

Objektorientierte Programmierung

Hochschule Bochum

WS 18/19

Dr.-Ing. Darius Malysiak

III. Vertiefung Java 1-8

Streams:

Schreiben Sie eine Methode zur Rückgabe des kleinsten Wertes in einem Array

```
ArrayList<Integer> numbers = new  
ArrayList<Integer>();  
numbers.add(-10);  
numbers.add(-456);  
numbers.add(-1);
```

III. Vertiefung Java 1-8

Streams:

Schreiben Sie eine Methode zur Rückgabe des kleinsten Wertes in einem Array

```
ArrayList<Integer> numbers = new  
ArrayList<Integer>();  
numbers.add(-10);  
numbers.add(-456);  
numbers.add(-1);  
  
Integer candidate = numbers.get(0);  
for(Integer i : numbers) {  
    if(i.compareTo(candidate) < 0) {  
        candidate = i;  
    }  
}  
System.out.println(candidate);
```

III. Vertiefung Java 1-8

Streams:

Schreiben Sie eine Methode zur Rückgabe des kleinsten Wertes in einem Array

```
int[] array = {-10, -456, -1};  
Arrays.stream(array).min().ifPresent(System.out::println);
```

III. Vertiefung Java 1-8

Streams:

Schreiben Sie eine Methode zur Rückgabe des kleinsten Wertes in einem Array

```
int[] array = {-10, -456, -1};  
Arrays.stream(array).min().ifPresent(System.out::println);
```



```
Arrays.asList(-10, -456, -1).stream().mapToInt(i->i).min()  
.ifPresent(System.out::println);
```

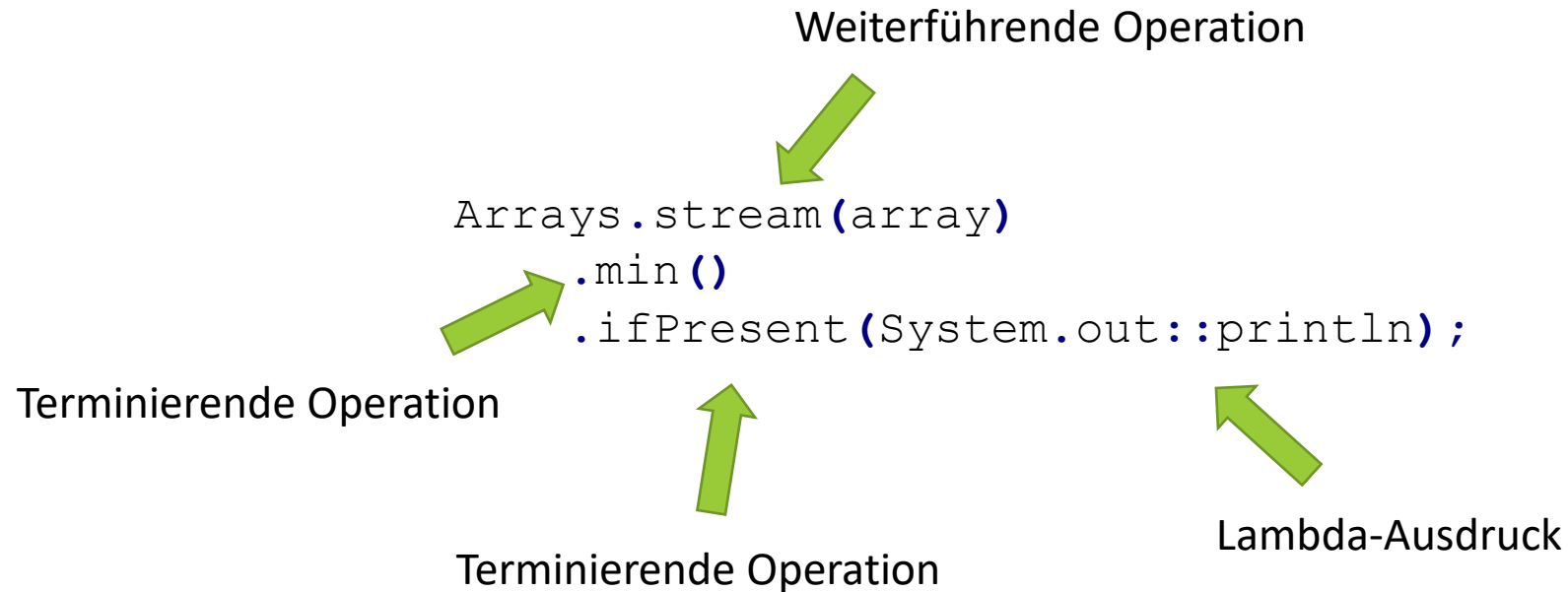
III. Vertiefung Java 1-8

Streams:

- Ein Stream ist eine Sequenz von Elementen auf welchen Operationen durchgeführt werden können.
- Eine Stream-Operation kann ‚weiterführend‘ oder ‚terminierend‘ sein, hierbei wird ein neuer Stream zurückgegeben bzw. ein Stream beendet.
- Terminierende Operation geben ‚null‘ oder ein nicht-stream Objekt zurück.
- Häufig erlauben Stream-Operation die Nutzung von Lambda Parametern.
- Java unterstützt Streams in direkter Art nur für ‚Set‘ und ‚List‘ Objekte.
- Weiterführende Operationen z.B.: ‚filter‘, ‚map‘.
Terminierende Operationen z.B.: ‚forEach‘, ‚ifPresent‘.
- Es existieren generische sowie spezialisierte Streams (z.B. für int, double).

III. Vertiefung Java 1-8

Streams:



III. Vertiefung Java 1-8

Streams:

Warum Streams? Was ist mit Iterator?

III. Vertiefung Java 1-8

Streams:

```
long tic = System.currentTimeMillis();
int size = 50000000;
int[] a = getRandomArray(size);
int candidate = a[0];

for(int i=0; i < size; i++)
    if(a[i] > candidate) candidate = a[i];

long toc = System.currentTimeMillis();
System.out.println("Time: "+(toc-tic));
```

~650 ms

III. Vertiefung Java 1-8

Streams:

```
long tic = System.currentTimeMillis();  
int size = 50000000;  
int[] a = getRandomArray(size);  
  
Arrays.stream(array).max();  
  
long toc = System.currentTimeMillis();  
System.out.println("Time: " + (toc - tic));
```



~750 ms

~15% langsamer

III. Vertiefung Java 1-8

Streams: Best Practices I

```
IntStream.range(1, 200).forEach(i -> System.out.println(i));
```



```
for(int i=1; i < 200; i++)  
    System.out.println(i);
```

III. Vertiefung Java 1-8





Streams: Best Practices II

```
IntStream.range(1, 200)
    .mapToObj(i -> "n"+i)
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .forEach(System.out::println);
```

Was passiert hier?

III. Vertiefung Java 1-8

Streams: Best Practices II

		<code>IntStream.range(1, 200)</code>
Umwandlung in Object Stream		<code>.mapToObj(i -> "n"+i)</code>
Transformation mit L-Funktion		<code>.map(s -> s.substring(1))</code>
Umwandlung in Integer Stream		<code>.mapToInt(Integer::parseInt)</code>
Anwenden eine L-Funktion auf jedes Element		<code>.forEach(System.out::println);</code>

Was passiert hier?

III. Vertiefung Java 1-8

Streams: Best Practices II

```
IntStream.range(1, 200)
    .mapToObj(i -> "n"+i)
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .forEach(System.out::println);
```

Generische und spezialisierte Streams können ineinander überführt werden!

III. Vertiefung Java 1-8

Streams: Best Practices III

```
Stream.of("a", "b", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    });
```

Filtert alle Elemente
für welche die L-Funktion
,true' ist.

Ausgabe: „“ ← ???

III. Vertiefung Java 1-8

Streams: Best Practices III

```
Stream.of("a", "b", "c")  
    .filter(s -> {  
        System.out.println("filter: " + s);  
        return true;  
    }).forEach( s -> {} );
```

Filtiert alle Elemente
für welche die L-Funktion
'true' ist.

Ausgabe: „a b c“

„Leere“ L-Funktion.

Weiterführende Operationen werden nur ausgeführt wenn eine terminierende Funktion existiert -> „Pipeline Prinzip“

3 Iterationen in diesem Beispiel.

III. Vertiefung Java 1-8

Streams: Best Practices IV

Stream Umwandlung

Terminierende Operation:
Sobald L-Funktion ‚true‘ liefert
wird der Stream beendet

```
Stream.of("a", "c", "c")
    .map(s ->
        { return s.toUpperCase();
        })
    .anyMatch( s ->
        {
            System.out.println(s);
            return s.startsWith("C");
        });
```

Ausgabe: „A C“

Nur zwei Iterationen!

III. Vertiefung Java 1-8

Streams: Best Practices V

```
Stream.of("a", "c", "d", "c", "c")  
    .map(s -> { return s.toUpperCase(); })  
    .filter(s -> { return s.startsWith("C"); })  
    .forEach(System.out::println);
```



```
Stream.of("a", "c", "d", "c", "c")  
    .filter(s -> { return s.startsWith("c"); })  
    .map(s -> { return s.toUpperCase(); })  
    .forEach(System.out::println);
```

Welche Variante ist effizienter?

III. Vertiefung Java 1-8

Streams: Best Practices VI

```
Stream.of("a", "c", "d", "c", "c")  
    .map(s -> { return s.toUpperCase(); })  
    .filter(s -> { return s.startsWith("C"); })  
    .forEach(System.out::println);
```

5 Aufrufe
5 Aufrufe
3 Aufrufe



Die Reihenfolge ist wichtig im Hinblick auf Performance;
besonders bei komplexen Operationen.

```
Stream.of("a", "c", "d", "c", "c")  
    .filter(s -> { return s.startsWith("c"); })  
    .map(s -> { return s.toUpperCase(); })  
    .forEach(System.out::println);
```

5 Aufrufe
3 Aufrufe
3 Aufrufe

III. Vertiefung Java 1-8

Streams: Best Practices V

```
Stream<String> stream = Stream.of("a", "b", "c")  
    .filter(s -> s.startsWith("b"));
```

```
stream.anyMatch(s -> true); stream.anyMatch(s -> true);
```



Streams können nur einmal verwendet (,durchlaufen') werden.

III. Vertiefung Java 1-8

Streams: Best Practices VI

```
List<?> l = Stream.of("a", "c", "d", "c", "c")  
    .map(s -> { return s.toUpperCase(); })  
    .filter(s -> { return s.startsWith("C"); })  
    .collect(Collectors.toList());
```



Terminierende Operation;
Sammelt Elemente in Container,
gibt Container zurück.



L-Funktion

III. Vertiefung Java 1-8


Streams: Best Practices VII

```
public class Dummy {  
    public String name;  
    public Dummy(String name) {this.name=name;}  
}  
  
public class ContainerC {  
    public String id;  
    public List<Dummy> dummies = new ArrayList<>();  
    ContainerC(String id) { this.id = id; }  
}
```

III. Vertiefung Java 1-8

Streams: Best Practices VII

```
public static void main(String[] args) {  
    List<ContainerC> l = IntStream.range(1, 4)  
        .mapToObj(i -> new ContainerC("Container"+i))  
        .collect(Collectors.toList());  
  
    l.forEach( c -> c.dummies = IntStream.range(1, 3)  
        .mapToObj(i -> new Dummy("Dummy"+i))  
        .collect(Collectors.toList()) );  
  
    l.stream().flatMap( c -> {System.out.println(c.id);  
                                return c.dummies.stream();}  
        ).forEach(d -> System.out.println(d.name)); }
```



Weiterführende Operation:

L-Funktion muss Object in Stream wandeln.

III. Vertiefung Java 1-8

Streams: Best Practices VII

```
public static void main(String[] args) {  
    List<ContainerC> l = IntStream.range(1, 4)  
        .mapToObj(i -> new ContainerC("Container"+i))  
        .collect(Collectors.toList());  
  
    l.forEach( c -> c.dummies = IntStream.range(1, 3)  
        .mapToObj(i -> new Dummy("Dummy"+i))  
        .collect(Collectors.toList()) );  
  
    l.stream().flatMap( c -> {System.out.println(c.id);  
        return c.dummies.stream();}  
        .forEach(d -> System.out.println(d.name)); }
```



Container1
Dummy1
Dummy2
Container2
Dummy1
Dummy2
Container3
Dummy1
Dummy2


III. Vertiefung Java 1-8

Streams: Best Practices VIII

```
List<ContainerC> l = IntStream.range(1, 4)
    .mapToObj(i -> new ContainerC("Container"+(i%2)))
    .collect(Collectors.toList());

l.forEach( c -> c.dummies = IntStream.range(1, 3)
    .mapToObj(i -> new Dummy("Dummy"+i))
    .collect(Collectors.toList()) );

l.stream()
    .reduce((c1, c2) -> c1.id.equals(c2.id) ? c1 : c2)
    .ifPresent(c -> System.out.println(c.id));
```



Terminierende Operation:

L-Funktion muss 2 Objekte des Streams auf ein Objekt abbilden.

Insgesamt existieren 3 Varianten dieser Operation.

III. Vertiefung Java 1-8

Streams: Best Practices VIII

```
List<ContainerC> l = IntStream.range(1, 4)
    .mapToObj(i -> new ContainerC("Container"+(i%2)))
    .collect(Collectors.toList());
```

```
l.forEach( c -> c.dummies = IntStream.range(1, 3)
    .mapToObj(i -> new Dummy("Dummy"+i))
    .collect(Collectors.toList()) );
```

```
l.stream()
    .reduce((c1, c2) -> c1.id.equals(c2.id) ? c1 : c2)
    .ifPresent(c -> System.out.println(c.id));
```



Container1

III. Vertiefung Java 1-8

Streams: Best Practices IX

```
public static void main(String[] args) {  
    List<ContainerC> l = IntStream.range(1, 4)  
        .mapToObj(i -> new ContainerC("Container"+i))  
        .collect(Collectors.toList());  
  
    l.forEach( c -> c.dummies = IntStream.range(1, 3)  
        .mapToObj(i -> new Dummy("Dummy"+i))  
        .collect(Collectors.toList()) );  
  
    l.parallelStream().flatMap( c -> {System.out.println(c.id);  
        return c.dummies.stream();}  
        .forEach(d -> System.out.println(d.name)); }  
    );  
}
```



Container2
Container3
Dummy1
Dummy2
Container1
Dummy1
Dummy2
Dummy1
Dummy2



Elemente des Parallel-Streams werden von
Threads eines Thread-Pools bearbeitet.