Streams

In Java 8 wurden mit dem Interface java.util.stream.Stream<T> mächtige Möglichkeiten zur Durchführung von Operationen auf Arrays und Listen eingeführt.

Streams des Interface java.util.stream.Stream<T> , nicht zu verwechseln mit den Ein- und Ausgabe-Streams des Packages java.io , stellen Ströme von Referenzen dar, die es erlauben, verkettete Operationen auf diesen Referenzen nacheinander oder parallel auszuführen.

Wichtig: Die Daten, die durch die Referenz repräsentiert werden, werden durch den Stream **nicht** verändert.

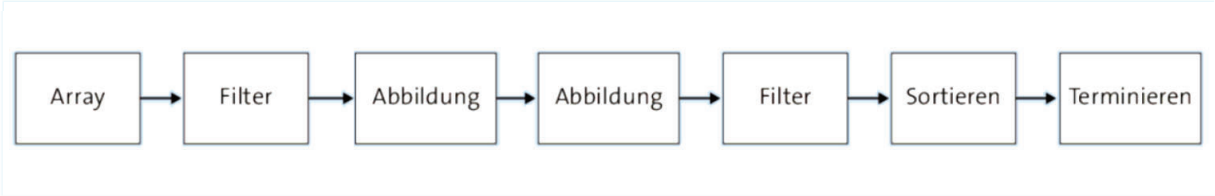
Das Interface Steam und die von ihm abgeleiteten Interfaces stellen lediglich eine Vielzahl von Methoden bereit, die in drei Hauptkategorien eingeteilt werden und meist Lambda Ausdrücke als Argumente übergeben bekommen:

- **Erzeugende** Operationen (create operations) erzeugen einen Stream (einmal zu Beginn)
- **Intermediäre** Operationen (intermediate operations) liefern wiederum einen Stream, der weiterverarbeitet werden kann (z.B. filter(), map(), distinct(), sorted(), etc.)
  (beliebig viele nach dem Öffnen)
- **Terminale** Operationen (terminal operations) führen ihrerseits Operationen auf den Referenzen des Streams aus (forEach(), reduce(), toArray(), etc.). Sie können einen Wert liefern und beenden den Strom. Ist ein Strom einmal geschlossen, so können keine weiteren Operationen auf ihm ausgeführt werden (einmal am Ende)

In folgendem Code-Ausschnitt wird ein Stream verwendet. Überlege dir für jeden Schritt wie der ursprüngliche Datensatz verändert wird und welche Werte am Schluss ausgegeben werden. Die Lösung ist im QR-Code abgebildet.

```
Object[] words = { " ", '3', null, "2", 1, "" };
Arrays.stream( words )                          // ein Stream wird erzeugt
      .filter( Objects::nonNull )               // die null-Objekte werden gefiltert
      .map( Objects::toString )                 // in String-Objekte gemapped
      .map( String::trim )                      // diese werden getrimmt
      .filter( s -> ! s.isEmpty() )             // nicht-leere Strings werden gefiltert
      .map( Integer::parseInt )                 // in int-Werte gemapped
      .sorted()                                 // sortiert
      .forEach( System.out::println );
```

Abbildung 21: Beispiel Streams



Im nächsten Beispiel soll die Länge von Strings in einer Liste ausgegeben werden. Zuerst wird die gewohnte Methode mithilfe einer for-each Schleife dargestellt. Darunter die Alternative mithilfe eines Streams.

```
//Standard mit for-each-Schleife
List<String> list = Arrays.asList("Fabian", "1516", "Baumgartner");
for(String s : list){
	System.out.println(s.length());
}
//mit Stream und anonymer Klasse
list.stream().forEach(new Consumer<String>(){
	@Override
	public void accept(String s){
		System.out.println(s.length());
	}
});
```

Abbildung 22: Ausgabe der Länge

Ausgabe:
7
4
11

Die Methode stream() aus dem Interface java.util.Collection erstellt einen Stream aus der Liste. Die Stream-Methode forEach() nimmt ein Objekt vom Typ Consumer entgegen, das nur eine Methode besitzt, nämlich accept(String item). Diese Methode wird für jedes Element des Streams aufgerufen.

Bei Betrachtung des anonymen Consumers wird deutlich, dass dieser ein Interface mit nur einer Methode ist, die ausprogrammiert werden muss. Hier kann ein Lambda verwendet werden.
Aufgabe: Verkürze den oberen Code mithilfe eines Lambda-Ausdruckes.

Möchte man nur den Inhalt der einzelnen Strings ausgeben, so geht es nochmal kürzer indem man Methodenreferenzen verwendet:

Einige Methoden des Stream Interfaces geben selbst wieder einen Stream zurück. So ist es möglich, Elemente in Streams zu filtern oder zu modifizieren.
Hier werden z.B. mit der Methode collect() die Werte des Integer-Streams wieder in eine Liste [6, 11] umgewandelt

```
List<Integer> laenge = list.stream()
                           .filter( s->s.length() >4)
                           .map(String::length)
                           .collect(Collectors.toList());
```

Abbildung 23: Verkettung von Methoden

Streams erzeugen

Streams können aus Arrays, Listen, anderen Collections sowie aus Einzelobjekten mittels sog. StreamBuilder erzeugt werden. Je nach verwendeter Methode kann das Ergebnis jedoch unterschiedlich ausfallen.

Variante 1: Erzeugen aus Arrays:

```
Arrays.stream(myArray)
```

Die Methode stream liefert einen Stream. Dies zeigt das folgende Beispiel anhand eines Arrays mit primitiven int -Werten:

```
int[] zahlen = {1,2,3,4,5};
Arrays.stream(zahlen)
       .forEach( z-> System.out.print(n+“ “);
```

Abbildung 24: Erzeugen von Streams aus einem Array

Ausgabe: 1 2 3 4 5

Es wird ein Stream erzeugt, nämlich ein **IntStream** aus den einzelnen primitiven Werten des Arrays. Im Gegensatz zum Interface Stream besitzt ein IntStream mit sum(), average(), count() usw. Methoden zur Weiterverarbeitung primitiver int-Werte.

Variante 2: Erzeugen aus Listen und Sets

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(32);
stack.push(1024);
stack.push(8);
stack.push(127);
stack.stream().sorted().forEach(n -> System.out.print(n + ","));
System.out.println();
for (int i : stack) {
    System.out.print(i + ", ");
}
```

Abbildung 25: Stream aus Stack

Ausgabe: 8, 32, 127, 1024
32, 1024, 8, 127

Beachte:

- Es wird aus einem Stack, einer Erweiterung von java.util.AbstractList , durch die Methode stream() ein Stream erzeugt, der dann sortiert und dessen Werte schließlich auf die Konsole ausgegeben werden.
- die Sortierung des Streams hat auf dem eigentlichen Stack keinen Einfluss, da die ursprüngliche Reihenfolge ausgegeben wird.

Die Erzeugung und Abarbeitung eines Streams hat **keinerlei** Einfluss auf die zugrunde liegende Datenstruktur!

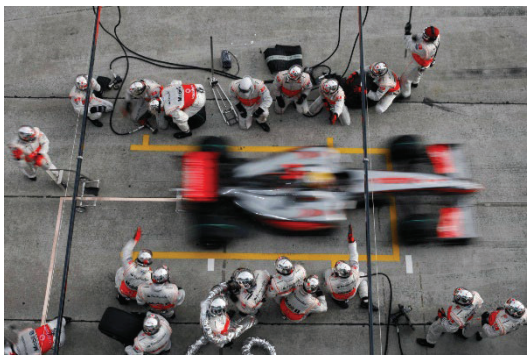
Variante 3: Erzeugung aus Einzel-Elementen

Die statische Methode of() des Interface Stream erzeugt hier einen Stream aus sieben Strings. Er wird dann, von vorne beginnend, auf 3 Elemente beschnitten und schließlich ausgegeben.

```
Stream .of("Ene", "mene", "muh", "und", "raus", "bist", "du")
       .limit(3)
       .forEach(System.out::println);
```

Abbildung 26: Erzeugen aus Einzel-Objekten

Methoden auf Streams anwenden



Die Hauptaufgabe eines Streams besteht darin, Aktionen auf Daten eines Streams auszuführen. Alle möglichen Aktionen sind in der Dokumentation von Streams dargestellt (Link zur Doku). Als Resultat einer solchen „intermediären" Operation entsteht wieder ein Stream, der weiterverarbeitet werden kann. Auf diese Art kann in einer solchen Pipeline ein ursprüngliches Array oder eine Liste schrittweise immer genauer spezifiziert, gefiltert oder angepasst werden.

Die am häufigsten genutzten Intermediären Operationen sind dabei:

- **filter(predicate)**
  Die filter-Methode benötigt wie bereits besprochen ein Predicate, auf welches ein Element untersucht wird. Sollte diese Untersuchung „false" liefern, wird das Element entfernt. Beachte: Hier kann ein Lambda verwendet werden
- **sorted()**
  Mithilfe dieses Operators werden die Elemente eines Streams sortiert. **Achtung:** diese Methode ohne Übergabeparameter ist nur möglich, wenn die Elemente das Interface Compareable implementiert haben. Ansonsten muss ein Comparator übergeben werden.
- **distinct()**
  Mithilfe dieser Methode kann ein Stream auf Duplikate untersucht werden. Mehrmals vorkommende Elemente werden dabei entfernt.
- **map(funcition)**
  Die map-Methode führt Methoden auf die Elemente des Streams aus. So kann beispielsweise ein Wert mit einer Zahl multipliziert, oder Strings mit toUpperCase in Caps-Lock geschrieben werden.

```
List<Integer> intList = List.of(15,20,48,63,49,27,56,32,9);
intList.stream()  .filter( element -> (element%2==0));
                  .…
```

Abbildung 27: Beispiel intermediäre Operation

Einen Stream beenden

Ein Stream muss zwingend beendet werden, da ansonsten auch keine intermediären Operationen ausgeführt werden. Diese Aktion schließt daraufhin den Stream. Gängige Methoden sind beispielsweise:

- **allMatch(predicate) → boolean**
  Diese Methode überprüft alle Elemente des Streams hinsichtlich der übergebenen Eigenschaft und liefert true zurück, wenn **alle** diese erfüllen. Ansonsten false.
- **anyMatch(predicate) → boolean**
  Diese Methode überprüft die Elemente des Streams hinsichtlich der übergebenen Eigenschaft und liefert true zurück, wenn **eines davon** dieses erfüllt. Ansonsten false.
- **forEach() → void**
  Mit dieser Methode wird ein Consumer übergeben, welcher auf alle Elemente angewendet wird. Mit system.out.println() können beispielsweise alle Elemente geprintet werden.
- **toArray() → Array des Stream-Typs**
  Alle Elemente des Streams werden in ein Array gespeichert und zurückgegeben.
- **count() → Ganzzahl**
  Hier wird die Anzahl an Elementen, die im Stream existieren zurückgegeben.

Beispiel:

```
public static void main(String[] args)    {

    List<String> stringList = new ArrayList<String>();
    stringList.add("one");
    stringList.add("two");
    stringList.add("three");
    stringList.add("four");
    stringList.add("five");

    Stream<String> stream = stringList.stream();
    long coutElements = stream
            .map(value -> value.toUpperCase())
            .count();
    System.out.println(coutElements);

}
```

Abbildung 28: Beispiel für terminale Operation