Lab
Stream API

Jakob Hutter & Tadhg Mulvey

December 3, 2024

Overview

The Stream API in Java provides a powerful framework for processing collections of data in a functional and declarative style. It simplifies bulk operations such as filtering, mapping, sorting, and reducing, enabling developers to write concise and readable code. By abstracting away the complexity of handling threads and synchronization, the Stream API also makes it easier to leverage parallel processing to optimize performance on multi-core systems.

In this Lab, you will explore the foundational concepts and functionalities of the Stream API by working with Streams. Your first task is to implement `StreamCalculator` class to perform various operations on a list of integers. To refine your skills with the Stream API, the second exercise lets you complement the class `StreamCollector`, exploring more advanced Stream API operations and challenging you to use the Collect Interface with streams.

Necessary imports for both exercises:

```
import java.util.*;
import java.util.stream.*;
```

1 StreamCalculator

It is expected that you present the functionality of your code along with some examples. Here is an example List to test the functionality of your code:

```
List<Integer> numbers = Arrays.asList(17, 11, 19, 13, 43, 37, 41, 31, 29, 97, 5, 3, 7, 23);
```

1.1 Initalize Class

The first task focuses on creating a class called `StreamCalculator`, with the following specifications:

1. Every instance takes a `List<Integer>` as required input.
2. Create a getter function that returns the instance list as a stream and a getter function that returns the list itself.

3. Create a function `streamToList()` that utilizes Stream API to transform an input stream and return a `List<Integer>`. *Tip: use `.collect(Collectors.toList())`, for more details on collect see Exercise 2.*

The second function is necessary because a stream can only be operated on once. Therefore, the information has to be stored as a list if it will be used again as a stream. We will demonstrate this here to exemplify the different operations.

1.2 Basic Stream Operations

Next, you have to extend the operability of the `StreamCalculator` by using stream operators such as `.map()`, `.filter()`, and others. For all functions, you are required to use only Stream API operators; do not work directly with the 'List' type. The functions should update the instance's internal `List<Integer>` accordingly.

1. `sort()` - Sorts the numbers.
2. `filter(start, end)` - Filters all numbers that are within the range of `start` and `end`.
3. `add(input)` - Adds the input integer to all elements of the list.

1.3 Printing

Using the `forEach()` operation of the Stream API, create a function `print()` that prints the list, given a specific separator.

1.4 Output

The functions created here shall return a single value using the Stream API. Utilize the Stream API method **.reduce()** for that.

1. **sum()** will return the sum of all elements in the instance's list. *Tip: set the identity to 0, which sets the start of the addition to 0.*
2. **product()** will multiply all elements of the instance's list and return the result. *Tip: set the identity to 1, to avoid multiplication with 0.*

1.5 Builder Blocks

A stream can only be operated on once. To always store the result of a single operation as a list and then reload the stream again, as required in earlier exercises, is computationally demanding. To resolve this, the Stream API allows you to construct "Builder Blocks," meaning that multiple operations can be run on a stream in succession.

The function **addAndSum()** returns the sum of all values of the list after adding an input integer to all elements (a combination of **add()** and **sum()**). Create this function utilizing the concept of Builder Blocks.

Bonus: Parallel Streams

The goal of this exercise is to demonstrate the use of `parallelStream()` in the Stream API to take advantage of parallel processing. Extend the `StreamCalculator` class to include a method that calculates the sum of all elements in the list using a parallel stream. Parallel streams can improve performance by distributing work across multiple CPU cores, making them useful for operations on large datasets, which usually appear in commercial use of the Stream API. In general parallel stream is recommended when (Source: Stack Overflow Question on Parallel Streams):

- Massive amount of items to process (or the processing of each item takes time and is parallelizable)
- Performance problem in the first place
- Not already running the process in a multi-thread environment

For the Bonuspoint add a method `sumParallel()` to the `StreamCalculator` class. This method should use the `parallelStream()` function to process the list in parallel. In the `main` method, demonstrate the use of the `sumParallel()` function by utilizing `import java.util.Random;` to generate a list with 100 random integer values between 1 and 100. Compare its output with the `sum()` function to verify correctness.

2 The Collect Function

The `Collector Interface` provides a way to define the reduction operations on the elements of a stream. This allows us to use the `collect()` function which is a terminal operation that transforms the elements of a stream into a different form, such as a `List, Set, Map`, or even a custom data structure.

In these next questions you'll be working with lists of two classes Book and Movie. Use the `collect()` function to turn the lists of these classes into maps or vice versa where applicable

1. Complete the `StreamCollector` class provided in the respective file.
   a. A completed constructor which the lists of the classes as parameters
   b. Getters for the Streams of each list (As in Exercise 1)
2. Create Two seperate methods that partition each list into Contemporary (After you were born) and Classics (Before) using the partitionBy() function
   You'll need to use a map() for this.
3. A method that prints out movies in the list that are based on books (Same title) also in a list using a filter() on the stream
4. Use the groupingBy() function to
   a. Group movies by their director and print it out to the terminal
   b. Group Books by their author and print it out to the terminal
   Note : Differnce between partitioningBy() and groupBy() is that partitioningBy() can only create one group and is based on a predicate whereas groupBy() can create multiple groups based on a classifier function

An understanding of how a Map works is essential to these questions, see here for details

Solutions

Exercise 1

```
import java.util.*;
import java.util.stream.*;
import java.util.Random; //for Bonus


public class StreamCalculator {

    //Q1.
    List <Integer> list;
    public StreamCalculator(List <Integer> inputList) {
        this.list = inputList;
    }
    //Q1.2
    public Stream<Integer> getStream() {
        return list.stream();
    }
    public List<Integer> getList() {
        return list;
    }
    //Q1.3
    public List<Integer> streamToList (Stream<Integer> inputStream) {
        return inputStream.collect(Collectors.toList());
    }

    //Q2.1
    public void sort() {
        // list = list.stream().sorted().collect(Collectors.toList());
        list = streamToList(getStream().sorted());
    }
    //Q2.2
    public void filter(Integer start, Integer end) {
        //list = list.stream().filter(x -> x >= start  x <=
            end).collect(Collectors.toList());
        list = streamToList(getStream().filter(x -> x >= start && x <= end));
    }

    //Q2.3
    public void add(Integer value) {
        //list = list.stream().map(x -> x + value).collect(Collectors.toList());
        list = streamToList(getStream().map(x -> x + value));
    }

    //Q3
    public void print (String separator) {
        getStream().forEach(s -> System.out.print(s + separator));
        System.out.println();
    }
```

```java
//Q4.1
public Integer sum() {
    // list.stream()
    return getStream().reduce(0, (x, y) -> x + y);
}
//Q4.2
public Integer product() {
    // list.stream()
    return getStream().reduce(1, (x,y)-> x*y);
}

//Q5
public Integer addAndSum(Integer value) {
    return list.stream()
            .map(x -> x + value)
            .reduce(0, (x, y) -> x + y);
}

//Bonus:
// Parallel Stream Example:  Calculate the sum using a parallel stream
public Integer sumParallel() {
    return list.parallelStream().reduce(0, Integer::sum);
}


public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(17, 11, 19, 13, 43, 37, 41, 31, 29, 97, 5, 3,
        7, 23);

    StreamCalculator calc = new StreamCalculator(numbers);
    if (calc.streamToList(calc.getStream()).equals(numbers)) {
        System.out.println("getStream and streamToList work as intended");
    } else {
        System.out.println("getStream and streamToList failed");
    }

    calc.filter(2,30);
    System.out.println(calc.getList());

    calc.print(", ");

    System.out.println(calc.sum());

    calc.filter(2,10);
    System.out.println(calc.product());

    System.out.println(calc.addAndSum(20));
    calc.print("\n");

    List<Integer> randomNumbers = new ArrayList<>();

    //Bonus
```

```
        Random random = new Random();
        // Loop to generate 100 random numbers between 1 and 99
        for (int i = 0; i < 100; i++) {
            // Generate a random number in the range [1, 99]
            int number = random.nextInt(99) + 1; // nextInt(99) generates a number from 0 to
                98
            // Add the number to the list
            randomNumbers.add(number);
        }
        StreamCalculator calcBonus = new StreamCalculator(randomNumbers);
        if (calcBonus.sum().equals(calcBonus.sumParallel())) {
            System.out.println("sum parallel work as intended");
        } else {
            System.out.println("sum parallel failed");
        }
    }
}
```

Exercise 2

```java
import java.util.*;
import java.util.stream.*;

public class StreamCollector
{
    // Example Classes Lab Questions
    record Book(String title, String author, int releaseYear){}
    record Movie(String title, String director, int releaseYear){}

    // Instance Lists
    List<Book> books;
    List<Movie> movies;

    // Q1.a Constructor
    public StreamCollector(List<Book> books, List<Movie> movies)
    {
        this.books = books;
        this.movies = movies;
    }

    // Q1.b Getters

    public Stream<Book> getBookStream()
    {
        return books.stream();
    }


    public Stream<Movie> getMovieStream()
    {
        return movies.stream();
    }

    // Q2 Partitions By Release Year

    public void PartitionBooksByReleaseYear(int partitionYear)
    {
        Map<Boolean, List<Book>> partitionedBooks = getBookStream()
            .collect(Collectors.partitioningBy(book -> book.releaseYear > partitionYear));

        // Print the partitions
        System.out.println("Contemporary books");
        partitionedBooks.get(true).forEach(System.out::println);

        System.out.println("");

        System.out.println("Classics");
        partitionedBooks.get(false).forEach(System.out::println);
    }
```

```
public void PartitionMoviesByReleaseYear(int partitionYear)
{
    Map<Boolean, List<Movie>> partitionedMovies = getMovieStream()
        .collect(Collectors.partitioningBy(movie -> movie.releaseYear > partitionYear));

    // Print the partitions
    System.out.println("Contemporary movies");
    partitionedMovies.get(true).forEach(System.out::println);

    System.out.println("");

    System.out.println("Classics");
    partitionedMovies.get(false).forEach(System.out::println);
}

// Q3 Movies Based on Books

public void MoviesBasedOnBooks()
{
    System.out.println("\nMovies Based on Books\n");

    List<Movie> moviesBasedOnBooks = getMovieStream()
    .filter(movie -> getBookStream()
        .anyMatch(book -> movie.title.contains(book.title)))
    .collect(Collectors.toList());

    moviesBasedOnBooks.forEach(System.out::println);
}

// Q4 Grouping by creator

public void GroupMoviesByDirector()
{
    Map<String, List<Movie>> moviesByDirector = getMovieStream()
        .collect(Collectors.groupingBy(movie -> movie.director));

    moviesByDirector.forEach((director, movieList) -> {
        System.out.println(director + ": " + movieList);
    });
}

public void GroupBooksByAuthor()
{
    Map<String, List<Book>> booksByAuthor = getBookStream()
        .collect(Collectors.groupingBy(book -> book.author));

    booksByAuthor.forEach((author, bookList) -> {
        System.out.println(author + ": " + bookList);
    });
}

// Main
```

```
    public static void main(String args [])
    {
        // List of books
        List<Book> books = List.of(
            new Book("Murder on the Orient Express", "Agatha Christie", 1934),
            new Book("Death on the Nile", "Agatha Christie", 1937),
            new Book("Frankenstein", "Mary Shelly", 1818),
            new Book("Dracula", "Bram Stoker", 1896),
            new Book("Minor Detail", "Adania Shibli", 2017),
            new Book("Septology", "Jon Fosse", 2019)
        );

        // List of movies
        List<Movie> movies = List.of(
            new Movie("Murder on the Orient Express", "Kenneth Branagh", 2017),
            new Movie("Frankenstein", "James Whale", 1932),
            new Movie("Dracula", "Francis Ford Coppola", 1993),
            new Movie("Apocolypse Now", "Francis Ford Coppola", 1979),
            new Movie("The Pianist", "Roman Polanski", 2002),
            new Movie("Dune 2", "David Villeneuve", 2024),
            new Movie("Arrival", "David Villeneuve", 2016)
        );

        System.out.print("\n\nMain Function : Exercise 2\n\n");

        StreamCollector collector = new StreamCollector(books, movies);

        collector.PartitionBooksByReleaseYear(2003); // Q2.1
        collector.PartitionMoviesByReleaseYear(2003); // Q2.2
        collector.MoviesBasedOnBooks(); // Q3
        collector.GroupMoviesByDirector(); // Q4.1
        collector.GroupBooksByAuthor(); // Q4.2
    }
}
```