Infozettel: Stream Operationen

Matthias Eurich – 2017-04-03

Table of Contents

Einführung

Mit Streams lassen sich viele Aufgaben mit `Collections` und ähnlichen Strukturen elegant lösen. In diesem Kapitel werden die Methoden `map()` und `filter()` vorgestellt.

Transformieren von Elementen: `map()`

Die Methode `map()` erlaubt das Anwenden einer `Function` auf jedes Element eines Datenstroms. Der Rückgabewert von `map()` ist ein `Stream`, der die Ergebnisse der Anwendung von `map()` enthält. Die ursprüngliche Datenstruktur wird dabei **nicht** verändert.

Beispiel:
Der folgende Code gibt alle Namen der Liste `friends` in Großbuchstaben unter der Verwendung von `map()` aus. Die Liste `friends` bleibt dabei unverändert.

```
List<String> friends = Arrays.asList("Liam", "Kim", "Laura");

friends.stream()
       .map(String::toUpperCase)
       .forEach(System.out::println);
```

Dieses Beispiel verdeutlicht, dass der Code lesbarer, verständlicher und damit auch wartbarer durch die Verwendung von `Streams` werden kann. Es werden keine Kontrollstrukturen oder temporäre Variablen für die Lösung des Problems benötigt.

Mehrdimensionale Sammelstrukturen zusammenführen: `flatMap()`

Mit der Methode `flatMap()` können mehrdimensionale Sammelstrukturen in einen `Stream` überführt werden. Als Parameter nimmt `flatMap` ein Objekt vom Typ `Function` entgegen.

Beispiel:
Der folgende Code überführt die Elemente einer mehrdimensionalen Liste in einen eindimensionalen Stream. Am Ende werden die Elemente ausgegeben.

```
List<List<String>> friends = Arrays.asList(
                              Arrays.asList("Liam", "Kim", "Laura"),
                              Arrays.asList("Milan", "Julia", "Emilia"));

friends.stream()
       .flatMap(List::stream)
       .forEach(System.out::println);
```

Als `Function` wird in diesem Beispiel eine Methodenreferenz auf die Methode `stream()` übergeben.

Den Typ des Datenstroms wechseln: `mapToInt`, `mapToDouble`, `mapToLong`

Für manche Anwendungsfälle ist es notwendig, den Typ des Datenstroms während der Verarbeitung zu ändern.

Beispiel:
Im folgenden Code wird das Alter aller Personen vom Typ `Person` summiert. Dazu muss der Stream aus Objekten vom Typ `Person` in einen `IntStream` umgewandelt werden, der nur noch das Alter der Personen enthält.

Zunächst die Klasse `Person`:

```
public class Person {
  private int age;
  private String name;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public int getAge() {
    return this.age;
  }
}
```

Nachfolgend der Code zum Summieren des Alters aller Personen einer Liste. Mit der Methode `mapToInt` wird der Datenstrom in einen `IntStream` umgewandelt. Als Parameter übergeben wir der Methode `mapToInt` einen Lambda-Ausdruck, der beschreibt, welchen Wert von Person wir haben möchten. Anschließend werden die Werte durch Aufruf der Methode `sum()` summiert.

```
List<Person> persons = Arrays.asList(new Person("Liam", 20),
                                     new Person("Kim", 28),
                                     new Person("Laura", 18));

persons.stream().mapToInt(person -> person.getAge()).sum();
```

Filtern von Elementen: `filter()`

Die Methode `filter()` nimmt als Parameter ein `Predicate` entgegen und erlaubt das Filtern nach bestimmten Kriterien. Der Rückgabewert von `filter()` ist ein `Stream`, de die Elemente enthält, auf die das übergebene `Predicate` passt.

Beispiel:
Der folgende Code gibt nur die Namen der Liste `friends` aus, die mit dem Buchstaben `L` beginnen.

```
List<String> friends = Arrays.asList("Liam", "Kim", "Laura");

friends.stream()
       .filter(name -> name.startsWith("L"))
       .forEach(System.out::println);
```

Last updated 2017-04-28 15:39:49 CEST