Infozettel: Stream Grundlagen

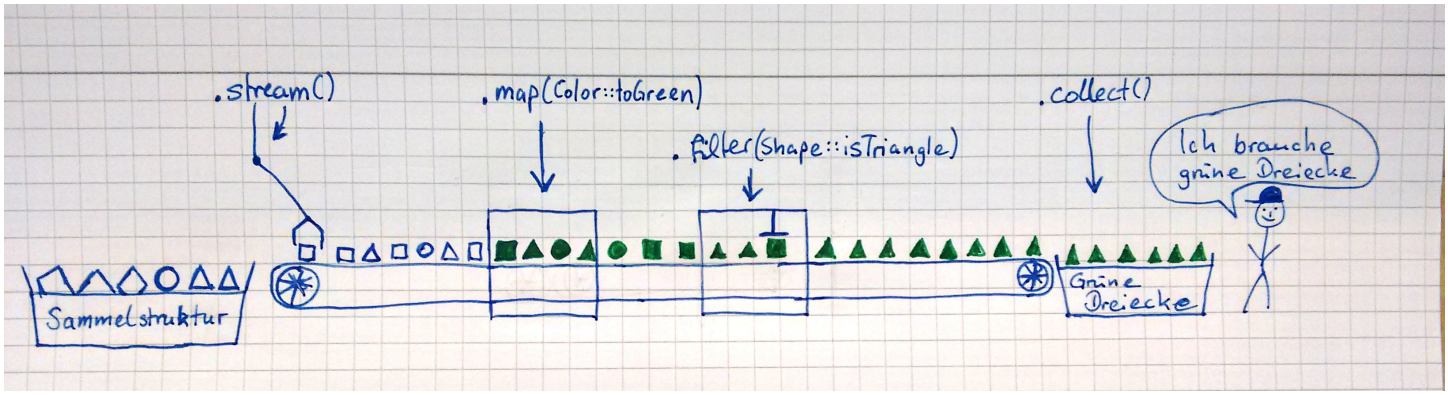
Matthias Eurich – 2017-03-21

Table of Contents

Einführung

Mit Java 1.8 wurden die Interfaces `Stream`, `IntStream`, `LongStream` und `DoubleStream` veröffentlicht. Mit Implementierungen dieser Interfaces ist es möglich verkettete Operationen auf Daten einer Sammelstruktur, z.B. einer `Collection`, sequenziell oder parallel, in Form eines Datenstroms (engl. stream) auszuführen.

Als Datenstrom können Sie sich eine Fertigungsstraße in der Industrie vorstellen, bei der eine Folge von Produkten, durch verschiedene, aufeinanderfolgende Operationen, bearbeitet wird.



Die Interfaces bieten eine Vielzahl von Methoden, für die Verarbeitung von Daten. Die Methoden lassen sich in zwei Kategorien einteilen:

1. Intermediäre Operationen (intermediate operations)
2. Terminale Operationen (terminal operations)

Intermediäre Operationen (intermediär = zwischenliegend) liefern immer einen Rückgabewert vom Typ `Stream`, der im Datenstrom weiterverarbeitet werden kann. *Terminale Operationen* (terminal = beendend) können einen Rückgabewert haben oder vom Typ `void` sein, beenden aber gleichzeitig den Datenstrom. Wird ein Datenstrom beendet, können keine weiteren Operationen auf ihm ausgeführt werden.

Im nachfolgenden Beispiel wird eine Liste mit Namen modifiziert und anschließend gefiltert. Ähnlich wie es mit den Formen im obrigen Bild geschehen ist.

Datenstrom beginnen

Durch Aufruf der Methode `stream()` lässt sich der Datenstrom beginnen, also die Daten auf das Band legen.

Beispiel:

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

jshell> $1.stream()
$2 ==> java.util.stream.ReferencePipeline$Head@59494225
```

In der JShell sehen wir, dass der Aufruf der Methode `stream()` ein Objekt vom Typ `ReferencePipeline`, das im Paket `java.util.stream` liegt, zurück gibt. Die Klasse `ReferencePipeline` implementiert das Interface `Stream`. Wir befinden uns nun also im Datenstrom.

Arbeiten mit dem Datenstrom

Schauen wir in der JShell nach, welche Methoden uns ein Objekt vom Typ `RefernecePipeline` anbietet:

```
jshell> $1.stream().
allMatch(          anyMatch(          close()            collect(
count()            distinct()         dropWhile(         equals(
filter(            findAny()          findFirst()        flatMap(
flatMapToDouble(   flatMapToInt(      flatMapToLong(     forEach(
forEachOrdered(    getClass()         hashCode()         isParallel()
iterator()         limit(             map(               mapToDouble(
mapToInt(          mapToLong(         max(               min(
noneMatch(         notify()           notifyAll()        onClose(
parallel()         peek(              reduce(            sequential()
skip(              sorted(            spliterator()      takeWhile(
toArray(           toString()         unordered()        wait(
```

Unter den Methoden sind einige, deren Funktion man auf Anhieb erkennt, z.B. `forEach()`, `filter()` oder `count()`. Andere Methoden, wie z.B. `map()` sehen Sie vielleicht zum ersten mal. Keine Sorge, wir bringen bald Licht ins Dunkel.

Wir haben den Datenstrom durch den Aufruf von `stream()` geöffnet und wollen jetzt mit der Verarbeitung der Daten beginnen. Da die Namen unserer Liste sowohl mit Groß- als auch mit Kleinbuchstaben beginnen, soll dies für die nachfolgende Prüfung vereinheitlicht werden. Mit der Methode `map()` kann eine Transformation, in unserem Fall `toUpperCase()`, auf jedes Element des Datenstroms angewendet (engl. *map*) werden.

```
jshell> $1.stream().map(String::toUpperCase)
$3 ==> java.util.stream.ReferencePipeline$3@5702b3b1
```

Der Rückgabewert ist ebenfalls ein Objekt vom Typ `ReferencePipeline`. Wir befinden uns also noch im Datenstrom. Im nächsten Schritt sollen die Daten gefiltert werden. Mit der Methode `filter()` wird geprüft, welche Namen mit dem Buchstaben **L** beginnen.

```
jshell> $1.stream().map(String::toUpperCase).filter(n -> n.startsWith("L"))
$4 ==> java.util.stream.ReferencePipeline$2@31a5c39e
```

Der Rückgabewert von `filter()` ist wieder ein Objekt vom Typ `ReferencePipeline`. Wir befinden uns also noch immer im Datenstrom.

In konventionellem Java-Code ist das `if` der Filter.

Datenstrom beenden

Mit der Methode `count()` erfahren wir, wie viele Elemente unseres Datenstromes die Bedingung von `filter()` erfüllt haben.

```
jshell> $1.stream().map(String::toUpperCase).filter(n -> n.startsWith("L")).count()
$5 ==> 2
```

Die Methode `count()` ist eine *Terminale Operation* und gibt einen Rückgabewert vom Typ `long` zurück. In unserem Fall eine **2**. Da `long` keine Implementierung von `Stream` ist, kann an dieser Stelle keine weitere Operation verkettet werden. Der Datenstrom wurde durch die Reduzierung auf einen Ergebniswert abgeschlossen.

Das *Neue* an Streams ist, dass die Evaluierung, im Gegensatz zu normalen Anweisungen, erst durch den Aufruf einer *Terminalen Operation* gestartet wird. Erst durch Abschließen des Streams durch den Aufruf von `count()` wurde die Berechnung, also die Transformation von `{"Liam", "kim", "Laura"}` zu `{"KIAM", "KIM", "LAURA"}` mit anschließender Filterung zu `{"LIAM", "LAURA"}`, gestartet. Konkret bedeutet das, dass bis zum Abschluss des Datenstroms nur Ausdrücke gesammelt, aber nicht berechnet werden. Diese Auswertungsstrategie wird als **Lazy Evaluation** bezeichnet.

Die Verwendung eines Datenstroms kann unter Umständen langsamer sein als eine optimierte Implementierung mit klassischen Kontrollstrukturen. Je nach Anwendungsfall wird dieses Manko zu Gunsten der besseren Lesbarkeit in Kauf genommen. Wird ein Datenstrom parallelisiert, ist die Verarbeitung deutlich schneller.

Andere Formen von Sammelstrukturen

Arrays.stream(Object[])

Ein `Array` bietet, im Gegensatz zu einer `Collection` keine Methode `stream()` an. Möchte man trotzdem verkettete Operationen auf ein `Array` anwenden, muss man die Hilfsklasse `Arrays` benutzen.

Beispiel:

```
jshell> String[] friends = new String[]{"Liam", "Kim", "Laura"}
friends ==> String[3] { "Liam", "Kim", "Laura" }

jshell> Arrays.stream(friends)
$2 ==> java.util.stream.ReferencePipeline$Head@78c03f1f
```

Stream.of(T... values)

Mit der statischen Fabrik-Methode `Stream.of(T… values)` ist es möglich einen Stream zu erzeugen, dessen Werte den Parametern der Methode `of(T… values)` entsprechen.

Beispiel:

```
Stream.of("Liam", "Kim", "Laura")
$1 ==> java.util.stream.ReferencePipeline$Head@7085bdee
```

To Be Continued

Im nächsten Infozettel werden die Methoden `map` und `filter` im Detail betrachtet.

Last updated 2021-05-04 18:22:41 +0200