Activities analyzer

Vlad-Cristian Buda 30423

Table of Contents (no numbering)

1. Objective
2. Analysis of the problem
   1. Assumptions
   2. Modeling
   3. Use cases
   4. Errors
3. Design
4. Implementation
   1. Class MonitoreData
   2. Class Start
5. Results
6. Conclusions
7. Biobliography

1. Objective

The main objective of this project is to develop a Java program which analyzes the data recording the activities of a person during a period of time. Data is recorded in a text file and passed to the program.

The process of the data inside the file will be made using Java 8 streams and Lambda expresions. First of all, in order to process the given data, a suitable representation inside the program is needed. The given information, i.e. the start date, end date, and the activity name will be stored in a separate class, containing a string field for each characteristic.

2. Analysis of the problem

The solution of the problem should be able to perform the following tasks on the given data: count how many days of monitored data appears in the log; count how many times has appeard each activity over the entire monitoring period; count how many times has appeared each activity for each day over the monitoring period; for each line from the file map the activity labvel the duration recorded on that line (end time – start time); for each activity compute the entire duration over the monitoring period; filter the activities that have 90% of the monitoring records with duration less than 5 minutes.

a) Assumptions

The nature of the problem suggests that a small number of assumptions can be made. Although, we assume that the input file has the correct format (start date with the valid syntax, end date with the valid syntax and the acitvity name). The output for each requirement is saved in a separate .txt file with a sugestive name. We also assume that the user understands the meaning of each output file and knows how to interpret the results.

b) Modeling

As mentioned above, the input data must be represented in a suitable form. With this in mind, class MonitoredData was created in order to store the start date, end date and activity name for each line of the input data file. Besides this, the required operations are implemented in a separate class where the main function also resides.

c) Use cases

The use cases for this program are straightforward. Once the user runs the programs, a .txt file is created with the output for each required operation. No input other that the file containing the data to be analyzed is necessary.

d) Errors

If the user respects the assumptions, no errors should appear during the execution of the program. Therefore, once the input file has the specified format, is safe to say that the program is error free.

3. Design

The design of the application is made in an object oriented manner. As mentioned earlier, data read from the input file is represented in a List of objects of type MonitoredData (the class which stores the data to be processed). Specific to this application will be the use of the new features introduced by Java 8.

**Stream API** is used for data processing which supports parallel operations. It enables data processing in a declarative way. Streams are sequences of elements that support different operations. Streams are lazily computed on demand when elements are needed. The stream is like a recipe that gets executed when actual result is needed. Following are the characteristics of a Stream:

* **Sequence of elements** − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
* **Source** − Stream takes Collections, Arrays, or I/O resources as input source.
* **Aggregate operations** − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
* **Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
* **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Stream operations are divided into ***intermediate*** and ***terminal*** operations combined to form stream pipelines. Intermediate operations return a new stream. Executing an intermediate operation such as ***filter()*** does not actually perform any filtering, but instead creates a new stream. Terminal operations on the other hand, such as ***collect()***generates a result or final value. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used. Intermediate and terminal operators, such as ***limit()*** or ***findFirst()*** can be short-circuiting, once they achieve their goal they stop further stream processing. Intermediate operations are further divided into ***stateless*** and ***stateful*** operations. Stateless operations, such as ***filter()*** and ***map()***, retain no state from the previously seen element when processing a new element, hence each element can be processed independently of operations on other elements. Stateful operations, such as ***distinct()*** and ***sorted()***, may incorporate state from previously seen elements when processing new elements. For example, one cannot produce any results from sorting a stream until one has seen all elements of the stream. As a result, under parallel computation, some pipelines containing stateful intermediate operations may require multiple passes on the data or may need to buffer significant data. Stateful operations should be carefully considered when constructing stream pipeline because they might require significant resources.

With Java 8, Collection interface has two methods to generate a Stream.

* **stream()** − Returns a sequential stream considering collection as its source.
* **parallelStream()** − Returns a parallel Stream considering collection as its source.

**Lambda expression** is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface (an interface which has only one abstract method) . It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code. Java lambda expression is treated as a function, so compiler does not create .class file.

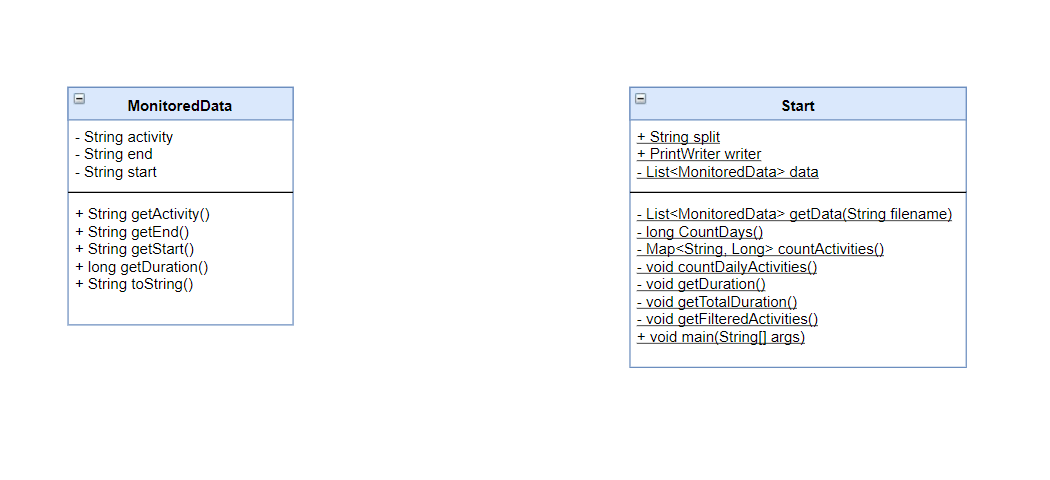
A lambda expression is characterized by the following syntax:

parameter -> expression body

Following are the important characteristics of a lambda expression:

* **Optional type declaration** − No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
* **Optional parenthesis around parameter** − No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
* **Optional curly braces** − No need to use curly braces in expression body if the body contains a single statement.
* **Optional return keyword** − The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

The following UML diagram shows the entire structure of the program along with all the classes, interfaces and relationships between them.



As we can see from the UML diagram, there are only two classes. As mentioned earlier, data read from the file is represented in the class MonitoredData. Class Start contains the operations required to be performed on the set of data.

When in comes to data representation, each line from read from the file will be stored in an object of type MonitoredData. Then, each object is inserted in a List of MonitoredData objects and all the required operations will be performed on that list.

All the main classes will be explained in detail in the following section along with their implementation and main methods

4. Implementation

In what follows, we’ll explain each class in detail along with the attributes and methods contained.

Class MonitoredData stores the information read from a line of the input file. The information extracted from a line is the following: start time of the activity (stored in a String object), end time of the activity (stored in a String object), and the name of that activity. Besides this, the class also contains getter methods for obtaining information regarding to each activity. The method getDuration() returns the duration of that event, computed in minutes, based on its starting and ending time.

Class Start contains the implementation of each requirement. The class main field is the list of MonitoredData, constructed based on each line of the input file. The main methods of this class which also implements the functionality will be explained below.

Method List<MonitoredData> getData(String filename) is the first method called when the program starts. It reads the input file using streams and maps each line to an object of type MonitoredData after splitting the line. It also uses a collector to create the list of the specified type of objects and returns it. The code implementing the functionality is listed below:

Method long countDays() returns the total number of monitored days. It creates a stream out of the list of MonitoredData, saves the starting date of the previous activity and filters the current activity based on a comparison made on the start date of the previous activity and the end date of the current one. If those are different, the stream is filtered and also each entry is counted, returning a long value containing the number of days.

Method Map<String, Long> countActivities() returns a Map containing for each activity the number of times it has appeared over the monitored data. It also uses streams to iterate over the list of objects and creates a group for each activity where the count result of how many times it appears is placed.

Method void countDailyActivities() counts the number of times each activity has appeared in a day. It is similar to the previous method. It uses a Map where the key is the date and the value is represented by another Map with activity name as the key and the number of occurances the value it maps to.

Method void getDuration() calculates the duration of each activity as it appears on each line of the input file. It uses the getDuration() method to compute the duration of that activities and collects the results in a list of long values.

Method void getTotalDuration() calculates the duration of each activity over the total monitored period. It still uses method getDuration() to get the duration of that activity and groups it in a map, where the key is the activity name and the value is the total duration computed in minutes.

Method void getFilteredActivities() filters the activity with the duration time in 90% of the recording smaller than 5 minutes. It computes the total number of occurances of each activity using the previous function and the total number of occurances for each activity with duration smaller than 5 minutes. Then it stores the result in a list of strings containing the name of the acitvities which respects the condition.

5. Results

After all the necessary steps (design, implementation) we managed to obtain a fully functional application which performs the necessary requierements. After the user runs the program, a text file containing the output of each requirement is created on the disk. The result for each function is stored in that file.

7. Conclusions

During the design and implementation of this application, I was able to learn some new things regarding Java programming. For example, it is the first time I worked the new features introduced by Java 8: streams and lambda expressions. I was able to get a hands-on experience with this tools and learn new features.

8. Biobliography

<https://stackoverflow.com> (for various implementation questions)

<http://www.mkyong.com/tutorials/java-8-tutorials/>

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

<https://www.draw.io/> (for drawing the UML diagram)

Java: How to Program, 9th Edition (Deitel) (for Java, GUI and UML recap)