Java Programming
Lab Streams 1

Stream Basics

1. Lab objectives

In this lab you will write some basic code to filter, sort and print a list of arbitrary string objects using two different approaches: the traditional imperative approach, and then a functional approach using streams, with a few variations on some of the stream code thrown in.

2. Lab Setup

You can do this whole lab in one project with different packages for the imperative and the streams solutions

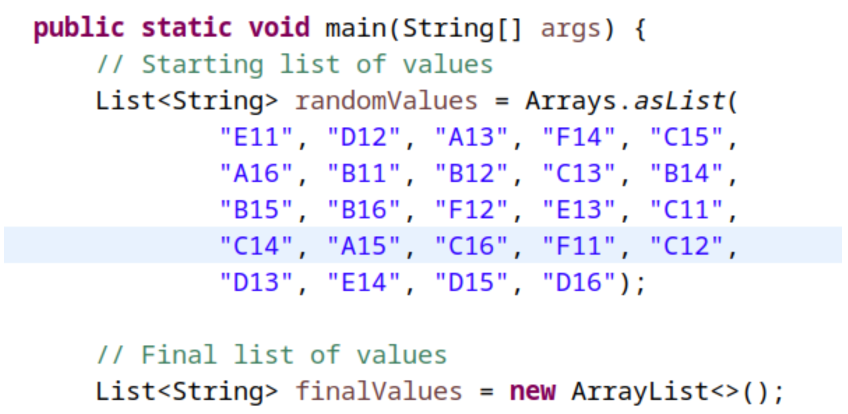
3. Lab Solutions

Solutions for each lab are provided in the repository in the labs directory for each module. These are not the only possible solutions and your solution to the labs may be a different in some ways than the reference solutions.

Since the demos are based on the labs, if you are having problems getting started, refer to the demo code, also in repository, to give you some inspiration.

Part 1: The Imperative Code

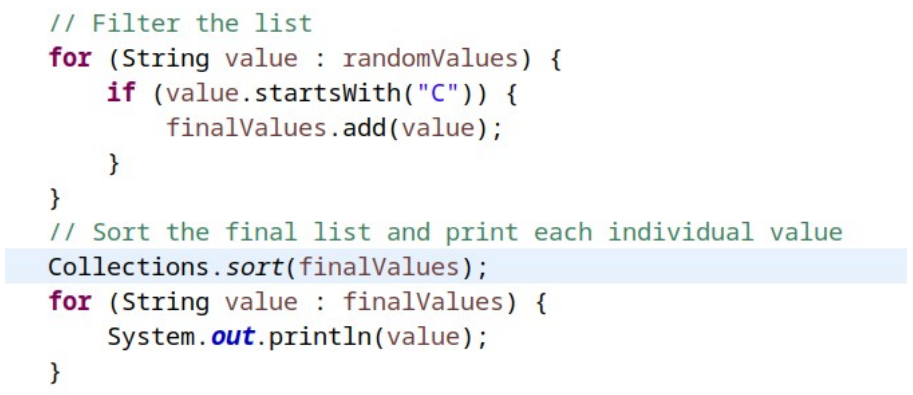
1. Create an `imperative` package (or you can call it whatever you like)
2. Add a `Main` class with a `main()` method. All you your code will be written inside of the main method so you don't need any other classes
3. Create a list of strings to use as the input data. You don't have to use exactly the data shown but you should have something similar.
4. Create an empty list to hold the results.



```java
public static void main(String[] args) {
    // Starting list of values
    List<String> randomValues = Arrays.asList(
            "E11", "D12", "A13", "F14", "C15",
            "A16", "B11", "B12", "C13", "B14",
            "B15", "B16", "F12", "E13", "C11",
            "C14", "A15", "C16", "F11", "C12",
            "D13", "E14", "D15", "D16");

    // Final list of values
    List<String> finalValues = new ArrayList<>();
```

5. Write some standard imperative code to select only the strings that start with 'C', sort the results and then print each element in turn. Note that we are using a built in sort method from the Collections API, otherwise the code would be even longer.



```java
// Filter the list
for (String value : randomValues) {
    if (value.startsWith("C")) {
        finalValues.add(value);
    }
}
// Sort the final list and print each individual value
Collections.sort(finalValues);
for (String value : finalValues) {
    System.out.println(value);
}
```

6. Test your code to ensure that it works.

Part Two: Using Streams

1. In a new package, in the solutions the package is named streams, create a `Main` class to hold a `main()` method and set up the initial list as before. However, you do not need a list of final values.

```
public static void main(String[] args) {

    List<String> randomValues = Arrays.asList(
            "E11", "D12", "A13", "F14", "C15",
            "A16", "B11", "B12", "C13", "B14",
            "B15", "B16", "F12", "E13", "C11",
            "C14", "A15", "C16", "F11", "C12",
            "D13", "E14", "D15", "D16");
```

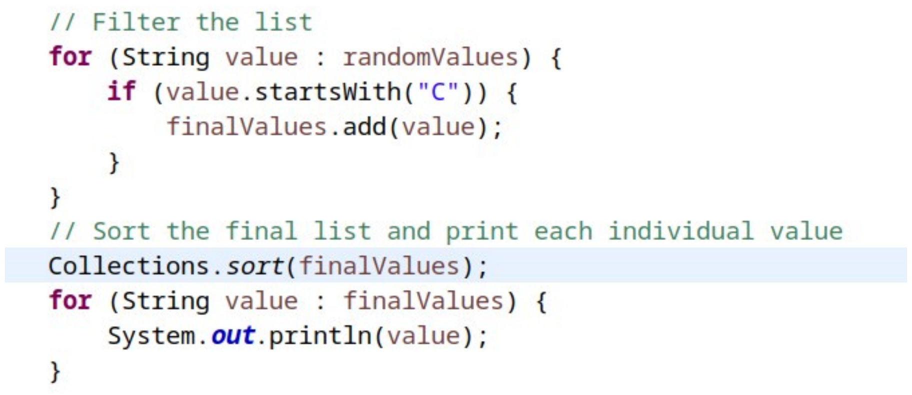
2. Now use the `.stream()` method to create a stream from the List
3. Add a `.filter()` pipeline method that uses a Lambda function to test whether or not to remove the data item from the stream
4. Then add a `.sorted()` pipeline method that sorts the stream
5. Finally, a `.forEach()` method applies the `println()` function in the `System.out` package to each of the elements in the stream and terminates the stream.
6. As an alternative to the `foreach(System.out:println)` is to provide a Lambda function that does the same job in a little more traditional programming format – which you use is matter of taste.

```
randomValues.stream()                       // Create the streams
.filter(value->value.startsWith("C"))       // Filter pipeline method
.sorted()                                   // Sort pipeline method
.forEach(System.out::println);              // Terminal forEach
// .forEach(x -> System.out.println(x));
}
```

7. Test your code to see that it works

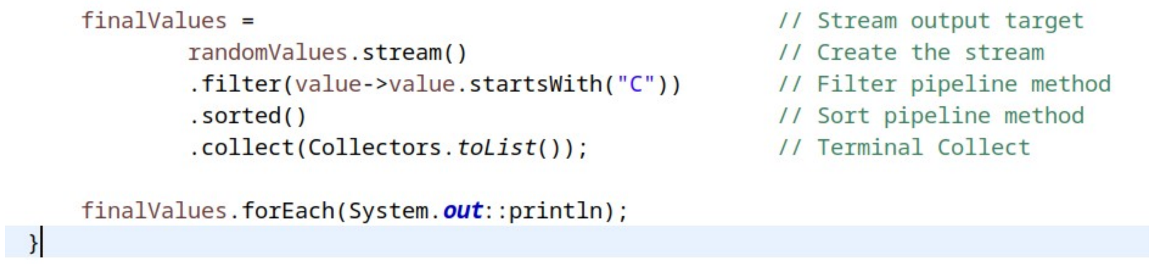
Part Three: Terminal Collection

1. In this section, you change the terminal action to return a collection. You can modify the code from the last part for this part. In the solutions, this code is in the package `streams2`.
2. You will need to add a list to hold the final results.



```
// Filter the list
for (String value : randomValues) {
    if (value.startsWith("C")) {
        finalValues.add(value);
    }
}
// Sort the final list and print each individual value
Collections.sort(finalValues);
for (String value : finalValues) {
    System.out.println(value);
}
```

3. Since the collector will produce a list, the result of the whole stream pipeline has to be assigned to a variable of the appropriate type.
4. To create a collection, replace the `.forEach()` function with the function collector function `.collect(Collectors.toList())`



```
    finalValues =                                               // Stream output target
            randomValues.stream()                               // Create the stream
            .filter(value->value.startsWith("C"))               // Filter pipeline method
            .sorted()                                           // Sort pipeline method
            .collect(Collectors.toList());                      // Terminal Collect

    finalValues.forEach(System.out::println);
}
```

5. Note that the stream no longer prints out the elements so we have to add the extra line to do that.
6. Test your code to ensure that it works.

Part Four: Challenge Section

In this section, a problem statement will be given. Without any coding hints, you will write the appropriate Java stream code to solve the problem. This section assumes you are comfortable working with Java Collections as well as writing Java code.

You will start with a collection of three character strings representing currency abbreviations. A list of input data you can use is in the lab resources folder for Lab 1 in the repository. Write a pipeline that does the following:

1. Convert any instance of “GBP” (Pounds Sterling) to “EUR” (Euro). The input strings may be in any combination of upper and lower case.
2. Remove any element that is not one of: USD","CDN","EUR","JPY" (US and Canadian dollars, Euros and Yen)
3. Print out a frequency count for each of the valid currencies in the stream.

Some suggestions

1. Use a `Java Set` to hold the reference currencies in point 2 above
2. Use a `Java Map` to hold the frequency count
3. Break down the process into a series of transformations of the streams
4. The terminal operation should update the frequency count map

The solution provided in one possible solution. Your solution is correct if you get the frequency count:

```
{JPY=2, EUR=5, USD=4, CDN=3}
```

However, even if you get the right result you should be striving for code that is elegant and easy to understand.

End of Lab