Processing sensor data of daily living activities

Rad Lidia

Group 30424

UTCN

Table of contents:

1. Objective

The principal objective is to design an application for analysing the behaviour of a person recorded by a set of sensors, using lambda expressions and stream processing. The historical log of the person's activity is stored as tuples (start_time, end_time, activity_label), where start_time and end_time represent the date and time when each activity has started and ended while the activity label represents the type of activity performed by the person: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare_Time/TV, Grooming. The data is spread over several days as many entries in the log Activities.txt given.

In order to obtain this functionality, the main problem can be decomposed into several steps needed to reach the goal:

• Creating a stream out of the given Activities.txt file, in order to read the data.

• Choosing the right structure of data needed.

• Implementing algorithms to process the data coming from the stream, in order to achieve the desired results.

• Display the results nicely, so that the user can read it easily.

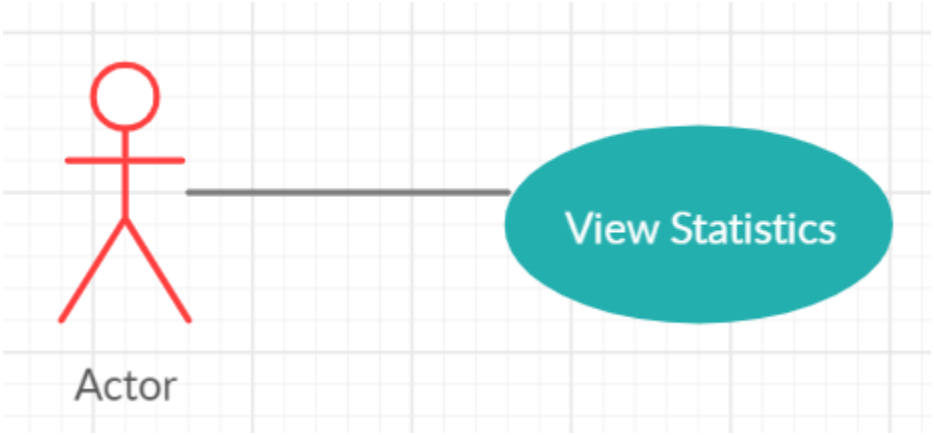
2. Analysis of the problem

This application is built to analyse the data received Activities.txt file, regarding the activities performed by a person during a certain period of time. The file is organised as follows: each line contains information about one activity: start time, end time, and activity label. The Activities.txt file is created and then filled with data by the sensors.

The main goal of the application is to manage this input data in order to compute certain states (e.g. the total time spent showering, the total time in a day spent as Spare_Time/TV, how many days the sensors tracked the activities).

a) Use Cases

The use-case diagram is minimum:



b) Modeling

In order to design an application which fulfills the above mentioned description we have to understand first how streams can be useful in an application in which data is abundant, and also how lambda expressions work.

Stream:

A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements: stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result.

In the example below, filter, map and sorted are intermediate operations whereas forEach is a terminal operation. Such a chain of stream operations as seen in the example below is also known as operation pipeline.

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);

// C1
// C2
```

Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behaviour of the operation. Most of those operations must be both non-interfering and stateless. A function is non-interfering when it does not modify the underlying data source of the stream, for example in the above example, no lambda expression does modify myList by adding or removing elements from the collection.

A function is stateless when the execution of the operation is deterministic, for example in the above example no lambda expression depends on any mutable variables or states from the outer scope which might change during execution.

Lambda expressions:

A lambda expressions provides a shorthand notation for creating an instance of an anonymous inner class implementing a functional interface (single-abstract-method interface).

Lambda expressions allow us to create instances of classes with a single method in a much more compact way.

- The *lambda expression* consists of the method parameter and the method body, separated by ->.
- The *lambda expression* is an *expression* which evalutes to a value. It evaluates to an instance of an anonymous inner class implementing a functional interface.
- The name of the functional interface can be deduced from the type declaration. The parameters-type and return-type of the method can also be deduced, because there is only one method in the functional interface.

3. Design

The application was designed for analysing the behaviour of a person recorded by a set of sensors. The purpose is to manage this input data in order to compute certain states (e.g. the total time spent showering, the total time in a day spent as Spare_Time/TV, how many days the sensors tracked the activities). Next, we will present the data structures used for achieving this purpose.

Data Structures:

Data structures used:

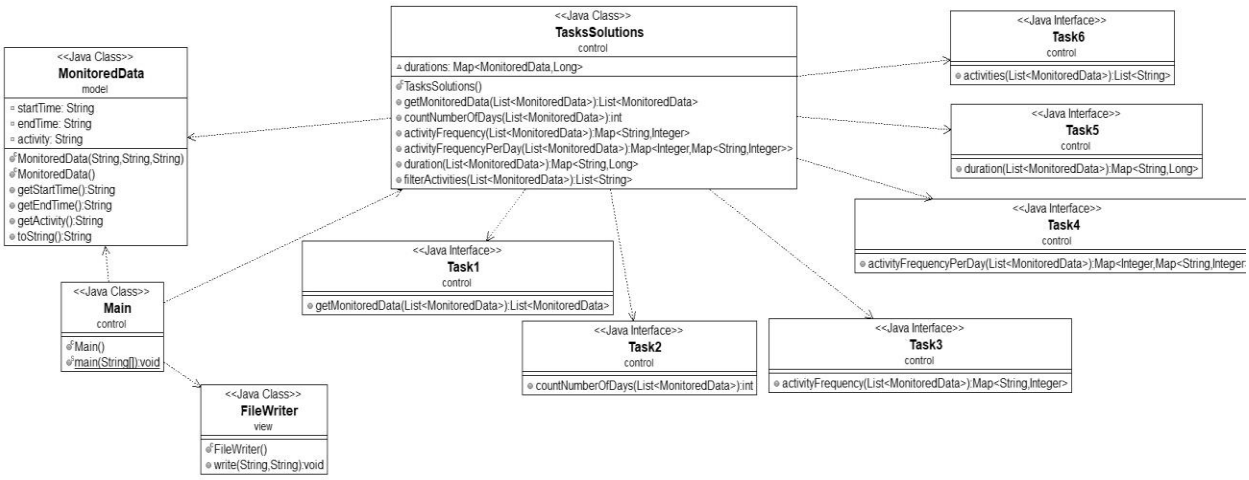
- a List of monotoredData for storing the activities information of the log;

- a Map<String, Integer> structure representing the mapping of each distinct activity to the number of occurrences in the log;

- a Map<Integer, Map<String, Integer>> that contains the activity count for each day of the log; therefore the key of the Map represent an Integer object corresponding to the number of the monitored day, and the value will represent a Map - in this map the String key corresponds to the name of the activity, and the Integer value corresponds to the number of times that activity has appeared within the day;

-a Map<String, Long> in which the key of the Map represents a String object corresponding to the activity name, and the value represents a Long object corresponding to the entire duration of the activity over the monitoring period.

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



The design of this app follows a MVC design architecture, even if we don't have an interface, the view package will contain the FileWriter class, that generate the .txt files required. So, we have 3 packages: model, view, control.

The model package is represented only by the class MonitoredData that describes the logic of the monitored data.

The view package contains the WriteFile class, as mentioned before, that generates the .txt files required.

The control package contains the interface for each 6 tasks, the class TasksSolutions that implements these interfaces and the main class.

4. Implementation

The project contains four classes and 6 interfaces, for each tasks. Next, we will explain each class and the important methods contained.

- View package:

The FileWriter class has the write method with the parameters: the name of the file where the solution of the task will be printed and the taskInfo - the information of the task solutions that will be printed.

```java
package view;

import java.io.PrintWriter;

public class FileWriter {
    public void write(String filename, String taskInfo) {
        try {
            PrintWriter file = new PrintWriter(filename, "UTF-8");
            file.write(taskInfo);
            file.close();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

- Model package:

MonitoredData class has fields: startTime, endTime and activity, all represented in String objects. Also, it has getters, for getting the end time, start time and activity name and toString() method to nicely print a specific monitored activity.

```java
package model;

public class MonitoredData {
    private String startTime;
    private String endTime;
    private String activity;

    public MonitoredData(String startTime, String endTime, String activity) {
        this.startTime = startTime;
        this.endTime = endTime;
        this.activity = activity;
    }
```

- Control package:

Contains six interfaces for each task, named Task1, Task2 and so on. Task1 for example looks like this:

```java
package control;

import java.util.List;

public interface Task1 {

    List<MonitoredData> getMonitoredData(List<MonitoredData> monitoredData);

}
```

TasksSolutions class implements each of these interfaces. This class has six methods for implementing each single-abstract-method of the interface.

Method getMonitoredData(List<MonitoredData> monitoredData) implements the first task that requires to read the data from the file Activity.txt using streams and split each line in 3 parts: start_time, end_time and activity_label, creating a list of objects of type

MonitoredData. We define an anonymous inner class implementing the interface, construct an instance and invoke the method, by using lambda expression as a shorthand.

Two lists are created, one that will be returned “monitoredData” and another that will contain String objects that are read from the file “activityList”. For each String, the separation is made in three Different strings, using the split() function. Using the constructor, the m object of type MonitoredData is created and added to the “monitoredData list. The method used to read from the file throw an exception of type IOException, an exception that is intercepted in the catch clause if is the case.

```
20    public List<MonitoredData> getMonitoredData(List<MonitoredData> monitoredData) {
21
22        Task1 obj = (activity) -> {
23            String fileName = "C:\\Users\\Lidia\\eclipse-workspace\\Processing sensor data\\Activities.txt";
24            List<String> activityList = new ArrayList<String>();
25            try (Stream<String> stream = Files.lines(Paths.get(fileName))) {
26
27                activityList = stream.collect(Collectors.toList());
28                for (int i = 0; i < activityList.size(); i++) {
29                    String data = activityList.get(i);
30                    String[] aux = data.split("\t\t");
31                    MonitoredData m = new MonitoredData(aux[0], aux[1], aux[2]);
32                    monitoredData.add(m);
33                }
34
35            } catch (IOException e) {
36                e.printStackTrace();
37            }
38            return monitoredData;
39        };
40        return obj.getMonitoredData(monitoredData);
41    }
```

Method countNumberOfDays(List<MonitoredData> monitoredData) implements the second task that requires to count the distinct days that appear in the monitoring data. In order to do that, a list “listOfStartTime” that will contains the start dates of each activity is built. This is done by mapping by start time and then extracting the date by taking the String that starts at position 0 and ends at position 10. Then, on the list obtained the distinct() function is applied to obtain only the distinct data, data that will be counted later using the count() function.

```
43    public int countNumberOfDays(List<MonitoredData> monitoredData) {
44        Task2 obj = (activity) -> {
45            List<String> listOfStartTimes = monitoredData.stream().map(m -> m.getStartTime()).map(s -> s.substring(0, 10))
46                    .collect(Collectors.toList());
47            long numberOfDays = listOfStartTimes.stream().distinct().count();
48            return (int) numberOfDays;
49        };
50        return obj.countNumberOfDays(monitoredData);
51    }
```

Method Map<String, Integer> activityFrequency(List<MonitoredData> monitoredData) implements the third task that requires to count how many times each activity has appeared over the entire monitoring period. This is done by creating a list of all distinct activities, using the stream operation map by activity and the distinct() method. For each activity in the list of distinct activities, a long type value is created that will contain the number of occurrences. This is made by mapping on the activity and using a filter that will test if the current activity in the initial list is

equal to the activity for which the count is performed. The activities remaining after filtering are counted, the value obtained being inserted in the Map.

```
public Map<String, Integer> activityFrequency(List<MonitoredData> monitoredData) {
    Task3 obj = (activity) -> {
        HashMap<String, Integer> activityFrequency = new HashMap<String, Integer>();
        List<String> listOfActivities = monitoredData.stream().map(m -> m.getActivity()).collect(Collectors.toList());
        for (String a : listOfActivities) {
            long count = monitoredData.stream().map(m -> m.getActivity()).filter(currentActivity -> a.equals(currentActivity))
                    .count();
            activityFrequency.put(a, (int) count);
        }
        return activityFrequency;
    };
    return obj.activityFrequency(monitoredData);
}
```

Method Map<Integer, Map<String,Integer>> activityFrequencyPerDay(List<MonitoredData monitoredData) implements the fourth task that requires to count for how many times each activity has appeared for each day over the monitoring period. The method starts by creating two lists that will contain the distinct start dataof each activity (done as in Task 2) and the distinct activities (as in Task3). For each date in the listOfStartTimes, take the integer number of day, scroll through the list distinct activities and count the number of appearances. This is done by applying a filter that will check the start time of the activity is equal to the current day and if the activity is equal to the current activity. Data left in after filtering are counted. Then number of day, name of the activity and the number of occurances will be added to the activityFrequencyPerDay Map required.

```
    public Map<Integer, Map<String, Integer>> activityFrequencyPerDay(List<MonitoredData> monitoredData) {
        Task4 obj = (activity) -> {

            //nr of day, activity name, nr of times
            Map<Integer, Map<String, Integer>> activityFrequencyPerDay = new HashMap<Integer, Map<String, Integer>>();
            List<String> listOfStartTimes = monitoredData.stream().map(m -> m.getStartTime()).map(s -> s.substring(0, 10))
                    .distinct().collect(Collectors.toList());
            List<String> listOfActivities = monitoredData.stream().map(m -> m.getActivity()).distinct()
                    .collect(Collectors.toList());
            for (String day : listOfStartTimes) {
                Map<String, Integer> activityFrequency = new HashMap<String, Integer>();
                String[] aux = day.split("[- ]+");
                int dayInt = Integer.parseInt(aux[2]);
                for (String currentActivity : listOfActivities) {
                    long count = 0;
                    count = monitoredData.stream()
                            .filter(n -> n.getStartTime().contains(day) && n.getActivity().equals(currentActivity)).count();
                    activityFrequency.put(currentActivity, (int)count);
                }
                activityFrequencyPerDay.put(dayInt,activityFrequency);
            }
            return activityFrequencyPerDay;

        };
        return obj.activityFrequencyPerDay(monitoredData);
    }
```

Method Map<String, Long> duration() implements the fifth task that requires to compute the entire duration over the monitoring period for each activity. We perform a simple mapping using lambda expressions and the toMap method. The getTime method gives the time in milliseconds and therefore will be divided by 1000 to obtain the time in seconds. The duration is calculated as the

difference between the end time and the start time in seconds. Assigning a-> a makes sense because the stream traverses MonitoredData elements and saves objects of the same type as keys. Then, we go through each pair of form <MonitoredData, Long> and we will determine if there are equal keys. If so, the values will be added using summingLong method from Collectors and map on the first key, the other being deleted.

```
94      Map<MonitoredData, Long> durations;
95      public  Map<String, Long> duration(List<MonitoredData> monitoredData) {
96          Task5 obj = (activity) -> {
97              SimpleDateFormat date = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
98              durations = monitoredData.stream()
99                      .collect(Collectors.toMap(a->a,(a->{
00                          try {
01                              return date.parse(a.getEndTime()).getTime()/1000-date.parse(a.getStartTime()).getTime()/1000;
02                          } catch (ParseException e) {
03                              e.printStackTrace();
04                          }
05                          return null;
06                      })));
07              Map<String,Long> durationsGlobal = durations.entrySet().stream()
08                      .collect(Collectors.groupingBy(entry-> entry.getKey().getActivity(),
09                              Collectors.summingLong(entry->entry.getValue())));
10              return durationsGlobal;
11          };
12          return obj.duration(monitoredData);
13      }
```

The last method List<String> filterActivities() implements the last task that requires to filter the activities that have more than 90% of the monitoring records with duration less than 5 minutes. We start by obtaining a list of distinct activities. Applying a filter to the list we obtained only those activities that last less than 5 minutes. Then, for each distinct activity, we calculate the total number of appearances and the total number of appearances in the list of the activities already filtered. If these 2 numbers are not zero, if the current activity is equal to the activity in the list and if the number of existing activities after filtering is greater than or equal to 90% of the total number of activities, then the activity is put in the required activities List<String>.

```
115     public List<String> filterActivities(List<MonitoredData> monitoredData) {
116         Task6 obj = (activity) -> {
117             List<String> activities = new ArrayList<String>();
118             List<String> listOfActivities = monitoredData.stream().map(m -> m.getActivity()).distinct()
119                     .collect(Collectors.toList());
120             SimpleDateFormat date = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
121             List<MonitoredData> result = monitoredData.stream().filter(m -> {
122                 try {
123                     return 5 > TimeUnit.MINUTES.convert(
124                             date.parse(m.getEndTime()).getTime() - date.parse(m.getStartTime()).getTime(),
125                             TimeUnit.MILLISECONDS);
126                 } catch (ParseException e) {
127                     System.out.println("Error at parsing");
128                 }
129                 return false;
130             }).collect(Collectors.toList());
131             for (String a : listOfActivities) {
132                 long totalAppearances = monitoredData.stream().filter(act -> act.getActivity().equals(a)).count();
133                 long totalNumberAfterFilter = result.stream().filter(act -> act.getActivity().equals(a)).count();
134                 activities = result.stream()
135                         .filter(r -> totalAppearances != 0 && totalNumberAfterFilter != 0 && r.getActivity().equals(a)
136                         && totalNumberAfterFilter >= 0.9 * totalAppearances)
137                         .map(r -> r.getActivity()).distinct().collect(Collectors.toList());
138             }
139             return activities;
140         };
141         return obj.activities(monitoredData);
142     }
```

The Main Class calls all these methods described above, the results obtained being displayed in a text file, using FileWriter.

5. Results

After all the necessary steps, design, implementation, we managed to obtain all the statistics required. By running the application, the result of all the processing can be seen in the 6 text files generated by the program.

6. Conclusions

During the making of this assignment I improve my Java programming skills, by learning how use functional programming in Java with lambda expressions and stream processing, I noticed how much the size of the code is reduced if they areused.

Future improvements of the project can be:

- method that calculates the percentage of occurrence of each activity on monitoring period

- method that will determine the average duration of each activity.

- designing graphics based on methods that are already implemented

7. Biobliography

https://mkyong.com/java8/java-8-stream-read-a-file-line-by-line/ (how to read line by line with stream)

https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/ (for learning how to use streams)

https://www.java67.com/2014/04/how-to-make-executable-jar-file-in-Java-Eclipse.html

(for generating .jar file)

https://app.creately.com/diagram/OI01BYOepkA/edit (for use case diagram)

https://stackoverflow.com (for various questions)