Functional programming

Managing collections with streams

In this document we are going to introduce streams, a powerful tool that lets us manage Java collections from a functional point of view, so that our code will be shorter, less error prone and more efficient.

1. Introduction to streams

A stream is a new concept designed to process large (or small) amounts of data using a new level of abstraction in combination with lambda expressions that can be easily parallelizable to take advantage of all available CPU cores without having to do it ourselves manually (split the computation and create new threads).

A stream:

- Is a special object on which we can define operations (using lambda expressions, usually)
- It doesn't hold any data (acts as an intermediary).
- It doesn't change the data it processes. The compiler allows you to do so but you shouldn't, since you could get an unpredictable behaviour.
- It processes all data of an operation and passes it to the next operation. It doesn't do anything until you call a *final* or terminal operation (we'll see what's a final operation)
- It behaves, in some ways, similar to SQL.

Streams are a really powerful (and maybe hard to understand) concept of Java 8. They are useful when we want to filter and process a large list or amount of data.

1.1. Stream vs Collection

First of all, for compatibility issues, the *Collection* framework hasn't been transformed to work like streams do, so they are separate things. Most of the time, streams are generated from collections and several times they generate a new collection as a result, but they're not the same.

With a collection, the programmer has to iterate and operate with all its values manually, and if he wants to parallelize the operation, he has to create the necessary threads and divide the problem by himself. With a stream, you only have to define the operations that will be done with all data, and it will automatically iterate and apply the defined operations, and also divide the problem and generate threads when using parallel streams.

1.2. Defining a stream

In order to define a stream, we usually get it from a collection, through the `stream` method that was incorporated in Java 8:

```
List<String> texts = ...
Stream<String> stream = texts.stream();
```

From this stream, there are two types of operations that can be done:

- **Intermediary** or *lazy* operations: they transform this stream into another
- **Final** operations: they close the stream and produce a final result. This result can be a collection, a numeric value, and so on.

2. Intermediary or lazy operations

These operations return another stream, so that we can chain as many of them as we want. The most typical intermediary operations are filters and mappings, but there are some other useful intermediary operations that we will see here.

2.1. Filters

A filter operation takes a `Predicate` object as an argument, meaning that it will accept data that matches the condition given by this predicate and reject data that doesn't. For instance, if we have a list of `Person` objects:

```
List<Person> people = new ArrayList<>(10);
people.add(new Person(16, "Peter"));
people.add(new Person(22, "Mary"));
people.add(new Person(43, "John"));
people.add(new Person(70, "Amy"));
```

We can get a stream from that list, and filter those objects whose age is older than 18. Then, we can explore this stream and print the selected objects:

```
Stream<Person> stream = people.stream();
// Intermediary
Stream<Person> stream2 = stream.filter(p -> p.getAge() >= 18);

// A shorter way to do exactly the same
Stream<Person> stream = people.stream().filter(p -> p.getAge() > 18);
```

The parameter that we use in the `filter` method is a `Predicate`, a functional interface provided by Java 8. It has a method called `test` that returns a boolean that tells if a given test is true or false. In our case, we use a lambda expression to implement this predicate, and the test that must be passed is that every selected person `p` must have at least 18 years (`p.getAge() >= 18`).

2.2. Mappings

Mapping operations use a `Function` object (another functional interface brought by Java 8), taking an item (input) and returning a different output. We can use mappings to extract some specific information from complex objects, or to transform an element into another, different element. For example, we can use it to extract person ages in our previous example:

```
Stream<Integer> ages =
people.stream()
      .filter(p -> p.getAge() >= 18)
      .map(p -> p.getAge());
```

Note that we can chain as many intermediary operations as we need. In this case, we are chaining a filtering operation with a mapping, and the result is always another stream.

2.3. Another intermediary operations

Apart from filters and maps, there are some other intermediary operations available in Stream interface. For instance:

- `sorted` operation takes a `Comparator` as argument, so that it can sort the stream elements according to this comparator. This example sorts people by age in ascending order

```
Stream<Person> stream =
people.stream()
      .sorted((p1, p2) -> Integer.compare(p1.getAge(), p2.getAge()));
```

- `limit` operation takes an integer N as argument, and returns the first N elements of the stream. This example returns a stream with the 3 youngest people of original *people* list:

```
Stream<Person> stream =
people.stream()
      .sorted((p1, p2) -> Integer.compare(p1.getAge(), p2.getAge()));
      .limit(3);
```

3. Final operations

Final operations close the stream and get some kind of object (for instance, a Collection) or final value (for instance, an average, or maximum), or they just process the stream data in a final way (for instance, they print the results in a text file or screen). An important thing to note is that a stream can only have one final operation. Once it has processed data, the stream can't be reused, so you need to create another stream.

3.1. Reduction operations

Reductions return one element that can be of any type. Sometimes, they won't return anything (for example, the minimum element of an empty list), and that would be a problem if we assign that result to a variable. We can use the class `Optional` to manage those situations.

Types of reductions:

- Return `Optional`: `max`, `min`, `findAny`, `findFirst`
- Return long: `count`
- Return boolean: `allMatch`, `noneMatch`, `anyMatch`
- Other (the return value depends on the operation done): `reduce`

Let's see an example: if we want to sum the ages of all the `Person` objects older than 18, we could use the `reduce` method like this:

```
int sumAges = people.stream()
    .filter(p -> p.getAge() > 18)
    .map(p -> p.getAge())
    .reduce(0, (a,b) -> a + b);
```

The `reduce` method's first argument is the base case, the value that will be taken if the list is empty. Then, for every age found in the filtered list, it is added to this base case.

Other operations like `max`, doesn't need to take a base value, so they will return an `Optional` (in case the list is empty there's no result). Here we get the maximum age of all the filtered objects:

```
Optional<Integer> maxAge = people.stream()
    .filter(p -> p.getAge() > 18)
    .map(p -> p.getAge())
    .reduce(Integer::max);

// Will print Optional[70]
System.out.println(maxAge);
// Better
System.out.println(maxAge.isPresent()?maxAge.get():"No max age");
```

With `mapToInt()`, `mapToDouble()`, etc... values are casted into primitive types and you can do more and simpler operations. Like calculate the average of all ages (older than 18):

```
OptionalDouble avgAge = people.stream()
    .filter(p -> p.getAge() > 18)
    .mapToInt(p -> p.getAge()).average();

System.out.println(
    avgAge.isPresent()?
        "Avg: " + avgAge.getAsDouble()
        :"No ages");
```

Let's see another example: we ask if there is any `Person` object younger than 18:

```
if(people.stream().anyMatch(p -> p.getAge() < 18))
{
    System.out.println("There are children in the collection!");
}
```

3.2. Collecting operations

These kind of operations are also final. Instead of returning an object, a primitive value, or nothing, they usually return a Collection, or sometimes a String (joining operation). The method used for this operations is `.collect()`.

The most basic usage is to pass one parameter, a `Collector` object, which we can create from the `Collectors` class. This example returns a string with all the person names joined by commas:

```
String names = people.stream()
    .filter(p -> p.getAge() >= 18)
    .map(p -> p.getName())
    .collect(Collectors.joining(",", "Adults: ", ""));

System.out.println(names); // Adults: Mary,John,Amy
```

We can also generate new Lists:

```
List<Person> older = people.stream()
    .filter(p -> p.getAge() >= 18)
    .collect(Collectors.toList());
```

We can even return a `Map` using `Collectors.groupingBy(key, calculated value)`. For instance, in this example we get a map with all the ages, and the number of people having each age:

```
Map<Integer, Long> ages = persons.stream()
    .collect(Collectors.groupingBy(
        p -> p.getAge(),
        Collectors.counting()));

System.out.println(ages.toString()); // {70=1, 22=1, 43=1}
```

3.3. Other final operations

There are other final operations that do not produce any particular result. For instance, we can use `forEach` operation, which takes a `Consumer` object. This is a functional interface that takes an object as a parameter and returns nothing. It is usually applied to printing data on a screen or file.

```
people.stream().filter(p -> p.getAge() > 18).forEach(System.out::println);
```

Exercise 1:

Create a project called **Hotels**. Define a class called `Hotel`, with three attributes: the hotel name (string), the hotel location (a city name) and the hotel rating (a floating point value between 0 and 5, both included). Define the constructor to set those values, and the corresponding getters or setters that you may need.

Then, in the main program, create a list of hotels (at least with 5 hotels) and do the following:

- Sort the hotels by rating in descending order using a lambda expression, and show the result in the console.
- Get all the hotels whose rating is greater than 3. Store the result in a list and show it in the console.
- Do the same operation, but now do not store the result in a list, just print it in the console.
- Print in the screen the names of the hotels in Alicante, separated by commas.
- Get the number of hotels whose rating is 5 and print the result in the screen

Exercise 2:

Using the project **ListFilter** of previous document, we're going to add some things:

1. In the `Main` class, add another static method called `getOldestNames (List<Student> list)` that returns a `List<String>`. Inside, you'll have to create a stream from the list received as parameter, and generate a list of the names of the three oldest people of the list. Try it and print the results to see if it works well.
2. Create another static method in the `Main` class called `getAllSubjects (List<Student> list)` that returns a `Set<String>`. In this method we're going to create a stream out of the student's list and generate a set that contains all subjects which

> appear at least in one student (or more) of the list, ordered alphabetically. Try it passing some list and printing the results.

4. Some advanced concepts regarding streams

In this last section of the document, we are going to learn some advanced concepts regarding stream management, such as how to chain some interface objects, or deal with files using streams.

4.1. Chaining functional interfaces

In some functional interfaces, such as `Consumer` or `Predicate`, we have some default, implemented methods that let us chain these objects somehow. For instance, we can check two predicates at once using `and` operation:

```
Predicate<String> p1 = s -> s.length() < 20;
Predicate<String> p2 = s -> s.length() > 10;
Predicate<String> p3 = p1.and(p2); // Applies both p1 and p2
```

This example applies two `Consumer` objects sequentially. The first one prints the list items in the screen, and the second one adds the elements of another list to the first one.

```
List<String> strings = Arrays.asList("one", "two", "three", "four", "five");
List<String> result = new ArrayList<>();
Consumer<String> cPrint = System.out::println;
Consumer<String> cAdd = result::add;
// Will print and then add to the other list (chaining consumers)
strings.forEach(cPrint.andThen(cAdd));
```

The `Function` interface takes an object (or more) and returns another object. There are several types, such as:

```
@FunctionalInterface
public interface Function<T ,R>
{
    R apply(T t);
}

@FunctionalInterface
public interface BiFunction<T, U ,R>
{
    R apply(T t, U u);
}
```

We can combine two or more of these functions to make a more complex one. Let's see an example:

```java
/* This BiFunction takes a string s and an integer i and returns a
   substring of s containing the first i characters */
BiFunction<String, Integer, String> biFunc = (s,i) -> s.substring(i);
// This Function takes a string and converts it to uppercase
Function<String, String> func = s -> s.toUpperCase();
// This BiFunction combines both previous functions in a third one
BiFunction<String, Integer, String> biFunc2 = biFunc.andThen(func);
```

Then, we can apply this complex function this way:

```java
String test = "This is a test string";
int number = 7;
String result = biFunc2.apply(test, number); // "This is"
```

4.2. Functional programming and I/O

Let's see some new features added to Java regarding functional programming and I/O tasks, such as file reading or writing.

`.lines()` method was added to `BufferedReader`. It returns a `Stream<String>` with all the lines of the file. This example prints in the screen of the lines of a file containing text "*Login:*".

```java
try(BufferedReader reader = new BufferedReader(
        new FileReader("/home/arturo/file.txt")))
{
    reader.lines()
          .filter(line -> line.contains("Login:"))
          .forEach(System.out::println);
} catch ...
```

`Files.lines()` method even creates the Stream without having to create a `BufferedReader`, and it's `AutoCloseable`, so you can create it inside a try:

```java
try(Stream<String> stream = Files.lines(
    Paths.get("/home/arturo", "file.txt")))
{
    stream.filter(line -> line.contains("Login:"))
          .forEach(System.out::println);
} catch ...
```

`Files.list()` returns a `Stream<Path>` containing a list of all files and directories present in the current directory (passed as a parameter):

```
try(Stream<Path> stream = Files.list(Paths.get("/home/arturo")))
{
    // Prints subdirectories
    stream.filter(path -> path.toFile().isDirectory())
          .forEach(System.out::println);
} catch ...
```

`Files.walk()` is very similar but it also explores subdirectories. The second parameter is the maximum depth you want to go in the directory tree:

```
try(Stream<Path> stream = Files.walk(Paths.get("/home/arturo"), 2)) {
    // Prints subdirectories
    stream.filter(path -> path.toFile().isDirectory())
          .forEach(System.out::println);
} catch ...
```

4.3. More collection features

Regarding collections, there are also some new features since Java 8. Let's see some of them...

There is a new method, `forEach()`, in Iterable collections:

```
List<Integer> list = Arrays.asList(4,6,7,9);
list.forEach(elem -> {
    if(elem % 2 == 0)
    {
        System.out.println(elem);
    }
});
```

We can also chain comparison criteria. Let's see a comparator to order people by their last name, and if it's equal, use the first name (Java 7 vs Java 8):

```
List<Person> list = new ArrayList<>();

// JAVA 7
list.sort(new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2)
    {
        int last = p1.getLastName().compareTo(p2.getLastName());
        if(last == 0)
        {
            // Equal last name
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
        return last;
    }
});

// JAVA 8
list.sort(Comparator.comparing(Person::getLastName)
    .thenComparing(Person::getFirstName));
```

There's a `Map.forEach(BiConsumer)` method that works like `List.forEach` but with `BiConsumer (key,value)`, so that we can get the key and the value separately in each iteration.

```
Map<String, Person> map = new HashMap<>();
...

map.forEach((key,person) ->
    System.out.println("Key: " + key +
        ". Person: " + person.getFullName()));
```

Regarding maps, there are other useful methods, such as `getOrDefault(key, defaultValue)` that gets a default value if key doesn't exist, or `putIfAbsent(key, value)` that puts the new value only if key is not present (old `put()` method overwrites previous value if key exists, without asking)

```
Map<String, Person> map = new HashMap<>();
map.putIfAbsent(key, person);
```

Exercise 3:

You will be provided with a sample file with two source files: a `Product` class with some attributes, constructors, getters and setters, and a `Main` application that creates a list of `Product` objects.

> Place these files in a project and, from this point, we will define some streams, lambda expressions and functions to get results from this collection.
>
> - Define a `Comparator` using a lambda expression to sort the product list by category and name (in this order). Sort the list using `Collections.sort` method and show it to check that the order is correct.
> - Create a stream that shows the products whose category is "Tablets". The stream will return a list, and we'll iterate over it to show the products information. How would you do this without generating any secondary list?
> - Create a stream that calculates the average of prices for a given category. Test this stream in the main application and show the results for "Videogames" category.
> - Create a `BiFunction` that takes two parameters: a list of `Product` objects and a price, and returns a String containing all the product names (separated by commas) whose price is higher than the one provided as a parameter. Use this function in the main application with a price of 100 and show the results.
> - Create a stream that counts how many products are there grouped by category. Explore the resulting map and show it in the console.