

STREAMS

Streams vereinfachen u. A. das Arbeiten mit Collections und Arrays.

Anstatt, dass mit Hilfe von Schleifen und Bedingungen der **Lösungs-Algorithmus** beschrieben wird, wird mit Streams die **Lösung** beschrieben.

Beispiel: Ermittle, wie viele Zahlen in einem Array kleiner als 10 sind

(klassische Lösung)

```
int [] a = {1, 2, 5, 12, 9, 11, 6, 88};

int counter = 0;
for (int n:a) {
    if (n < 10) {
        counter++;
    }
}

System.out.println("kleiner als 10 sind " + counter + " Elemente");
```

(mit Streams)

```
int [] a = {1, 2, 5, 12, 9, 11, 6, 88};

long counter = Arrays.stream(a)
    .filter(n -> n<10)
    .count();

System.out.println("kleiner als 10 sind " + counter + " Elemente");
```

Im Einzelnen:

-) Arrays.stream(a) .. erzeugt einen Stream aus dem Array a
-) .filter (IntPredicate p) .. **intermediate Operation** -> liefert Stream (zur Weiterverarbeitung)
 - IntPredicate : „Functional interface“ mit der Methode test(T t) , die bestimmt, nach welchen Kriterien ein Wert zu überprüfen ist
 - Lambda-Ausdruck
-) .count() .. **terminal Operation** -> liefert ein (long) Ergebnis (danach wird der Stream gelöscht)

Generell:

Erzeugen → intermediate Operations (bearbeiten) -> terminal Operation

Beispiel2:

Ermittle die Summe der drei größten durch 3 und durch 7 teilbaren Zahlen der Liste numbers :

```
// durch 3 und durch 7 teilbaren Zahlen sind: 21 (2x), 63 (3x), 147 (1x), 189 (1x),  
// die Summe der drei größten Zahlen davon ist: 189 + 147 + 63 = 399  
  
List<Integer> numbers = Arrays.asList(63, 63, 21, 21, 3, 7, 189, 14, 63, 147, 6, 99);  
int sum = numbers.stream()  
    .filter(n->n%3==0 && n%7==0)  
    .sorted((x,y) -> y-x)  
    .limit(3)  
    .mapToInt(i->i.intValue()) //alternative: (i->i)  
    .sum();  
  
System.out.println(sum);
```

Im Einzelnen:

-) numbers.stream .. jede Collection hat eine Methode .stream()
-) .filter (IntPredicate p) .. s.o.
-) .sorted(Comparator ...) → Interface-Methode compare(int o1, int o2) -> Lambda
-) limit(long n) ... beschränke auf n Elemente
-) mapToInt(i->i.intValue()) .. im Stream sind Objekte -> sum() fkt. nicht
-) sum() -> terminale Operation, liefert Summe aller (verbliebenen) Elemente des Streams

Unterscheidung :

- Allgemeine Streams ; oft durch Generics typisiert (<String>, <...>) ...
- **IntStream**, **LongStream**, **DoubleStream** (haben spezielle Eigenschaften (operations))

Es ist auch möglich, eine **Instanz** von einem Stream anzulegen, die man verwenden kann, so lange noch keine terminale Operation ausgeführt wurde. Nach dem Ausführen einer terminalen Operation wird der Stream „ungültig“.

```
Stream <String> strS = ....;
```

0a) Streams ausgeben (Vorschau terminale Operation)

```
...  
streamName.forEach(x -> System.out.println(x));
```

0b) Methodenreferenzen

You can use a *"method reference"* for expressions like this:

```
numbers.forEach(x -> System.out.println(x));
```

Here, you don't actually *need* the name `x` in order to invoke `println` for each of the elements. That's where the method reference is helpful - the `::` operator denotes you will be invoking the `println` method with a parameter, which name you don't specify explicitly:

```
numbers.forEach(System.out::println);
```

1) Erzeugen von Streams

- aus einer Collection (List, Set, Collection)

```
List<String> li = Arrays.asList("Karl", "Franz", "Kurt");  
Stream<String> streamList = li.stream();
```

```
Set<String> ts = new TreeSet<>(li);  
Stream<String> streamSet = ts.stream();
```

- aus einem Array

```
String [] names = {"Karl", "Franz", "Kurt"};  
Stream<String> fromArray = Arrays.stream(names);
```

```
int [] a = {1, 2, 5, 12, 9, 11, 6, 88};  
IntStream fromIntArray = Arrays.stream(a);
```

- StreamOf()

```
//einfache Aufzählungen  
Stream s1 = Stream.of(1,2,3,4,5);  
  
Stream<String> s2 = Stream.of("a", "bb", "ccc");  
  
//aus Array  
Double [] doubles = {1.2, 2.2, 3.3};  
Stream s3 = Stream.of(doubles);
```

- Ein einzelner Wert

```
Stream<String> singleS = Stream.of("Hallo");
```

- Zusammenhängende Serie von Zahlen (int, long)

```
IntStream ints = IntStream.range(10,15); //zweite Grenze exklusiv
```

```
LongStream longS = LongStream.rangeClosed(5,10); // zweite Grenze inklusiv
```

- Zufallszahlen (Random - Objekt)

```
Random r = new Random();

DoubleStream rndS = r.doubles(); // Zufallszahlen (0 .. 0.99999999)
DoubleStream rndS2 = r.doubles(5); // 5 Zufallszahlen (0 .. 0.99999999)
```

```
Random r = new Random();
IntStream rndInts = r.ints();
    //unendlich viele Zufallszahlen (gesamter Wertebereich)
IntStream rndInts2 = r.ints(6);
    // liefert IntStream mit 6 Zufallszahlen (gesamter Wertebereich)
```

```
Random r = new Random();
IntStream rndS = r.ints(6,1,11); // IntStream mit 6 Zufallszahlen (1 .. 10)
```

- Zeichen eines Strings

```
String str = "Hallo Welt";
IntStream charS = str.chars();
```

(es gibt keinen CharStream → IntStream)

- Stream.iterate (seed , function) ..

seed : 1. Element,
function : beschreibt, wie die folgenden Elemente erzeugt werden
 !!Achtung : limit(..) notwendig, weil sonst unendlich

```
Stream<Integer> sIter = Stream.iterate(1,n->n+3).limit(10);
```

- `Stream.generate (Supplier ) ..`

```
Stream s7 = Stream.generate(()->Math.random()).limit(5);
s7.forEach(System.out::println);
```

oder eine selbst geschriebene Methode ...

```
public class Main {
    public static void main(String[] args) {
        Stream.generate(Main::next)
            .limit(5)
            .forEach(System.out::println);
    }

    static int i=0;
    private static int next(){
        i++;
        return i;
    }
}
```

oder Erzeugen ein und desselben Elements ...

```
public class Main {
    public static void main(String[] args) {
        IntStream.generate(() -> 0)
            .limit(5)
            .forEach(System.out::println);
    }
}
```

- Dateien lesen

```
Path path = Paths.get("res/file.txt");
try {
    Stream<String> streamOfStrings = Files.lines(path);
    Stream<String> streamWithCharset = Files.lines(path,
        Charset.forName("UTF-8"));
} catch (IOException e) {
    e.printStackTrace();
}
```