werden vorausgesetzt?

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

Was kann euch der Präsentator empfehlen?

- Max. 5 Operationen pro Ausdruck
- Eine Operation pro Zeile
- Seiteneffekte müssen vermieden werden und foreach sollte selten benutzt werden
- Ausdrücke sollten leicht zu lesen sein, sonst verfehlen sie ihre Wirkung
- Wenn man sehr viel nachdenken muss, wie ein Ausdruck aussehen muss, dann lieber keinen Stream Ausdruck benutzen
- Mehrere kleine Ausdrücke statt einem sehr großen “Stream Monolith” oft besser
- Kleine Ausdrücke lassen sich einfacher verbessern/verändern
- Im Zweifel eine Lösung mit und eine ohne Streams ausprobieren und beide vergleichen
- Kollegen/in fragen, was ein Stream-Ausdruck macht

Was sind Seiteneffekte und wie können sie vermieden werden? (1)

- Zugriff aus dem Stream-Ausdruck nach außen: “**Lesen OK, aber Schreiben nicht!**”

```
1:  Set<Integer> acceptables = Set.of(1, 2, 3, 5, 6, 7, 9, 11);
2:  Set<Integer> inputs = Set.of(0, 2, 3, 4, 8, 9);
3:
4:  List<Integer> flawed = new ArrayList<>();
5:  inputs.stream()
6:      .filter(acceptables::contains) // OK
7:      .forEach(flawed::add); // NO!
8:
9:  List<Integer> better = inputs.stream()
10:     .filter(acceptables::contains)
11:     .collect(toList()); // Choose the right terminal operation!
```

Was sind Seiteneffekte und wie können sie vermieden werden? (2)

- Zustandsbehaftete Prädikate bzw. Lambdas:

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

Wie kann ein “Stream Monolith” zerlegt werden? (1)

- Ein “Stream Monolith”:

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

Wie kann ein “Stream Monolith” zerlegt werden? (2)

- Erste Zerlegung:

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

Wie kann ein “Stream Monolith” zerlegt werden? (3)

- Zweite Zerlegung:

```
1: SortedMap<Long, List<String>> top10 =
2:     wordsByFrequencies.entrySet().stream()
3:     .flatMap(entry -> entry.getValue().stream().map(value -> Map.of(entry.getKey(), value)))
4:     .flatMap(map -> map.entrySet().stream())
5:     .limit(10)
6:     .collect(
7:         groupingBy(Map.Entry::getKey, () -> new TreeMap<>(reverseOrder()),
8:         mapping(Map.Entry::getValue, toList())));
```

Wie kann ein “Stream Monolith” zerlegt werden? (4)

- Records (Java 16+) können helfen:

```
1: record WordEntry(long frequency, String word) { }
2: SortedMap<Long, List<String>> top10 = wordsByFrequency.entrySet().stream()
3:     .flatMap(entry -> entry.getValue().stream()
4:         .map(value -> new WordEntry(entry.getKey(), value)))
5:     .limit(10)
6:     .collect(
7:         groupingBy(WordEntry::frequency, () -> new TreeMap<>(reverseOrder()),
8:         mapping(WordEntry::word, toList())));
```

Wie kann ein Stream-Ausdruck weiterentwickelt werden? (1)

- Anpassungen fallen leichter – **erster** Versuch:

```
1:  SortedMap<Long, List<String>> top10Plus = wordsByFrequency.entrySet().stream()
2:      .map(entry -> Map.of(entry.getKey(), entry.getValue()))
3:      .collect(() -> new TreeMap<>(reverseOrder()),
4:          (result, map) -> result.putAll((10 <= sumSizesOfValues(result)) ? emptyMap() : map),
5:          TreeMap::putAll);
6:  public static long sumSizesOfValues(SortedMap<Long, List<String>> map) {
7:      return map.values().stream().mapToLong(List::size).sum();
8:  }
9:  // {42=[the], 39=[swallowed], 38=[she], 23=[to], 21=[catch], 14=[a, fly, that], 13=[and],
10: // 12=[spider, wiggled]} instead of
11: // {42=[the], 39=[swallowed], 38=[she], 23=[to], 21=[catch], 14=[a, fly, that], 13=[and],
12: // 12=[spider]}
```

Wie kann ein Stream-Ausdruck weiterentwickelt werden? (2)

- Anpassungen fallen leichter – **zweiter** Versuch:

```
1:  SortedMap<Long, List<String>> top10Plus = wordsByFrequency.keySet().stream()
2:      .map(wordsByFrequency::headMap)
3:      .filter(headMap -> (10 <= sumSizesOfValues(headMap))
4:      .map(TreeMap::new)
5:      .findFirst().orElse(new TreeMap<>(wordsByFrequency));
6:  public static long sumSizesOfValues(SortedMap<Long, List<String>> map) {
7:      return map.values().stream().mapToLong(List::size).sum();
8:  }
9:  // {42=[the], 39=[swallowed], 38=[she], 23=[to], 21=[catch], 14=[a, fly, that], 13=[and],
10: // 12=[spider, wiggled]} instead of
11: // {42=[the], 39=[swallowed], 38=[she], 23=[to], 21=[catch], 14=[a, fly, that], 13=[and],
12: // 12=[spider]}
```

Was und wie sind Streams?

- Streams
  - sind Ausdrücke, aber keine Programme
  - sind Pipelines, aber weder Iteratoren noch Schleifen
  - sind eindimensional, aber nicht mehrdimensional
  - liefern immer **ein** Ergebnis, aber nie mehrere
  - lesen nur aus der “Quelle”, aber verändern sie nicht
  - können unendlich sein, aber müssen endlich gemacht werden
  - erzeugen nur Overhead, wenn sie leer bleiben

Wofür sind Streams geeignet? (1)

- Eine Query formulieren:

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

- In CamelCase umwandeln:

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

Wofür sind Streams geeignet? (2)

- Checks mit allMatch() (oder anyMatch() oder noneMatch()):

```
1:  public static int parseAndSum(List<String> numbersAsStrings) {
2:      if (numbersAsStrings.stream().allMatch(str -> str.matches("-?\\d+"))) {
3:          return numbersAsStrings.stream()
4:              .mapToInt(Integer::parseInt)
5:              .sum();
6:      } else {
7:          String nonInt = numbersAsStrings.stream()
8:              .filter(str -> !str.matches("-?\\d+"))
9:              .findFirst().get();
10:         throw new IllegalArgumentException("' + nonInt + "' is not an int.");
11:     }
12: }
```

Wofür sind Streams geeignet? (3)

- 2 Maps mit minimalen Werten durch einen Stream-Ausdruck mergen:

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

- Eine Liste von Namen in Großbuchstaben ausgeben:

```
9: names.stream() // List<String> names
10:     .map(String::toUpperCase)
11:     .forEach(System.out::println);
```

Wofür sind Streams geeignet? (4)

- Unendliche Streams:

```
1:  var first100Primes = IntStream.iterate(2, i -> i + 1)
2:      .filter(i -> isPrime(i))
3:      .limit(100)
4:      .boxed()
5:      .toList(); // result : [2, 3, 5, … , 521, 523, 541]
6:  // ------------------------------------------------------------
7:  public static boolean isPrime(int n) {
8:      for (int i = 2; i < n; i++) {
9:          if(n % i == 0) { return false; }
10:     }
11:     return 1 < n;
12: }
```

Wofür sind Streams geeignet? (5)

- “Mehrdimensionales Flatten” durch mapMulti():

```
1:  List<Object> intTree = List.of(1, List.of(2, 3), List.of(List.of(4, 5)));
2:  List<Integer> intList = intTree.stream()
3:      .mapMultiToInt((node, downStream) -> visit(node, downStream))
4:      .limit(4)
5:      .boxed()
6:      .toList(); // result : [1, 2, 3, 4]
7:  // ------------------------------------------------------------
8:  public static void visit(Object node, IntConsumer downStream) {
9:      if (node instanceof Iterable<?> iterable) {
10:         for (Object e : iterable) { visit(e, downStream); }
11:     } else if(node instanceof Integer i) { downStream.accept(i); }
12: }
```

Wofür sind Streams ungeeignet? (1)

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

2:  System.out.println("-".repeat(120));
3:  ints.stream().forEach(i -> {
4:      System.out.println(i);
5:      System.out.println("-".repeat(120));
6:  });
```

```
7:  System.out.println("-".repeat(120));
8:  for (Integer i : ints) {
9:      System.out.println(i);
10:     System.out.println("-".repeat(120));
11: }
```

Wofür sind Streams ungeeignet? (2)

- Pipeline trotz leerem Stream:

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

- Iterator statt Stream:

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

Wofür sind Streams ungeeignet? (3)

- Matrix durch 2 Schleifen ohne Stream transponieren:

```
1:  int[][] matrix = new int[][] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
2:  for (int r = 1; r < matrix.length; r++) {
3:      for (int c = 0; c < r; c++) {
4:          int temp = matrix[c][r];
5:          matrix[c][r] = matrix[r][c];
6:          matrix[r][c] = temp;
7:      }
8:  }
9:  // matrix :
10: // [[1, 4, 7],
11: //  [2, 5, 8],
12: //  [3, 6, 9]]
```

Wofür sind Streams ungeeignet? (4)

- Fibonacci **mit** Stream:

```
1:  public static long fibonacciByStream(int n) {
2:      long[] results = IntStream.rangeClosed(3, n)
3:                          .boxed()
4:                          .reduce(new long[] {0, 1, 1},
5:                                  (fib, i) -> {
6:                                      fib[i % 3] = fib[(i - 2) % 3] + fib[(i - 1) % 3];
7:                                      return fib;
8:                                  },
9:                                  (a, b) -> null);
10:     return results[n % 3];
11: }
```

Wofür sind Streams ungeeignet? (5)

- Fibonacci **ohne** Stream:

```
1: public static long fibonacciByLoop(int n) {
2:     long[] fib = new long[] {0, 1, 1};
3:     for (int i = 3; i <= n; i++) {
4:         fib[i % 3] = fib[(i - 2) % 3] + fib[(i - 1) % 3];
5:     }
6:     return fib[n % 3];
7: }
```

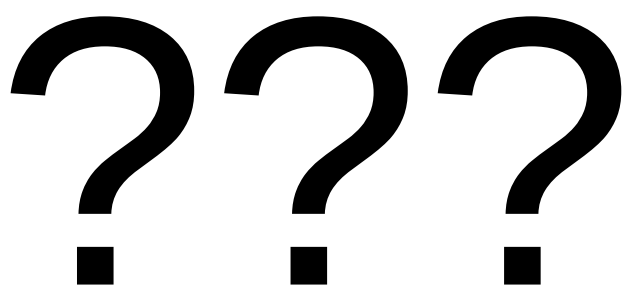
Zusammenfassung

- Max. 5 Operation pro Ausdruck mit je einer Zeile pro Operation
- Stream-Ausdrücke sollten leicht zu lesen sein, sonst verfehlen sie ihre Wirkung
- Seiteneffekte
  - wie das Schreiben in äußere Datenstrukturen von innerhalb des Streams und
  - wie zustandsbehaftete Lambdas für Operationen im Ausdruck

  sollten unbedingt vermieden werden, weil sie oft schwer verständlich sind
- Lange Stream-Ausdrücke können in kurze Ausdrücke für Variablen zerlegt werden
- Stream-Ausdrücke sind eindimensionale Pipelines und liefern **ein** Ergebnis
- **Verwendet Stream-Ausdrücke bitte nur da, wo sie sinnvoll sind und nicht überall!**

Danke für eure Aufmerksamkeit!

- Noch Fragen?



- Beispiele und Folien unter
  https://github.com/mmirwaldt/JavaStreamsForTheAdvanced