Practical Guide to Java Stream API

Praveer Gupta
Nov 25, 2018 · 6 min read

```
        people                  ctingExample() {
der, Double> averageAgeByGender =
  people.stream()
          .collect(Collectors.grouping
                 Person::getGender,
                 Collectors.averaging
 
 
der, List<String>> nameByGender =
  people.stream()
          .collect(Collectors.grouping
                 Person::getGender,
                 llectors.mapping(Person::get
```

Java Stream API

`Stream` API was introduced in Java 8. It provided a ***declarative programming*** approach to iterate and perform operations over a collection. Until Java 7, `for` and `for each` were the only options available, which was an ***imperative programming*** approach. In this article I will introduce you to the `Stream` API and how it provides an abstraction over the common operations performed over a collection.

> *While using **Imperative Programming**, the developer uses the language constructs to write both **what to do and how to do**. Whereas while using **Declarative Programming**, developer has to focus only on defining **what to do** and the language or the framework takes care of the **how to do** part. Hence in Declarative Programming, the code is concise and less error-prone.*

The operations that are generally performed over a collection can be categorized into the following. Though the list below is not exhaustive, it covers most of the use cases that we come across in our daily programming. I will be using the below mentioned operations in the examples for introducing the `Stream` API.

- Transforming
- Filtering
- Searching
- Reordering
- Summarizing
- Grouping

In the examples, I will use a collection of `Person` objects. For easy understanding the definition of the `Person` class is as shown below.

```
public class Person {
    private final String name;
    private final int age;
    private final Gender gender;

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

    public String getName() { return name; }

    public int getAge() { return age; }

    public Gender getGender() { return gender; }
}

public enum Gender {
    MALE, FEMALE, OTHER
}
```

A quick introduction to the Stream API

Before diving deep into the examples of operations performed over a collection using Stream API, let's use an example to understand the Stream API itself.

```
List<Person> people = ...

// bulding a stream
List<String> namesOfPeopleBelow20 = people.stream()
    // pipelining a computation
    .filter(person -> person.getAge() < 20)
    // pipelining another computation
    .map(Person::getName)
    // terminating a stream
    .collect(Collectors.toList());
```

In the above example, multiple operations are chained together to form something like a processing pipeline. This is what we refer as a ***stream pipeline***. The stream pipeline made of the following three parts:

1. ***Stream builder*** — In the above example, we have a collection of `Person` represented by `people`. The `stream()` method, that was added on the `Collection` interface in Java 8, is called upon `people` to build the stream. Common sources for stream apart from Collection are arrays (`Arrays.stream()`) and generator functions (`Stream.iterate()` and `Stream.generate()`).

2. ***Intermediate operation(s)*** — Once a stream object is created, you can apply zero, one or more than one operations on the stream by chaining the operations, like in a builder pattern. All the methods that you see in the above example, `filter` and `map`, are methods on the `Stream` interface and they return the instance of `Stream` itself to enable chaining. As the operations return `Stream` itself, they are called intermediate operations.

3. ***Terminal operation*** — Once all the computations are applied, you finish the pipeline by applying a mandatory terminal operator. The terminal operators are also methods on the `Stream` interface and return a resultant type that is not a Stream. In the example above the `collect(Collectors.toList())` returns an instance of `List`. The resultant type may or may not be a Collection based on which

terminal operation is used. It can be a primitive value or an instance of an object that is not a Collection.

Let’s now look at the basic operations that we can do using Stream. Though we will be learning about the operations applied individually on Stream, you can always mix and match them to derive different results.

Transforming

Transforming means converting the type of value that is stored in each element of a collection. Let’s say we want to derive a collection of names of people from the person collection. In such a case we have to use a transformation operation to transform the person to name.

In the example below, we use the `map` intermediate operator to transform `People` to a `String` holding the name of the person. `Person::getName` is a method reference and is equivalent to `person -> person.getName()` and is an instance of a Function.

```
List<String> namesOfPeople = people.stream()
    .map(Person::getName)
    .collect(Collectors.toList());
}
```

Filtering

As the word suggests, filtering operations allow objects to flow through itself only if the object fulfills the conditions laid upon by a Predicate. The filter operator is composed with the `Predicate` before it is applied to the Stream.

Filtering can also be thought of as selecting few elements based on count. Stream API provides `skip()` and `limit()` operators for this purpose.

In the first example below, the `person -> person.getAge() < 20` Predicate is used to build a collection containing only people below age of 20. In the second example below, 10 persons are selected after skipping first 2.

```
// filtering using Predicate
List<Person> listOfPeopleBelow20 = people.stream()
    .filter(person -> person.getAge() < 20)
    .collect(Collectors.toList());
```

```
// count based filtering
List<Person> smallerListOfPeople = people.stream()
    .skip(2)
    .limit(10)
    .collect(Collectors.toList());
```

Searching

Again as the word suggests, searching on a collection means to search for an element or the existence of an element based on a criteria, which again is represented as a `Predicate`. Searching for an element may or may not return a value, hence you get an Optional. Searching for existence of an element will return a `boolean`.

In the examples below, searching for an element is done using `findAny()` and searching for existence is done using `anyMatch()`.

```
// searching for a element
Optional<Person> any = people.stream()
    .filter(person -> person.getAge() < 20)
    .findAny();

// searching for existence
boolean isAnyOneInGroupLessThan20Years = people.stream()
    .anyMatch(person -> person.getAge() < 20);
```

Reordering

If you want to order the elements in a collection, you can use the `sorted` intermediate operator. It takes an instance of a `Comparator` interface. To create the instance I have used the `comparing` factory method on `Comparator`. If you are interested to know more around this, please checkout this link.

In the example below the resulting collection is sorted by age in descending order.

```
List<Person> peopleSortedEldestToYoungest = people.stream()
    .sorted(Comparator.comparing(Person::getAge).reversed())
    .collect(Collectors.toList());
```

> *Unlike other operations we have seen till now, the `sorted` operation is stateful. It means that the operator will have to see all the elements in the stream before the result of sorting*

> *can be provided to further intermediate or terminal operator. Another example of such an operator is `distinct`.*

Summarizing

Sometimes you want to derive information from a collection. As an example, deriving the sum of ages of all the people. In `Stream` API, this is achieved using ***terminal operators***. `reduce` and `collect` are the generic terminal operators provided for this purpose. There are high level operators also like `sum`, `count`, `summaryStatistics`, etc. that are built upon `reduce` and `collect`.

```
// calculating sum using reduce terminal operator
people.stream()
    .mapToInt(Person::getAge)
    .reduce(0, (total, currentValue) -> total + currentValue);

// calculating sum using sum terminal operator
people.stream()
    .mapToInt(Person::getAge)
    .sum();

// calculating count using count terminal operator
people.stream()
    .mapToInt(Person::getAge)
    .count();

// calculating summary
IntSummaryStatistics ageStatistics = people.stream()
    .mapToInt(Person::getAge)
    .summaryStatistics();
ageStatistics.getAverage();
ageStatistics.getCount();
ageStatistics.getMax();
ageStatistics.getMin();
ageStatistics.getSum();
```

> *`reduce` and `collect` are Reduction operations operations. `reduce` is meant for immutable reduction whereas `collect` is meant for mutable reduction. Immutable reduction is the preferred approach. However for situations where performance is important, mutable reduction is preferred instead of immutable.*

Grouping

Grouping can also be called classifying. Sometimes we want to break a collection into several groups. The resulting data structure in such a case is a `Map` where the key represents the grouping factor and the value represents the characteristic for a particular group. The `Stream` API provides `Collectors.groupingBy` for such scenarios.

In all the examples below, the grouping is done using gender. The difference is in the values. In the first example a collection of `Person` is created for each group. In the second, `Collectors.mapping()` is used to extract the name of each `Person` to create a collection of names. In the third, the age of each `Person` is extracted and an average age for each group is calculated.

```
// Grouping people by gender
Map<Gender, List<Person>> peopleByGender = people.stream()
    .collect(Collectors.groupingBy(
        Person::getGender,
        Collectors.toList()));

// Grouping person names by gender
Map<Gender, List<String>> nameByGender = people.stream()
    .collect(Collectors.groupingBy(
        Person::getGender,
        Collectors.mapping(Person::getName, Collectors.toList())));

// Grouping average age by gender
Map<Gender, Double> averageAgeByGender = people.stream()
    .collect(Collectors.groupingBy(
        Person::getGender,
        Collectors.averagingInt(Person::getAge)
    ));
```

Summary

In this guide we saw that the Java `Stream` API provides lots of in-built functionality to help in performing operations on a collection using a ***stream pipeline***. The API is declarative which makes the code precise and less error prone. I hope this guide provides you with enough information for you to get started on effectively using Java `Stream` API in your daily programming.

. . .

Also posted on https://praveer09.github.io/technology/2018/11/24/practical-guide-to-java-stream-api/