Streams

In this workshop you will build a few simple applications that make heavy use of lambda expressions, method references, and the Stream API. Starter code for all three applications is found in the project **Streams**. Each project occupies a different Java package in that project.

Word Counts

Take a look over the file **Speech.txt**. You'll be reading this file and developing some statistics about word counts and distributions, so you might want to keep it handy.

Create a class with a **main** method that reads this file as new **FileInputStream**.

Now, develop a static method **countWords** that takes an **InputStream** and returns a string. At the top of this method create a try-with-resources block that builds a **BufferedReader** on an **InputStreamReader** that is based on the given **InputStream**. Call **lines** on this reader, which gives you a stream of strings. Call **collect** on that stream, passing the **Collectors.joining()** collector with "\n" as the separator, to reconstitute the full text as a string called **contents**.

Create a helper method **toWords** that takes a string and returns a stream of strings. In it, call **split** on the given string, passing the following regular expression as the delimiter:

```
[\\s\\.,]+
```

This will yield an array of individual words, excluding all whitespace and any leading or trailing punctuation. Derive a stream of the words in that array. Filter the stream so that it excludes the string "—" (which does get past that initial regular-expression split), and any words shorter than three letters.

Now, in **countWords**, use that helper method to derive the following statistics, and format them into a report that you return as a string:

- A count of all words: expected count is 214
- A count of all distinct words – with case-insensitive comparisons for this purpose – expected count is 123
- The average word length of all words – 4.87
- The count of distinct words that begin with each letter of the alphabet – expected output as shown below:

```
A     9
B     6
C     9
D     7
E     4
F    13
G     5
H     5
I     1
L     8
M     4
N     8
O     1
P     8
R     5
S     7
T    11
U     2
V     1
W     8
Y     1
```

Road Race

In the **race** package you'll find a JavaBean **Result** that represents the results in a race for a single runner: name, age, and finishing time in minutes and seconds, and it has a read-only property that reports the total time in seconds. A starter application class **Analysis** has a couple of helper functions for formatting total seconds as minutes and seconds, and a **main** method with **//TODO** comments that ask you to analyze the results of a given race in various ways. The results themselves are in a file **Times.csv** available in this folder.

Start by loading the data from the file: call **Files.lines**, passing **Paths.get("Times.csv")**, and **map** the resulting stream of text lines by calling **split** on each; then **map** again, this time by passing the four elements in the array to the constructor of a new **Result** object. **collect** the resulting stream of results into a list, and capture that for repeated use in answering the remaining questions posed in the **//TODO** comments.

Now use the Stream API to find the requested data. There are multiple ways to solve each of these . A few tips:

- Remember that, when mapping from an object to a numeric field (integer or floating-point), you must **mapToInt** or **mapToDouble**, and not just **map**. These methods give you an **IntStream** or a **DoubleStream**, rather than a **Stream<T>**. To map from numbers to objects, use **mapToObj**.
- To get age groups, the simplest approach is to provide a key function that returns the age integer-divided by 10. So the key 2 would represent runners ages 20-29.
- Remember that many aggregate functions such as **average** return an **Optional<T>**, not a value of type **T**. This is just in case the stream was empty when averaged; but for your code to compile you'll need to call **get** on that optional when deriving the aggregate value.
- Do as much as you reasonably can with **Stream** and related classes; but remember that traditional collections can still be useful! and most modern Java code that processes complex information bounces between streams, collections, and even iterators, as each has its strengths.

Human Resources

In the **hr** package you'll find a data model of three classes – **Job**, **Department**, and **Employee** – and a **Data** class that stands in for a more complete and external database, with instances of all three classes already compiled into ID-keyed maps and already wired to one another. Employees have jobs and belong to departments, and departments know their employees.

There is a starter **Application** with a helper function **report** that makes it easy to produce the contents of a stream to standard output, and a **main** method that instantiates the in-memory **Data**. Then, again, there are //TODO comments that ask you to find specific things from the HR data. The first requirement is already met.

Some tips and clarifications:

- By "payroll" we just mean the sum of employees' salaries – per department in one case, and for the whole company in another. To "normalize" a salary is to increase it to the minimum for that employee's job description, or to decrease it to the maximum – to bring it into the allowable range.
- There is a subtle difference between the last two requirements. The penultimate one asks you to calculate the "impact," plus or minus, on total payroll, if salaries were to be normalized. **map** is a good way to do this, and of course you won't change the data while doing so. The last one asks you to make the changes, actually modifying the employee objects as part of stream processing. Consider the **peek** method for this – it's something like **forEach** but it isn't terminal, so you can continue to map, filter, sum, etc., after using a lambda or method reference to modify certain employees.

Correlation

Use the provided **com.amica.streams.Correlation** class to determine the correlation coefficient between the age in years and time in seconds of the runners in the **RoadRace** exercise. The **fromInts** method takes a collection of a specific type and two **Function**s that translate objects of that type to integers. Of course **results** is your collection, and remember that an easy way to provide a function that derives a single property from an object is to refer to the getter method for that property – you don't even need to write out a lambda expression.

Expected output is 0.9610.

Frequency Analysis

In **com.amica.cipher** the **Substitution** class implements substitution ciphers and a method to crack an unknown substitution cipher by frequency analysis. The existing implementation uses the Collections API and procedural programming to find the most common letters in an encoded string, and then map those to a known list of common letters in the original string, so as to decode the string.

Two methods present themselves as candidates for refactoring using the Stream API. The simpler one is **translate**, which ultimately is a map from one set of characters to another. You might want to develop a helper method that translates a single character, since this is more than just getting the corresponding character from the given cipher map: you have to pass unknown characters, verbatim.

The second is the **cipherFromFrequencies** method. There are two phases to the process: first we build a map of counts of letters, and then we descend through the map entries by count and build a final map of encoded letter to original letter. It is possible to carry that whole process out in a single chain of stream methods – although not really in a single pass through a stream chain. That is, you can stream your way to the intermediate map of counts; and then you can immediately take its **entrySet** and stream your way from that to the final cipher map. So it may in its final form look like the fluent style of streaming code, and for the most part it will be; but at runtime there still is an intermediate collection being collated in memory.

Give it a try! This is an unstructured challenge in Stream coding, and you will have to try a few things and feel your way along with the API.