Infozettel: Stream Reduktion

Matthias Eurich – 2017-04-04

Table of Contents

Einführung

In den vorherigen Kapiteln wurden die Grundlagen zur Verwendung von `Stream` näher beleuchtet. Werte wurden bisher auf der Standardausgabe ausgegeben. In diesem Kapitel werden verschiedene Möglichkeiten vorgestellt, einen Rückgabewert aus einem Datenstrom mit den Methoden `collect()` und `reduce()` zu erhalten, um die Verarbeitung eines Ausdrucks abzuschließen.

`collect()`

Mit der Methode `collect()` können die Elemente des Datenstroms gesammelt und z.B. als Liste zurückgegeben werden. Als Parameter nimmt `collect()` einen `collector` vom Typ `Collector` entgegen. Die API bietet mit der Klasse `Collectors` eine Implementierung des Interfaces `Collector` an, die die gängigsten Operationen zum *Einsammeln* von Elementen beinhaltet.

In der JShell sieht man schnell alle von `Collectors` angebotene Methoden.

```
jshell> Collectors.
averagingDouble(        averagingInt(           averagingLong(
class                   collectingAndThen(      counting()
filtering(              flatMapping(            groupingBy(
groupingByConcurrent(   joining(                mapping(
maxBy(                  minBy(                  partitioningBy(
reducing(               summarizingDouble(      summarizingInt(
summarizingLong(        summingDouble(          summingInt(
summingLong(            toCollection(           toConcurrentMap(
toList()                toMap(                  toSet()
```

Nachfolgend werden einige der Methoden vorgestellt.

`toList()`

Mit `Collectors.toList()` werden die Elemente des Datenstroms als Liste zurückgegeben. Im Nachfolgenden Beispiel wird eine Liste von Namen im Datenstrom zuerst mit `map()` modifiziert und anschließend als neue Liste zurückgegeben.

```
jshell> Arrays.asList("Kim", "Liam", "Laura")
$1 ==> [Kim, Liam, Laura]

jshell> $1.stream().map(String::toUpperCase).collect(Collectors.toList())
$2 ==> [KIM, LIAM, LAURA]
```

`toCollection()`

Mit Hilfe der Methode `toCollection()` können beliebige Arten von *Collections* erzeugt werden. Als Parameter muss ein Objekt vom Zieltyp angegeben werden. Im Nachfolgenden Beispiel werden die Elemente in ein `TreeSet` überführt.

```
jshell> Arrays.asList("Kim", "Liam", "Laura")

$1.stream().map(String::toUpperCase).collect(Collectors.toCollection(TreeSet::new)
)
$3 ==> [KIM, LAURA, LIAM]
```

`groupingBy()`

Mit der Methode `groupingBy()` lassen sich Maps vom Typ `Map` erzeugen. Als Parameter nimmt `groupingBy()` eine `Function` entgegen, die bestimmt, was *Key* und was *Value* der Map ist. Im Nachfolgenden Beispiel wird aus einer Liste von Namen eine

Map erzeugt. Der *Key* ist der erste Buchstabe des Namens. Das Ergebnis ist eine `Map` vom Typ `Map<String, List<String>>`.

Beispiel:

```
jshell> $1.stream().map(String::toUpperCase).collect(Collectors.groupingBy(n ->
n.substring(0, 1)))
$5 ==> {K=[KIM], L=[LIAM, LAURA]}
```

joining()

Die Methode `joining()` erlaubt das *Verbinden* (engl. join) der Elemente des Datenstroms zu einer Zeichenkette vom Typ `String`. Sollen die Elemente durch ein Zeichen von einander getrennt werden, kann dies als optionaler Parameter angegeben werden.

Beispiel:

```
jshell> $1.stream().map(String::toUpperCase).collect(Collectors.joining())
$6 ==> "KIMLIAMLAURA"

jshell> $1.stream().map(String::toUpperCase).collect(Collectors.joining(","))
$7 ==> "KIM,LIAM,LAURA"
```

`reduce()`

Mit der Methode `reduce()` lassen sich die Elemente eines Datenstroms zu einem Ergebniswert *verdichten*. Das geschieht häufig durch mathematische Operationen. Als Parameter nimmt `reduce()` eine `identitiy` vom generischen Typ `T` und einen `accumulator` vom Typ `BinaryOperator` entgegen. Der Rückgabewert dieser Methode ist `T`.

```
reduce(T identity, BinaryOperator<T> accumulator)
```

Mit der `identity` wird eine Basis übergeben, auf die die Funktion `accumulator` zusammen mit dem ersten Element des Datenstroms angewendet wird. Anschließend wird die Funktion auf das Ergebnis und den nächsten Wert des Datenstroms angewendet. Dies geschieht so lange, bis der Stream kein nächstes Element mehr enthält

Beispiel:

```
List<Integer> ints = Arrays.asList(1, 2, 3, 4, 5);

Integer summe = ints.stream().reduce(0, (a, b) -> a + b);

// kuerzer und lesbarer mit Methodenreferenz auf die Methode sum() der Klasse
Integer
Integer summeMR = ints.stream().reduce(0, Integer::sum);
```

Hilfsmethoden

`sum()`, `min()`, `max()`

Um sich die redundante Implementierung häufig genutzter, mathematischer, Anwendungsfälle zu sparen, bietet die Java API diese Implementierungen bereits an. Im Interface `Stream` finden sich die Methoden `sum()`, `min()` und `max()`. In den numerischen Datenstrom-Typen wie z.B. `IntStream` gibt es weitere Methoden, wie z.B. `average()`.

`findFirst()`, `findAny()`

Die Methode `findFirst()` gibt das erste Element des Datenstroms zurück, wenn die Elemente des Datenstroms eine geordnete Reihenfolge haben. Andernfalls kann ein zufälliges Element zurückgegeben werden. Der Rückgabewert von `findFirst()` ist vom

Typ `Optional`. `Optional` ist ein Container-Objekt, das den eigentlichen Ergebniswert verpackt.

Beispiel:
Im nachfolden Beispiel wird eine Liste von Namen gefiltert und durch den Aufruf von `findFirst()` das erste Element zurückgegeben.

```
jshell> $1.stream().filter(n -> n.startsWith("L")).findFirst()
$9 ==> Optional[Liam]
```

Mit `findAny()` bekommt man irgendein Element des Datenstroms geliefert.

To Be Continued

Im nächsten Infozettel wird das Container-Objekt `Optional` im Detail betrachtet.

Last updated 2017-05-08 13:46:50 CEST