Processing Data in Memory Using the Stream API

→

Adding Intermediate Operations on a Stream

Processing Data in Memory Using the Stream API

Introducing the Stream API

The Stream API is probably the second most important feature added to Java SE 8, after the lambda expressions. In a nutshell, the Stream API is about providing an implementation of the well known map-filter-reduce algorithm to the JDK.

The Collections Framework is about storing and organizing your data in the memory of your JVM. You can see the Stream API as a companion framework to the Collections Framework, to process this data in a very efficient way. Indeed, you can open a stream on a collection to process the data it contains.

It does not stop here: the Stream API can do much more for you, than just process data from your collections. The JDK gives you several patterns to create streams on other sources, including I/O sources. Moreover, you can create your own sources of data to perfectly fit your needs, with little effort.

When you master the Stream API, you are able to write very expressive code. Here is a little snippet, that you can compile with the right static imports:

```
List<String> strings = List.of("one","two","three","four");
var map = strings.stream()
                 .collect(groupingBy(String::length, counting()));
map.forEach((key, value) -> System.out.println(key + " :: " + value));
```

This code prints out the following.

- It groups the strings by their length with `groupingBy(String::length)`
- It counts the number of strings for each length with `counting()`
- It then creates a `Map<Integer, Long>` to store the result

Running this code produces the following result.

```
3 :: 2
4 :: 1
5 :: 1
```

Even if you are not familiar with the Stream API, reading code that uses it gives you an idea of what it is doing at the first glance.

Introducing the Map-Filter-Reduce Algorithm

Before you dive in the Stream API itself, let us see the elements of the map-filter-reduce algorithm that you are doing to need.

This algorithm is a very classic algorithm to process data. Let us take an example. Suppose you have a set of `Sale` objects with three properties: a date, a product reference and an amount. For the sake of simplicity, we will suppose that the amount is just an integer. Here is your `Sale` class.

```
public class Sale {
    private String product;
    private LocalDate date;
    private int amount;

    // constructors, getters, setters
    // equals, hashCode, toString
}
```

Suppose you need to compute the total amount for the sales in March. You will probably write the following code.

```
List<Sale> sales = ...; // this is the list of all the sales
int amountSoldInMarch = 0;
for (Sale sale: sales) {
    if (sale.getDate().getMonth() == Month.MARCH) {
        amountSoldInMarch += sale.getAmount();
    }
}
System.out.println("Amount sold in March: " + amountSoldInMarch);
```

You can see three steps in this simple data processing algorithm.

The first step consists in taking into account only the sales that occurred in March. You are *filtering* out some elements you are processing, on a given criteria. This is precisely the filtering step.

The second step consists in extracting a property from the `sale` object. You are not interested in the whole object; what you need is its `amount` property. You are *mapping* the `sale` object to an amount, that is, an `int` value. This is the mapping step; it consists of transforming the objects you are processing to other objects or values.

The last step consists of summing all these amounts into one amount. If you are familiar with the SQL language, you can see that this last step looks like an aggregation. Indeed, it does the same. This sum is a *reduction* of the individual amounts into one amount.

By the way, the SQL language does a very good job at expressing this kind of processing in a readable way. The SQL code you need is really very easy to read:

```
select sum(amount)
from Sales
where extract(month from date) = 3;
```

Specifying a Result Instead of Programming an Algorithm

You can see that in SQL, what you are writing is a description of the result you need: the sum of the amounts of all the sales that were made in March. It is the responsibility of your database server to figure out how to compute that efficiently.

The Java snippet that computes this amount is a step-by-step description of how this amount is computed. It is described precisely, in an imperative way. It leaves little room for the Java runtime to optimize this computation.

Two of the goals of the Stream API are to enable you to create more readable and expressive code and to give the Java runtime some wiggle room to optimize your computations.

Mapping Objects to Other Objects or Values

The first step of the map-filter-reduce algorithm is the *mapping* step. A mapping consists of the transforming the objects or the values that you are processing. A mapping is a one-to-one transformation: if you map a list of 10 objects, you will get a list of 10 transformed objects.

In the Stream API, the mapping step adds one more constraint. Suppose you are processing a collection of *ordered* objects. It could be a list, or some other source of ordered objects. When you map that list, the first object you get should be the mapping of the first object from the source. In other words: the mapping step respects the order of your objects; it does not shuffle them.

> *A mapping changes the types of objects; it does not change their number.*

A mapping is modeled by the `Function` functional interface. Indeed, a function may take any type of object and returns an object of another type. Moreover, specialized functions may map objects to primitive types and the other way round.

Filtering out Objects

On the other hand, filtering does not touch the objects you are processing. It just decides to select some of them, and to remove the others.

> *A filtering changes the number of objects; it does not change their type.*

A filtering is modeled by the `Predicate` functional interface. Indeed, a predicate may take any type of object or primitive type and returns a boolean value.

Reducing Objects to Produce a Result

The reducing step is more tricky than it looks like. For now, we are going to live with this definition, that it is just the same kind of thing as an SQL aggregation. Think about *COUNT*, *SUM*, *MIN*, *MAX*, *AVERAGE*. By the way all these aggregations are supported by the Stream API.

Just to give you a hint on what awaits you on this path: the reduction step allows you to build complex structures with your data, including lists, sets, maps of any kind, or even structures that you can build yourself. Just take a look at the first example on this page: you can see a call to a `collect()` method, which takes an object built by a `groupingBy()` factory method. This object is a *collector*. The reduction may consist in collecting your data using a collector. Collectors are covered in detail later in this tutorial.

Optimizing the Map-Filter-Reduce Algorithm

Let us take another example. Suppose you have a collection of cities. Each city is modeled by a `City` class, which has two properties: a name and a population, that is, the number of people living in it. You need to compute the total population living in cities that have more than 100k inhabitants.

Without using the Stream API, you are probably going to write the following code.

```
List<City> cities = ...;

int sum = 0;
for (City city: cities) {
    int population = city.getPopulation();
    if (population > 100_000) {
        sum += population;
    }
}

System.out.println("Sum = " + sum);
```

You can recognize another map-filter-reduce processing on a list of cities.

Now, let us make a little thought experiment: suppose the Stream API does not exist, and that a `map()` and a `filter()` method exists on the `Collection` interface, as well as a `sum()` method.

With these (fictitious) methods, the previous code could become the following.

```
int sum = cities.map(city -> city.getPopulation())
                .filter(population -> population > 100_000)
                .sum();
```

From a readability and expressiveness point of view, this code is very easy to understand. So you may be wondering: why these map and filter methods have not been added to the `Collection` interface?

Let us dig a little deeper: what would be the return type of these `map()` and `filter()` methods? Well, since we are in the Collections Framework, returning a collection seems natural. So you could write this code in this way.

```
Collection<Integer> populations         = cities.map(city -> city.getPopulation());
Collection<Integer> filteredPopulations = populations.filter(population -> population > 100_000);
int sum                                 = filteredPopulations.sum();
```

Even if chaining the calls improves readability, this code should still be correct.

Now let us analyze this code.

- The first step is the mapping step. You saw that if you have to process 1,000 cities, then this mapping step produces 1,000 integers and put them in a collection.
- The second step is the filtering step. It goes through all the elements and removes some of them following the given criterion. That's another 1,000 elements to test and another collection to create, probably smaller.

Because this code returns a collection, it maps all the cities, then filters the resulting collection of integers. This works very differently from the *for loop* that you wrote in the first place. Storing this intermediate collection of integers may may result in a lot of overhead, especially if you have a lot of cities to process. The for loop does not have this overhead: it directly sums up the integers in the result, without storing them in an intermediate structure.

This overhead is bad, and there are cases where it can be even worse. Suppose you need to know if there are cities of more than 100k inhabitants in the collection. Maybe the first city of the collection is such a city. In that case, you will be able to produce a result with almost no effort. First, building the collection of all the populations from your cities, then filtering it and checking if the result is empty or not would be ridiculous.

For obvious performances reasons, creating a `map()` method that would return a `Collection` on the `Collection` interface is not the right way to go. You would end up creating unnecessary intermediate structures with a high overhead on both the memory and the CPU.

This is the reason why the `map()` and `filter()` methods have not been added to the `Collection` interface. Instead, they have been created on the `Stream` interface.

The right pattern is the following.

```
Stream<City> streamOfCities         = cities.stream();
Stream<Integer> populations         = streamOfCities.map(city -> city.getPopulation());
Stream<Integer> filteredPopulations = populations.filter(population -> population > 100_000);
int sum = filteredPopulations.sum(); // in fact this code does not compile; we'll fix it later
```

The `Stream` interface avoids creating intermediate structures to store mapped or filtered objects. Here the `map()` and `filter()` methods are still returning new streams. So for this code to work and be efficient, no data should be stored in these streams. The streams created in this code, `streamOfCities`, `populations` and `filteredPopulations` must all be empty objects.

It leads to a very important property of streams:

> *A stream is an object that does not store any data.*

The Stream API has been designed in such a way that as long as you do not create any non-stream object in a stream pattern, no computation of your data is conducted. In the previous example, you are computing the sum of the elements processed by your stream.

This sum operation triggers the computation: all the objects of the `cities` list are pulled one by one through all the operations of the stream. First they are mapped, then filtered, and summed up if they pass the filtering step.

A stream processes the data in the same order as if you write an equivalent for loop. In this way there is no memory overhead. Moreover, there are cases where you can produce a result without having to go through all the elements of your collection.

Using streams is about creating pipelines of operations. At some point your data will travel through this pipeline and will be transformed, filtered, then will participate in the production of a result.

A pipeline is made of a series of method calls on a stream. Each call produces another stream. Then at some point, a last call produces a result. An operation that returns another stream is called an intermediate operation. On the other hand, an operation that returns something else, including void, is called a terminal operation.

Creating a Pipeline with Intermediate Operations

An intermediate operation is an operation that returns another stream. Invoking such an operation adds one more operation on an existing pipeline of operations without processing any data. It is modeled by a method that returns a stream.

Computing a Result with a Terminal Operation

A terminal operation is an operation that does not return a stream. Invoking such an operation triggers the consumption of the elements of the source of the stream. These elements are then processed by the pipeline of intermediate operations, one element at a time.

A terminal operation is modeled by a method that returns anything but a stream, including void.

You cannot call more than one intermediate or terminal method on a given stream instance. If you do so, you will get an `IllegalStateException` with the following message: "stream has already been operated upon or closed", like on the following example. You cannot call the `toList()` method on `stream`, because you already called `map()` on it.

```
var stream = Stream.of(1, 2, 3, 4);
var stream1 = stream.map(i -> i + 1);
var list = stream.toList();
```

Avoiding Boxing with Specialized Streams of Numbers

The Stream API gives your four interfaces.

The first one is `Stream`, which you can use to define pipelines of operations on any kind of objects.

Then there are three specialized interfaces to handle streams of numbers: `IntStream`, `LongStream` and `DoubleStream`.These three streams use primitive types for numbers instead of the wrapper types to avoid boxing and unboxing. They have almost the same methods as the methods defined in `Stream`, with a few exceptions. Because they are handling numbers, they have some terminal operations that do not exist in `Stream`:

- `sum()`: to compute the sum
- `min()`, `max()`: to compute the minimum or the maximum number of a stream
- `average()`: to compute the average value of the numbers

- `summaryStatistics()`: this call produces a special object that carries several statistics, all computed on one pass over your data. These statistics are the number of elements processed by that stream, the min, the max, the sum and the average.

Following Good Practices

As you have seen, you are allowed to call only one method on a stream, even if this method is intermediate. So it is useless, and sometimes dangerous, to store streams in fields or local variables. Writing methods that take streams as arguments may also be dangerous, because you cannot be sure that the stream you receive has not been already operated upon. A stream should be created and consumed on the spot.

A stream is an object connected to a source. It pulls the elements it processes from this source. This source should not be modified by the stream itself. Doing so will lead to unspecified results. In some cases, this source is immutable or read-only, so you will not be able to do that, but there are cases where you could.

There are plenty of methods available in the `Stream` interface, and you are going to see most of them in this tutorial. Writing an operation that modifies some variables or fields outside the stream itself is a bad idea that can always be avoided. A stream should not have any *side effects*.

Last update: *September 14, 2021*

Processing Data in Memory Using the Stream API

➜

Adding Intermediate Operations on a Stream

Home > Tutorials > The Stream API > Processing Data in Memory Using the Stream API