Java 8 Streams

Jan Trienes, Thijs Dorssers, Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek

March 13, 2018

Contents of this talk

Internal vs. External Iteration

- The collections framework now supports internal iteration
- This is a clear separation of concerns. The "how" and the "what" is encapsulated and independent

Consider the difference: both use

```
List<String> names = new ArrayList <>();
```

External iteration

```
Iterator<String> it = names.iterator();
while(it.hasNext()) {
  System.out.println(it.next());
}
```

versus

Internal iteration

```
names.forEach(name -> System.out.println(name));
```

Notice: the for-each loop is the same as external iteration.

Internal vs. External Iteration

External Iteration

The client takes full control over the iteration process and the actions that are applied to each item.

Internal Iteration

The sequence (produced by the collection) itself determines how elements in a sequence are accessed and the user defines the actions applied the items.

Streams API

New functionality for Collections framework

- Support for parallel bulk operations
- Heavily uses the stream concept (like in filter())
- Better separation of “what” or “how” based on lambda expressions.

What is a stream?

A stream seems in some way similar to a collection, allowing you to transform and retrieve data. But there are significant differences:

- A stream does not store any elements. It is more or less a n (active) bystander. The elements are 'produced' by an underlying collection or generated on demand.
- Stream operations do not mutate their sources. Instead, the source typically returns a new stream for further use.
- Stream operations are **lazy** when possible. This means they are not executed until their result is needed. For example, if you only ask for the first five long words instead of counting them all, then the filter will stop filtering after the fifth match.

Operations on Streams

Some of the new operations

`filter()` select. only those items, that match a predicate are contained. in the result. **I**

`map()` transform creates a new stream out of the results from the passed function. **I**

`distinct()` deduplicate returns a stream in which all items are distinct. Implies sorting of sorts. **I**

`count()` reduce returns the number of items in the stream. **T**

`forEach()` sic. applies a passed function to each item in the stream. **T**.

Where **I** is an intermediate operation, which produces a stream and **T** is a terminating operation, which *pulls* (starts) the stream and *consumes* it.

Note whitout terminating operation a stream will do nothing (as not advance)

because it is born lazy.

Terminal and Intermediate Operations

The previously mentioned operations belong to two categories:

Intermediate operation returns a stream, which can be used for further processing

Terminal operation returns a final result or void, which can not be used again

- **Intermediate** operations support a programming style which is known as *fluent programming*. It's the chaining of operations.
- The typical chain contains the operations *filter* (intermediate), *map* (intermediate), and *reduce*(terminal).
  - Reduce as in reduction[1]: indikken, inkoken, einkochen.
  - reduce examples: count, take average, collect into list.

[1] evaporate the contained water by heating

An Example using Streams

Goal: Find the names of all persons which are no older than 30 and which name ends with "daal".

Example of applying streams

```
void method() {
  List<Person> personList = new ArrayList<>();
  personList.add(new Person("John Doe", 15));
  personList.add(new Person("Jane Doe", 17));
  personList.add(new Person("Anne Modaal", 35));
  personList.add(new Person("Max Modaal", 19));
  personList.add(new Person("Erika Modaal", 28));

  personList
    .parallelStream() //supports parallel execution
    .filter(p -> p.age <= 30)
    .map(p::getName)
    .filter(s -> s.toLowerCase().endsWith("daal"))
    .forEach((String s) -> System.out.println(s));
}
```

Exercise See example above.

Find the stream implementation of the following: calculate the average age of these persons.

Additional Information

- Tutorial on Java 8 and lambda expressions by Angelika Langer and Klaus Kreft `http://angelikalanger.com/Lambdas/Lambdas.html`
- `java.util.stream`
- `java.util.function`