*Technical University of Cluj-Napoca*

*Faculty of Automation and Computer Science*

*2nd Semester 2019-2020*

**Programming Techniques**

**Homework 5**

Student: Osváth Tamás

Group: 30421

Teacher: Pop Cristina

Contents

[1. Objectives 3](#_Toc40701452)

[2. Problem Analysis, Modelling, Scenarios 3](#_Toc40701453)

[2.1. Problem Analysis 3](#_Toc40701454)

[2.2. Modelling 3](#_Toc40701455)

[2.3. Scenarios 4](#_Toc40701456)

[3. Design 4](#_Toc40701457)

[3.1. Class Design, Packages, Relationships and UML Diagrams 4](#_Toc40701458)

[3.2. Algorithms and Data Structures 5](#_Toc40701459)

[4. Implementation 6](#_Toc40701460)

[5. Usage and Testing 10](#_Toc40701461)

[6. Conclusions 10](#_Toc40701462)

[7. Bibliography 11](#_Toc40701463)

# 1. Objectives

The main objective of the assignment is to use Lambda Expressions and Stream processing for analysing the behaviour of a person.

Consider task of analyzing the behaviour of a person recorded by a set of sensors installed in its house. 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, taken from [1-2] and downloadable. Write a program that uses functional programming in Java with lambda expressions and stream processing to perform the tasks listed in the table below

# 2. Problem Analysis, Modelling, Scenarios

## 2.1. Problem Analysis

Working with sets of data registered into text files is used in many fields of programming, in many situations it is a necessary and a useful concept. Input data can be easily managed by loading them from text files, and results are useful to store in text files, because it is easy to read and analyze (for example logging or other generated or simulated data). Working with streams and lambda functions allow us functional programming, also they make us more familiarized with the Java 8 newly introduced features.

Tracking a person’s actions and activities helps us providing statistics and information about the person’s habits. Analyzing the provided data can help in scheduling, self-organizing or health of a person.

## 2.2. Modelling

In order to be able to work with the data stored in the input file, it is necessary to create a way in which it will be processed. Then, the processed data should be stored somewhere for further operations. Finally, after some operations, the results should be transmitted to an output file.

It is straightforward that minimum two classes are needed, one for storing a single piece of data, and one for implementing methods that filter the read data.

In order to process the data, lambda functions and streams are used.

## 2.3. Scenarios

There is a single scenario: the data is read from the input file, some operations are given in the Main class and the output for each task is written to an output file, in txt format.

# 3. Design

## 3.1. Class Design, Packages, Relationships and UML Diagrams

Since this is a simple project, without a graphical user interface, dynamical input or other features, only 2 packages should be enough. The packages are the main package and the model package.

The model package represents the central component of the application. It is responsible for managing the data, logic and rules of the application. It contains two classes: MonitoredData and Activities.

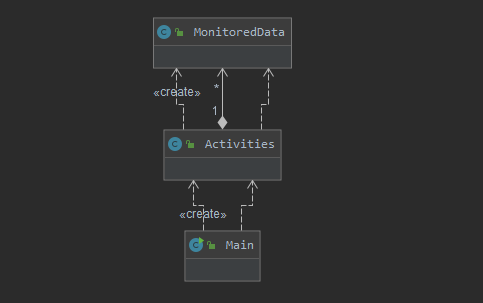
The **MonitoredData** class contains three fields for storing the data read from the input file, as specified in the requirements, constructors, getters, setters and some specific methods.

The **Activities** class contains methods for manipulating the data given in the input file. There are methods for reading the input data, writing to an output file and closing it, and also methods that process the input data to compute some statistics and so on.

The main package contains the Main class, in which an instance of the Activites class is created and specific methods are invoked.

Instead of a graphical user interface, the results are stored in text files, each task having its corresponding output file. For example, the first task will be written in the “TASK\_1.txt” file.

The UML class diagram is the following:



There is a dependency relationship between Main and Activities classes, since in the Main an instance of Activities is created.

Between Activities and MonitoredData exists an aggregation relationship, since Activities contain one or more MonitoredData objects, but MonitoredData can also exist outside the Statistics class.

## 3.2. Algorithms and Data Structures

In the development of the application, various algorithms were used to process the data. These consist of using streams and lambda functions for counting days, activities, calculating sums etc. The algorithms take as input some data structures, which will be presented in the next paragraph.

There are various data structures used in the application, most of them for storing some kind of information about the activities from the input file.

The monitored data is kept in a list (of type ArrayList).

The date and time of the activities are represented by using the Date data structure.

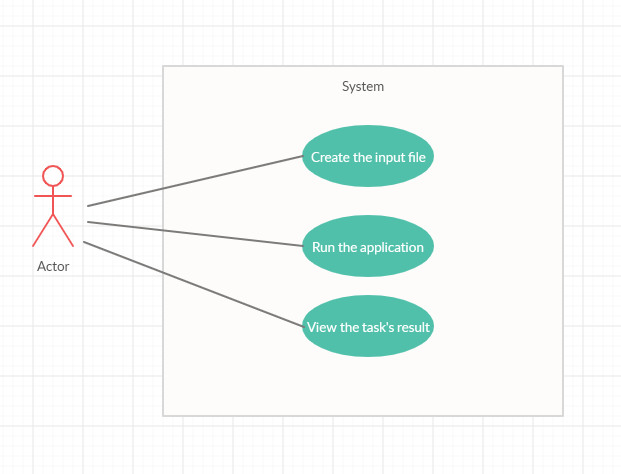
For storing keys and values, as stated in the requirements file, Maps were used.

For counting elements Set was used (of type HashSet), since duplicates should not be allowed.

**3.3. Use Case Diagram**

The Use Case Diagram is simple and intuitive, since there are only 3 tasks a user can do: give the input file, from where the data will be extracted, run the application for doing each task, and finally, visualize the results of the program.

The Use Case diagram is the following:



# 4. Implementation

In this chapter the implementation details of the classes will be presented in detail.

The **MonitoredClass** has three fields for storing the data, which are the following:

**private** Date startTime;

**private** Date endTime;

**private** String activity;

The startTime field represents the starting time of the activity, while endTime the ending time. Both use the Date data structure.

The actitivity field represents the name of the activity, and it is represented as a string.

The class has a constructor with three parameters, as well as getter and setter methods for all the fields.

The class overrides the toString() method for returning all the information about the fields.

There is a method named getDay() which return the current day in the “yyyy – MM - dd” format, since the user is interested only in getting the day, not the specific our.

**public** Date getDay() throws ParseException {

SimpleDateFormat dFormat = new SimpleDateFormat("yyyy-MM-dd");

return dFormat.parse(dFormat.format(startTime));

}

The getDay() method returns the startTime in a Hour format, without taking into consideration the hours, seconds, etc. This is used for comparisons with other days.

**public** Date getDay() throws ParseException {

SimpleDateFormat dFormat = new SimpleDateFormat("yyyy-MM-dd");

return dFormat.parse(dFormat.format(startTime));

}

The **Activities** class has two fields, which are the following:

**private** List<MonitoredData> data;

**private** **final** String filename = “Activities.txt”;

The data field is a list of type MonitoredData, and it is used to save all the read data from the input file.

The filename field sets the name of the file from which the data is to be read. Since the name of the file was given in the requirements file, I chose to set it final in order to give protection against modifications during runtime.

In the constructor method, the ‘data’ field is initialized as an Arraylist;

For **TASK 3,** the **readData()** method is responsible for reading the input data from the given file. The task is done using java streams. The reading is done line by line, and each line is split into three fields, since the fields are separated by two tabs (“ \t\t”). The data is parsed, created with the stream.map() method and finally collected as a list. The format follows the following pattern(“yyyy-MM-dd HH:mm:ss”). Also, for writing the list into the corresponding output file, the String is built with StringBuilder and the wrote to the file.

For **TASK 2**, the **countDays()** method is used. With the help of the getDay() method, the day for each activity is extracted. Since a Set has unique elements, there will be no duplicates, so the number of days will be the size of the set. Th

**public** **void** countDays() {

Set<Date> days = data.stream().map(d -> {

try {

return d.getDay();

} catch (ParseException e) {

e.printStackTrace();

}

return null;

}).collect(Collectors.toSet());

writeFile("TASK\_2.txt", "Number of days is: " + String.valueOf(days.size()));

}

For **TASK 3**, the **countActivities()** method is used. As in the requirements, a Map is used, which maps a String to a Long, an activity to a number. The groupingBy method is used to group the elements of the map, and counting() method is used to count each activity how many times appeared. The output file’s content is built using the StringBuilder type and then written.

**public** **void** countActivities() {

Map<String, Long> activities = data.stream().collect(Collectors.groupingBy(MonitoredData::getActivity, Collectors.counting()));

StringBuilder builder = new StringBuilder();

builder.append("Number of times each activity appeared:\n");

activities.forEach((key, value) -> builder.append(key + ": " + value + " times" + "\n"));

writeFile("TASK\_3.txt", builder.toString());

}

For **TASK 4**, the **countActivitiesPerDay()** was used, which basically works on the same principle as the countActivities() method, since it counts how many times each activity has appeared each day. The Map has the key of Date type, and the mapped value is a Map, consisting of a String, the name of the activity, and a Long, the number of times the activity has appeared. Then, in a forEach loop the output is built using StringBuilder and sent to the corresponding output file.

**public** void countActivitiesPerDay() {

SimpleDateFormat dFormat = new SimpleDateFormat("yyyy-MM-dd");

Map<Date, Map<String, Long>> activities = data.stream().collect(Collectors.groupingBy(

t -> {

try {

return t.getDay();

} catch (ParseException e) {

e.printStackTrace();

}

return null;

},

Collectors.groupingBy(MonitoredData::getActivity, Collectors.counting())));

StringBuilder builder = new StringBuilder();

builder.append("Each day activities\n");

activities.forEach((key, value) -> builder.append(dFormat.format(key) + " activities: " + value + "\n"));

writeFile("TASK\_4.txt", builder.toString());

}

For **TASK 5,** the **countTotalDuration()** method is used, which basically calculates the difference of the endTime and startTime using the getDiff() method. For calculating the total duration of an activity, a Map is used which has a String key, the name of the activity, and the value a Long, the period in minutes for the activity. The summingLong() method is used for computing the total duration.

**public** **void** countTotalDuration() {

Map<String, Long> result = data.stream().collect(Collectors.groupingBy(MonitoredData::getActivity,

Collectors.summingLong(MonitoredData::getDiff)));

StringBuilder builder = new StringBuilder();

builder.append("Duration of each activity\n");

result.forEach((key, value) -> builder.append(key + " " + value + " minutes\n"));

writeFile("TASK\_5.txt", builder.toString());

}

For **TASK 6,** the **frequency()** method was used to filter the activites that have 90% of the monitoring records with duration less than 5 minutes. First, it creates a map with the key of type String and value Long, where all activities which have less than 5 minutes of the duration are filtered. Then, in the activities map is created for calculating how many times each activity appeared during the entire monitoring period. The data is extracted to some lists, for the purpose of calculating probabilities, and if the probability is bigger than 90%, it is added to the StringBuilder. Finally, the activities are written to the “TASK\_6.txt” file.

**public void** frequency() {

Map<String, Long> freq = data.stream().collect(Collectors.groupingBy(MonitoredData::getActivity, Collectors.mapping(MonitoredData::getDiff,

Collectors.summingLong(d->{

if(d<5)

return 1;

else

return 0;}))));

Map<String, Long> activities = data.stream().collect(Collectors.groupingBy(MonitoredData::getActivity, Collectors.counting()));

StringBuilder builder = new StringBuilder();

builder.append("Activities:\n");

List<String> names = new ArrayList<>(freq.keySet());

List<Long> favorable = new ArrayList<> (freq.values());

List<Long> total = new ArrayList<>(activities.values());

for(int i = 0; i < freq.size(); i++) {

if((double) favorable.get(i)/total.get(i) >= 0.9) {

builder.append(names.get(i) + "\n");

}

}

writeFile("TASK\_6.txt", builder.toString());

}

The **writeFile()** method is responsible for writing into files. If the file is not created yet, it creates it. Then, writes the content of the content String into the file and closes the file. This is done in a try catch block .

**private void** writeFile(String fileName, String content){

FileWriter fileWriter = null;

try {

fileWriter = new FileWriter(new File(fileName));

fileWriter.write(content);

fileWriter.close();

} catch (IOException e) {

e.printStackTrace();

}

}

# 5. Usage and Testing

The usage of the application is based on the fact that the input file has a correct format. The format is the same as given in the input file, on each row there is a starting time, ending time and the name of the activity, each one separated with two tabs (\t\t). The application computes the result based on the valid inputs.

Testing was done by partitioning the input file in smaller groups, to be able to follow the output closely. Also, intermediate results were printed to the console to trace the evolution of the program.

# 6. Conclusions

Streams are a good practice for writing shorter and more meaningful code, since the coding style can easily be understood. Also, iterating through collections becomes much easier with streams. There are also some disadvantages of using streams, for example regarding the performance: a for loop through an array is extremely lightweight both in terms of heap and CPU usage. If raw speed and memory thriftiness is a priority, using a stream is worse, but in our assignment it’s clear that streams are a better approach than iterative programming.

After completing this assignment, I can say that I have learned new concepts like Java 8 lambda functions and streams, used them in some simple, intermediate level examples. There exercises helped me to understand the power and effectiveness of using functional programming.

Future developments:

* Implementation of more statistics, keeping track of a history
* Graphical user interface to choose the operations
* Inclusion of more relevant information in the output file
* Input and output file to be assigned dinamically

# 7. Bibliography

<https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

<http://tutorials.jenkov.com/java/lambda-expressions.html>

<https://www.tutorialspoint.com/java8/java8_lambda_expressions.html>

<https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>