

Streams

go with the flow

7. Großübung

Ulm University | Florian Sihler, Raphael Straub und Matthias Tichy | 4. Juli 2024



Software Engineering
Programming Languages



universität
uulm

Hehoo!

Gibt es vorab Wünsche?



Themenübersicht

1. Einfache Streams

2. Reducer

3. Sources

4. Primitive Streams

5. Lazy & Parallel

6. Es gibt noch mehr!

Streams

... verarbeiten eine Folge von Elementen

`squareOfOdds(List.of(1,2,3,7,6))`

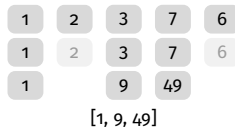
```
List<Integer> squareOfOdds(List<Integer> l) {  
    return l.stream()  
        .filter(n -> n % 2 != 0)  
        .map(n -> n * n)  
        .toList();  
}
```

Erzeuge Stream


Nur ungerade Zahlen

Quadriere jede Zahl

Sammle in Liste



Ein bisschen Lambda

```
List<Integer> squareOfOdds(List<Integer> l) {  
    return l.stream() Predicate<Integer>  
        .filter(n -> n % 2 != 0)  boolean test(Integer t)  
        .map(n -> n * n)  
        .toList(); Function<Integer,Integer>  
}
```

Methoden-Referenzen

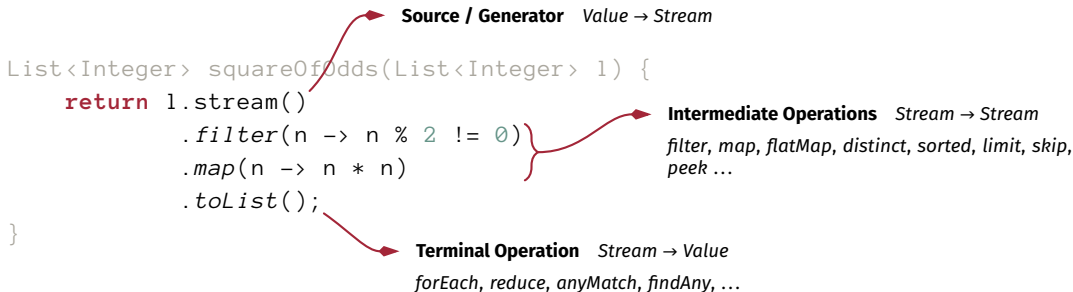
```
class Main {  
    private static int square(int n) {  
        return n * n;  
    }  
  
    List<Integer> squareOfOdds(List<Integer> l) {  
        return l.stream()  
            .filter(n -> n % 2 != 0)  
            .map(Main::square)  
            .toList();  
    }  
}
```

Verwende Funktion als Lambda

Achtung!
Lambdas sind in Java **Objekte** einer *anonymen Klasse*,
die das geforderte funktionale Interface implementiert.



Ein kleiner Stream — Anatomie



Aufgabe 1:

Implementieren Sie die Funktion `int summing(Collection<Integer> c)`, die die Summe aller Elemente in `c` berechnet.

Lösungsvorschläge

Aufgabe 1:

Implementieren Sie die Funktion `int summing(Collection<Integer> c)`, die die Summe aller Elemente in `c` berechnet.

```
int summing(Collection<Integer> c) {  
    return c.stream().reduce(0, (a, b) -> a + b);  
}
```

```
int summing(Collection<Integer> c) {  
    return c.stream().reduce(0, Integer::sum);  
}
```

- Wie könnte eine Lösung für `Collection<Long>` aussehen?

```
long summing(Collection<Long> c) {  
    return c.stream().reduce(0L, Long::sum);  
}
```


Aufgabe 2: Die teuersten Items

```
public record Valuable(String name, int price) { }
```

Aufgabe 2:

Implementieren Sie die Funktion `List<Valuable> topN(Collection<Valuable> c, int n)`, die die `n` teuersten Elemente zurückliefert.

Tipp:

Hierfür kann ein `Comparator<Valuable>` nützlich sein, der zwei `Valuables` nach ihrem Preis vergleicht. Für zwei Zahlen kann man hierbei einfach die Differenz bilden: $(a, b) \rightarrow b - a$.

Lösungsvorschlag

```
List<Valuable> topN(Collection<Valuable> c, int n) {  
    return c.stream().sorted((a, b) -> b.price() - a.price())  
        .limit(n).toList();  
}
```

```
List<Valuable> topN(Collection<Valuable> c, int n) {  
    return c.stream().sorted(Comparator.comparingInt(Valuable::price).reversed())  
        .limit(n).toList();  
}
```

Themenübersicht

1. Einfache Streams

2. Reducer

3. Sources

4. Primitive Streams

5. Lazy & Parallel

6. Es gibt noch mehr!

Reducer

- `Optional<T> reduce(BinaryOperator<T> accumulator)`, bzw.
`T reduce(T identity, BinaryOperator<T> accumulator)`

Erlaubt es die Stream-Elemente zu einem einzigen Wert zu reduzieren („fold“).

- `<R, A> R collect(Collector<? super T, A, R> collector)`

Ein *collector* sammelt die Stream-Elemente in einer Datenstruktur. Man *kann* auch eigene bauen, im Allgemeinen hilft aber die `Collectors` Hilfs-Klasse. Ein *collector* ist auch nur ein (abstrahierter) *reducer*.

- Gängige Reducer sind direkt vordefiniert:
toList, count, findAny, findFirst, max, min, ...

Aufgabe 3: Suche nach Items

```
public record Valuable(String name, int price) { }
```

Aufgabe 3:

Implementieren Sie `Optional<Valuable> get(Collection<Valuable> c, String name)`, die ein beliebiges Element mit dem gesuchten `name` zurückliefert, sofern dieses existiert.

Lösungsvorschlag

Aufgabe 3:

Implementieren Sie `Optional<Valuable> get(Collection<Valuable> c, String name)`, die ein beliebiges Element mit dem gesuchten `name` zurückliefert, sofern dieses existiert.

```
Optional<Valuable> get(Collection<Valuable> c, String name) {  
    return c.stream().filter(v -> v.name().equals(name)).findAny();  
}
```

```
Optional<Valuable> get(Collection<Valuable> c, String name) {  
    return c.stream().filter(v -> v.name().equals(name))  
        .reduce((a, b) -> b);  
}
```

Themenübersicht

1. Einfache Streams

2. Reducer

3. Sources

4. Primitive Streams

5. Lazy & Parallel

6. Es gibt noch mehr!

Sources — Quellen

- `Collection::stream` `List.of(1,2,3).stream()`
Erzeugt einen Stream aus einer Collection.
- `Stream::of` `Stream.of(1,2,3)`
Erzeugt einen Stream aus gegebenen Elementen.
- `Stream::iterate` `Stream.iterate(1, n -> n + 1)`
Erzeugt einen Stream, der mit einem Start-Element beginnt und mit einer Funktion das nächste Element berechnet.
- `Arrays::stream` `Arrays.stream(new int[] {1,2,3})`
Erzeugt einen Stream aus einem Array.
- So gibt es einige Möglichkeiten, Streams zu erzeugen.

Aufgabe 4: Calculating Squares

Aufgabe 4:

Implementieren Sie `List<Integer> squares(int n)`, die von 1 an die ersten `n` Quadratzahlen liefert.

Beispiel:

```
squares(4) // → [1, 4, 9, 16]
```

Lösungsvorschlag

Aufgabe 4:

Implementieren Sie `List<Integer> squares(int n)`, die von 1 an die ersten `n` Quadratzahlen liefert.

```
List<Integer> squares(int n) {  
    return Stream.iterate(1, i -> i + 1).map(i -> i * i)  
        .limit(n).toList();  
}
```

Themenübersicht

1. Einfache Streams

2. Reducer

3. Sources

4. Primitive Streams

5. Lazy & Parallel

6. Es gibt noch mehr!

Streams primitiver Datentypen

- Generics akzeptieren in Java **keine** primitiven Datentypen.
- Dafür gibt es mit z.B. `IntStream` und `DoubleStream` spezialisierte Streams für *einige*.
- Funktionen wie `mapToInt`, `mapToDouble`, erlauben die Umwandlung in primitive Streams.

```
int summing(Collection<Integer> c) {  
    return c.stream().mapToInt(i -> i).sum();  
}
```

```
List<Integer> squares(int n) {  
    return IntStream.range(1, n + 1).map(i -> i * i).boxed().toList();  
}
```

Aufgabe 5: Getting A Domain

Aufgabe 5:

Implementieren Sie `int delta(Collection<Integer> n)`, die die Differenz des größten und kleinsten Elements in einer Collection zurückliefert.

Beispiel:

```
delta(List.of(1, 2, 0, 4, 5)) // → 5 - 0 = 5
```

Lösungsvorschlag

```
int delta(Collection<Integer> n) {  
    var max = n.stream().max(Integer::compare).orElseThrow();  
    var min = n.stream().min(Integer::compare).orElseThrow();  
    return max - min;  
}
```

```
int delta(Collection<Integer> n) {  
    var s = n.stream().mapToInt(i -> i).summaryStatistics();  
    return s.getMax() - s.getMin();  
}
```

Themenübersicht

1. Einfache Streams

2. Reducer

3. Sources

4. Primitive Streams

5. Lazy & Parallel

6. Es gibt noch mehr!

Parallele Streams

- Wenn unsere Operationen keinen Zustand brauchen und sich nicht gegenseitig beeinflussen, können wir Streams parallelisieren.
- Wir bieten Java dies durch `parallel()` an.

```
void myAmazingStream() {  
    List.of(1, 2, 3, 4, 5).stream().parallel()  
        .forEach(System.out::println);  
}
```


Lazy

- Stream-Operationen sind lazy
- Wenn die Implementation herausfindet, dass Operationen nicht notwendig sind, werden sie nicht ausgeführt.

```
List<Integer> myLazyStream() {  
    return List.of(1, 2, 3, 4, 5).stream()  
        .map(i -> { System.out.print(i); return i; })  
        .limit(2).toList();  
}
```

Themenübersicht

1. Einfache Streams

2. Reducer

3. Sources

4. Primitive Streams

5. Lazy & Parallel

6. Es gibt noch mehr!

Es gibt noch so viel mehr...

- Intermediate Funktionen wie *flatMap* erlauben es, verschachtelte Strukturen zu plätten
- *concat* fügt Streams zusammen
- Andere Funktionen können Streams bewusst erzeugen um z.B. die lazy-ness auszunutzen (wie `Files::lines`).
- ...