Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistema e Industrial
Computación Paralela y Distribuida

Ejercicio 2

Descomprima el archivo ejercicio_2.zip anexo

Las modificaciones deben ser realizadas dentro del archivo `StudentAnalytics.java`.

No se puede modificar las firmas (signatures) de los métodos públicos ni protegidos dentro de `StudentAnalytics` ni borrarlos. Puede adicionar los métodos que desee.

Implementar `StudentAnalytics.averageAgeOfEnrolledStudentsParallelStream` para que haga lo mismo que `averageAgeOfEnrolledStudentsImperative` pero usando streams paralelos de java.

Implementar `StudentAnalytics.mostCommonFirstNameOfInactiveStudentsParallelStream` para que haga lo mismo que `mostCommonFirstNameOfInactiveStudentsImperative` usando streams paralelos.

Implementar `StudentAnalytics.countNumberOfFailedStudentsOlderThan20ParallelStream` para que haga lo mismo que `countNumberOfFailedStudentsOlderThan20Imperative` usando streams paralelos.

Documentación:

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html**Java 8 Streams: An Intro to Filter, Map and Reduce Operations**

After shying away from them for years, Java finally embraced functional programming constructs in the spring of 2014. Java 8 includes support for lambda expressions, and offers a powerful Streams API which allows you to work with sequences of elements, such as lists and arrays, in a whole new way.

In this tutorial, I'm going to show you how to create streams and then transform them using three widely used higher-order methods named `map`, `filter` and `reduce`.

Code: https://github.com/sitepoint-editors/MapFilterReduceOperations

Creating a Stream

As you can tell from its name, a stream is just a sequence of items. Although there are lots of approaches to stream creation, for now, we'll be focusing only on generating streams from lists and arrays.

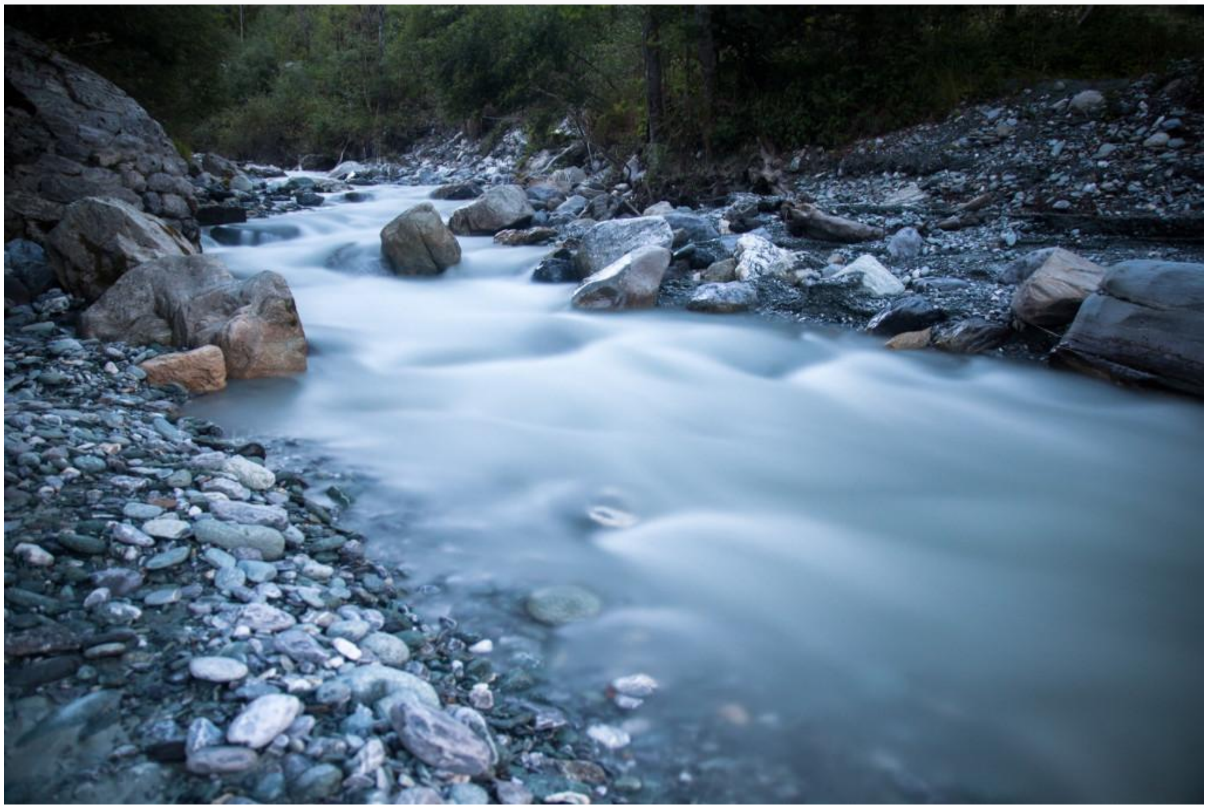
In Java 8, every class which implements the `java.util.Collection` interface has a `stream` method which allows you to convert its instances into `Stream` objects. Therefore, it's trivially easy to convert any list into a stream. Here's an example which converts an `ArrayList` of `Integer` objects into a `Stream`:

```
// Create an ArrayList
List<Integer> myList = new ArrayList<Integer>();
myList.add(1);
myList.add(5);
myList.add(8);

// Convert it into a Stream
Stream<Integer> myStream = myList.stream();
```

If you prefer arrays over lists, you can use the `stream` method available in the `Arrays`class to convert any array into a stream. Here's another example:

```
// Create an array
Integer[] myArray = {1, 5, 8};

// Convert it into a Stream
```

```
Stream<Integer> myStream = Arrays.stream(myArray);
```



The `map` Method

Once you have a `Stream` object, you can use a variety of methods to transform it into another `Stream` object. The first such method we're going to look at is the `map`method. It takes a lambda expression as its only argument, and uses it to change every individual element in the stream. Its return value is a new `Stream` object containing the changed elements.

To give you a realistic example, let me show you how you can use `map` to convert all elements in an array of strings to uppercase.

You start by converting the array into a `Stream`:

```
String[] myArray = new String[]{"bob", "alice", "paul", "ellie"};
Stream<String> myStream = Arrays.stream(myArray);
```

Then you call the `map` method, passing a lambda expression, one which can convert a string to uppercase, as its sole argument:

```
Stream<String> myNewStream =
             myStream.map(s -> s.toUpperCase());
```

The `Stream` object returned contains the changed strings. To convert it into an array, you use its `toArray` method:

```
String[] myNewArray =
              myNewStream.toArray(String[]::new);
```

At this point, you have an array of strings, all of which are in uppercase.

I hope you are now beginning to realize that with this style of programming, you can do away with loops, and the code you write can be very concise and readable.

The `filter` Method

In the previous section, you saw that the `map` method processes every single element in a `Stream` object. You might not always want that. Sometimes, you might want to work with only a subset of the elements. To do so, you can use the `filter` method.

Just like the `map` method, the `filter` method expects a lambda expression as its argument. However, the lambda expression passed to it must always return a `boolean` value, which determines whether or not the processed element should belong to the resulting `Stream` object.

For example, if you have an array of strings, and you want to create a subset of it which contains only those strings whose length is more than four characters, you would have to write the following code:

```
Arrays.stream(myArray)
      .filter(s -> s.length() > 4)
      .toArray(String[]::new);
```

The code above looks a lot more concise than the code we wrote in the previous example because I've chained all the `Stream` methods. Most developers prefer writing functional code in this manner because, usually, there's no need to store references to intermediate `Stream` objects.

Reduction Operations

A reduction operation is one which allows you to compute a result using all the elements present in a stream. Reduction operations are also called terminal operations because they are always present at the end of a chain of `Stream` methods. We've already been using a reduction method in our previous examples: the `toArray`method. It's a terminal operation because it converts a `Stream` object into an array.

Java 8 includes several reduction methods, such as `sum`, `average` and `count`, which allow to perform arithmetic operations on `Stream` objects and get numbers as results. For example, if you want to find the sum of an array of integers, you can use the following code:

```java
int myArray[] = { 1, 5, 8 };
int sum = Arrays.stream(myArray).sum();
```

If you want to perform more complex reduction operations, however, you must use the `reduce` method. Unlike the `map` and `filter` methods, the `reduce` method expects two arguments: an identity element, and a lambda expression. You can think of the identity element as an element which does not alter the result of the reduction operation. For example, if you are trying to find the product of all the elements in a stream of numbers, the identity element would be the number 1.

The lambda expression you pass to the `reduce` method must be capable of handling two inputs: a partial result of the reduction operation, and the current element of the stream. If you are wondering what a partial result is, it's the result obtained after processing all the elements of the stream that came before the current element.

The following is a sample code snippet which uses the `reduce` method to concatenate all the elements in an array of `String` objects:

```java
String[] myArray = { "this", "is", "a", "sentence" };
String result = Arrays.stream(myArray)
                .reduce("", (a,b) -> a + b);
```

Conclusion

You now know enough to start using the `map`, `filter` and `reduce` methods in your projects. For the sake of the brevity, throughout this tutorial, I've used only serial streams. If you have computationally intensive map operations, or if you expect your streams to be very large, you should consider using parallel streams instead. Fortunately, its very easy to convert any stream into its parallel equivalent. All you need to do is call its `parallel` method.

I'd also like to let you know that if you prefer not to use lambda expressions while working with the `map`, `filter`, and `reduce` methods, you can always use method references instead.

To learn more about the Streams API and the other methods it has to offer, you can refer to its documentation