Java Lambda Streams Cheat Sheet

forEach(Consumer<T>)

Führt eine Aktion für jedes Element aus.

–

```
a b c
```

```
Stream.of("a", "b", "c")
    .forEach(XPrint::print);
```

forEach(Consumer<T>) mit stream()

Führt eine Aktion für jedes Element aus und verwendet `stream()`.

–

```
b a c
```

```
List<String> list = List
    .of("b", "a", "c");
list.stream().forEach(XPrint::print);
```

forEach(Consumer<T>) mit parallelStream()

Führt eine Aktion parallel für jedes Element aus.

–

```
b a c
```

```
List<String> list = List
    .of("b", "a", "c");
list.parallelStream()
     .forEach(XPrint::print);
```

max(Comparator<T>)

Findet das größte Element gemäß dem Comparator.

–

```
Optional[ccc]
```

```
Optional<String> max = Stream
    .of("a", "bb", "ccc")
    .max(Comparator
         .comparing(String::length));

System.out.print(max);
```

min(Comparator<T>)

Findet das kleinste Element gemäß dem Comparator.

–

```
Optional[a]
```

```
Optional<String> min = Stream
    .of("a", "bb", "ccc")
    .min(Comparator
         .comparing(String::length));

System.out.print(min);
```

peek(Consumer<T>)

Wendet eine Funktion auf jedes Element an, ohne den Stream zu verändern.

–

```
 ...: a ...: b ...: c
```

```
List<String> result = Stream
    .of("a", "b", "c")
    .peek(s ->
       System.out.print(" ...: " + s))
    .collect(Collectors.toList());
```

sorted()

Sortiert die Elemente im Stream.

–

```
a b c
```

```
List<String> sorted = Stream
    .of("c", "a", "b")
    .sorted()
    .collect(Collectors.toList());

sorted.forEach(XPrint::print);
```

sorted(Comparator<T>)

Sortiert die Elemente im Stream mit einem Comparator.

–

```
c b a
```

```
List<String> sorted = Stream
    .of("c", "a", "b")
   .sorted(Comparator.reverseOrder())
   .collect(Collectors.toList());

sorted.forEach(XPrint::print);
```

forEachOrdered()

Gibt jedes Element des Streams in der Reihenfolge aus, in der es im Stream erzeugt wurde.

–

```
b a c
```

```
List<String> list = List
     .of("b", "a", "c");
list.parallelStream()
     .forEachOrdered(XPrint::print);
```

map(Function<T, R>)

Wendet eine Funktion auf jedes Element an und gibt einen Stream der Ergebnisse zurück.

–

```
1 4 9 16
```

```
List<Integer> squares = Arrays
    .asList(1, 2, 3, 4).stream()
    .map(n -> n * n)
    .collect(Collectors.toList());

squares.forEach(XPrint::print);
```

map(Function<T, R>)

Wandelt alle Buchstaben in Großbuchstaben um.

–

```
C A B
```

```
Arrays.asList("c", "a", "b")
    .stream().map(String::toUpperCase)
    .forEach(XPrint::print);
```

mapToInt(ToIntFunction<T>)

Wandelt die Elemente in primitive int-Werte um.

–

```
1 2 3
```

```
int[] lengths = Stream
    .of("a", "ab", "abc")
    .mapToInt(String::length)
    .toArray();

Arrays.stream(lengths)
.forEach(XPrint::print);
```

flatMap(Function<T, Stream<R>>)

Wandelt jedes Element in einen Stream um und flacht das Ergebnis zu einem einzigen Stream ab.

–

```
a b c d
```

```
List<String> flatList = Arrays
  .asList(
     Arrays.asList("a", "b"),
     Arrays.asList("c", "d")).stream()
       .flatMap(List::stream)
       .collect(Collectors.toList());

flatList.forEach(XPrint::print);
```

mapToDouble(ToDoubleFunction<T>)

Wandelt die Elemente in primitive double-Werte um.

–

```
1.0 2.0 3.0
```

```
double[] lengths = Stream
    .of("a", "ab", "abc")
    .mapToDouble(String::length)
    .toArray();

Arrays.stream(lengths)
.forEach(XPrint::print);
```

mapToLong(ToLongFunction<T>)

Wandelt die Elemente in primitive long-Werte um.

–

```
1 2 3
```

```
long[] lengths = Stream
    .of("a", "ab", "abc")
    .mapToLong(String::length)
    .toArray();

Arrays.stream(lengths)
.forEach(XPrint::print);
```

range(startInclusive, endExclusive)

Erzeugt einen Stream mit einer Zahlenfolge.

–

```
1 2 3 4 5 6 7 8 9
```

```
IntStream stream = IntStream
    .range(1, 10);

stream.forEach(XPrint::print);
```

rangeClosed(startInclusive, endInclusive)

Erzeugt einen Stream mit einer Zahlenfolge.

–

```
1 2 3 4 5 6 7 8 9 10
```

```
IntStream stream = IntStream
    .rangeClosed(1, 10);

stream.forEach(XPrint::print);
```

generate(Supplier<? extends T> s)

Erzeugt einen Stream mit Zufallszahlen.

–

```
0.9356835104779333 ❯
0.7417061950743408
```

```
Stream<Double> stream = Stream
  .generate(Math::random).limit(2);

stream.forEach(XPrint::print);
```

generate(Supplier<? extends T> s)

Erzeugt einen Stream mit Zufallszahlen zwischen 1 und 6.

–

```
6 3 3 6 1 6 1 5
```

```
Stream<Integer> stream = Stream
  .generate(() ->
     (int) (Math.random() * 6) + 1)
  .limit(8);

stream.forEach(XPrint::print);
```

Files.lines(path)

Erzeugt einen Stream mit Zeilen aus einer Textdatei.

–

```
Name,Vorname,Note
Müller,Anna,1
Schmidt,Peter,2
```

```
Path path = Paths.get("../../src/main/❯
resources/noten.csv");

try (final Stream<String> stream =
     Files.lines(path);) {
   stream.limit(3)
         .forEach(System.out::println);
} catch (final IOException e) {
    e.printStackTrace();
}
```

filter(Predicate<T>)

Filtert Elemente basierend auf einer Bedingung.

–

```
Bob
```

```
List<String> list = Arrays
    .asList("Alice", "Bob", "Charlie")
    .stream()
    .filter(name -> name.length() <= 3)
    .collect(Collectors.toList());

list.forEach(XPrint::print);
```

distinct()

Gibt einen Stream mit eindeutigen Elementen zurück.

–

```
1 2 3
```

```
List<Integer> list = Arrays
    .asList(1, 2, 2, 3, 3, 3).stream()
    .distinct()
    .collect(Collectors.toList());

list.forEach(XPrint::print);
```

limit(long maxSize)

Begrenzt die Anzahl der Elemente im Stream.

–

```
1 2 3
```

```
List<Integer> list = Stream
    .of(1, 2, 3, 4, 5)
    .limit(3)
    .collect(Collectors.toList());

list.forEach(XPrint::print);
```

skip(long n)

Überspringt die ersten `n` Elemente im Stream.

–

```
4 5
```

```
List<Integer> list = Stream
    .of(1, 2, 3, 4, 5)
    .skip(3)
    .collect(Collectors.toList());

list.forEach(XPrint::print);
```

count()

Zählt die Ergebnismenge.

–

```
2
```

```
long c = Stream
    .of(1, 2, 3, 4, 5)
    .skip(3)
    .count();
System.out.print(c);
```

reduce(BinaryOperator<T>)

Reduziert die Elemente zu einem einzigen Ergebnis.

–

```
10
```

```
int sum = Arrays
    .asList(1, 2, 3, 4).stream()
    .reduce(0, (a, b) -> a + b);

System.out.print(sum);
```

reduce(T identity, BinaryOperator<T>)

Wie oben, aber mit einem Startwert (`identity`).

–

```
24
```

```
int product = Arrays
    .asList(1, 2, 3, 4).stream()
    .reduce(1, (a, b) -> a * b);

System.out.print(product);
```

reduce(T identity, BiFunction<T, U, T>, BinaryOperator<T>)

Reduziert mit einem Startwert und zwei Funktionen.

–

```
10
```

```
int sum = Arrays
    .asList(1, 2, 3, 4).stream()
    .reduce(0, Integer::sum);

System.out.print(sum);
```

collect(Collectors.toList())

Sammelt die Stream-Elemente in eine Liste.

–

```
a b c
```

```
List<String> list = Stream
    .of("a", "b", "c")
    .collect(Collectors.toList());

list.forEach(XPrint::print);
```

collect(Collectors.toSet())

Sammelt die Stream-Elemente in ein Set.

–

```
a b c
```

```
Set<String> set = Stream
    .of("a", "b", "c")
    .collect(Collectors.toSet());

set.forEach(XPrint::print);
```

collect(Collectors.toMap())

Sammelt die Stream-Elemente in eine Map.

–

```
bb -> 2
a -> 1
bbc -> 3
```

```
Map<String, Integer> map = Stream
    .of("a", "bb", "bbc")
    .collect(Collectors.
        toMap(s -> s, String::length));

map.keySet().forEach(k ->
    System.out.println(k + " -> " +
        map.get(k)));
```

joining()

Verbindet die Elemente zu einem String.

–

```
a, b, c
```

```
String joined = Stream
    .of("a", "b", "c")
    .collect(Collectors.joining(", "));

System.out.print(joined);
```

allMatch()

Prüft, ob alle Elemente des Streams mit der angegebenen Bedingung übereinstimmen.

–

```
true
```

```
Stream<String> stream = Stream
    .of("c", "a", "b");
boolean result = stream
    .allMatch(s -> s.matches("[a-c]"));

System.out.print(result);
```

anyMatch()

Prüft, ob mindestens ein Element des Streams mit der angegebenen Bedingung übereinstimmt.

–

```
true
```

```
Stream<String> stream = Stream
    .of("c", "a", "b");
boolean result = stream
    .anyMatch(s -> s.equals("a"));

System.out.print(result);
```

dropWhile()

Gibt einen neuen Stream zurück, bei dem die führenden Elemente entfernt werden, solange die Bedingung wahr ist. Sobald ein Element die Bedingung nicht erfüllt, werden die restlichen Elemente behalten.

–

```
a b
```

```
Stream<String> stream = Stream
    .of("c", "a", "b");
Stream<String> stream2 = stream
    .dropWhile(s -> s.equals("c"));

stream2.forEach(XPrint::print);
```

findAny()

Gibt ein beliebiges Element aus dem Stream zurück. In parallelen Streams kann das Ergebnis nicht vorhersehbar sein.

–

```
c
```

```
Stream<String> stream = Stream
    .of("c", "a", "b");
stream.findAny()
    .ifPresent(XPrint::print);
```

findAny()

Prüft, ob kein Element des Streams mit der angegebenen Bedingung übereinstimmt.

–

```
true
```

```
Stream<String> stream = Stream
    .of("c", "a", "b");
boolean result = stream
    .noneMatch(s -> s.equals("d"));
System.out.print(result);
```