Technical University of Cluj-Napoca

Faculty of Automation and Computer Science

2nd Semester 2018-2019

Programming Techniques
Homework 5

Student: Elekes Lukacs-Roland

Group: 30424

Supervising Teacher: Viorica Chifu

Contents

1. Objectives

The main objective is to use **Lambda Expressions and Stream Processing** for the following assignment:

Consider the task of analyzing the behavior of a person recorded by a set of sensors. 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 from the file Activities.txt located in this folder. Write a Java 1.8 program using lambda expressions and stream processing to do the tasks defined below.

2. Problem Analysis, Modelling, Scenarios

2.1. Problem Analysis

Working wit 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

We need to create access to the data that is stored in an input file, for which we have to use reading from a file. The information read should be stored somewhere for further processing. For outputting the results, we need to create a file and have access to it for writing in the file.

A data model is needed to store the read data and we need a class where we can implement the required methods for determining the statistics.

For processing the data streams are mainly used and lambda functions. Data can be easily handled using the Java Streams and combining them with lambda functions makes it even simpler.

2.3. Scenarios

The scenario is reading the data, processing the data and outputting the result into a file.

3. Design

3.1. Class Design, Packages , Relationships and UML Diagrams

In this project the classes is partitioned in two packages. The packages are the main package and model package.

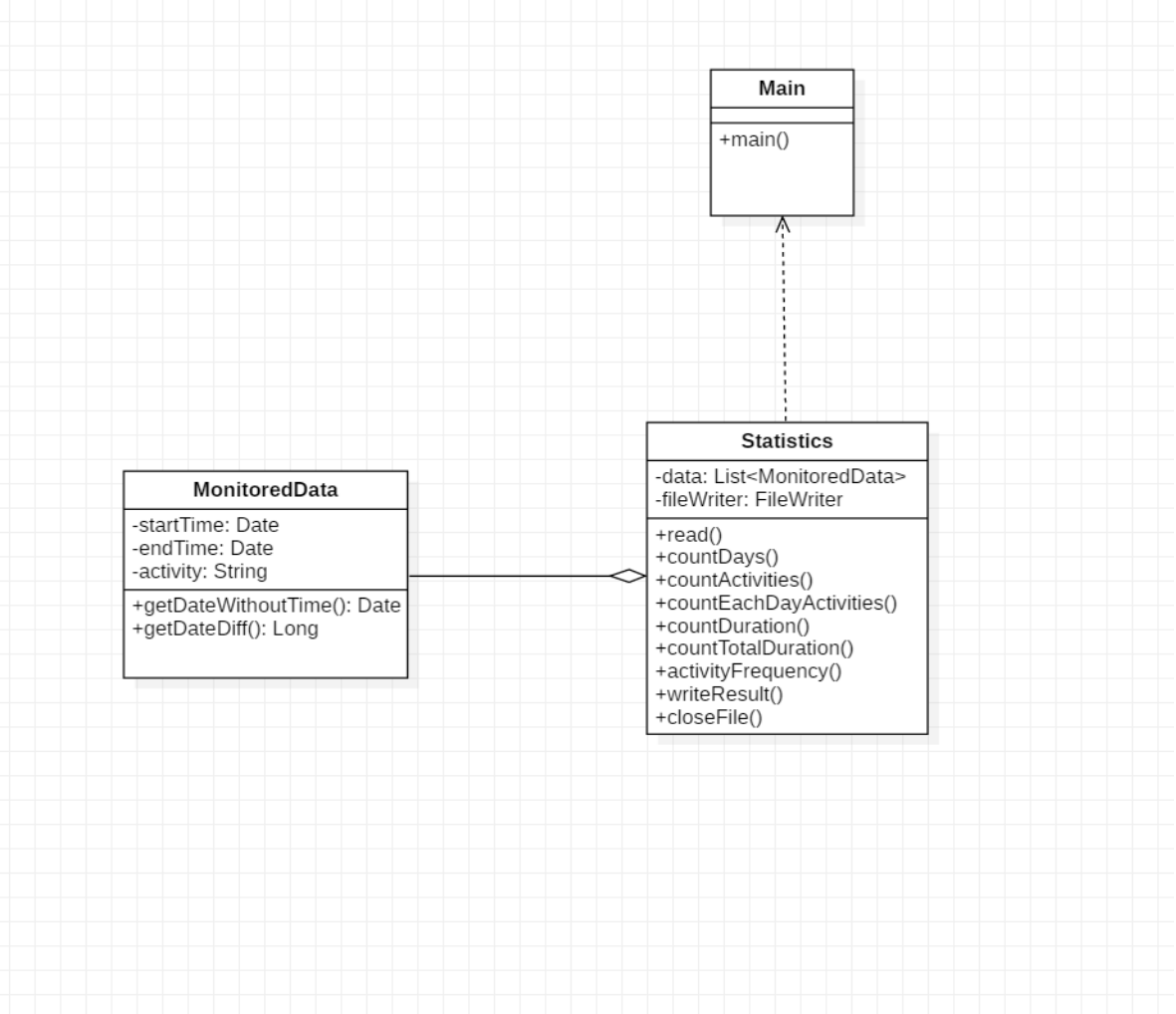
The model package contains two classes: MonitoredData and Statistics. MonitoredData contains three field for storing the data read from the file and includes some getter and setter methods.

The Statistics class contains the utility methods for reading the data, writing and closing the output file, furthermore the methods that process the data can be found in this class.

The main package contains the Main class, in which the methods of Statistics class are invoked.

Instead of a graphical user interface, the results are stored in a text file.

The UML class diagram is the following:



There is an aggregation relationship between the MonitoredData and Statistics classes, because the Statistics contain several MonitoredData, but MonitoredData can exist without Statistics.

Dependency can be observed between Statistics and Main classes, because Main classes uses and instance of Statistics.

3.2. Algorithms and Data Structures

There are used several data structures mostly to store processed data. All he monitored data is stored in a list of type ArrayList. For counting unique elements a Set of type HashSet was used, and for storing some keys and values Maps were used. For representing the date and time, Date data structure was used.

The algorithms consist of using the stream and lambda properties to count, sum etc. the elements that were in the stream, for which the source was always one of the above mentioned data structures.

4. Implementation

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

```
private Date startTime;
private Date endTime;
private String activity;
```

where startTime and endTime are chosen to be represented by the Date data structure and they are the time limits, when the activity starts and ends, and the String activity is for the activity label/name.

The class has a constructor with three parameters for initializing the fields.

Some getter and setter methods are defined for having access to the private fields.

There is a method named getDateWithoutHour() which returns the date from the start time without the hour. In implementation that means that the hour is set to 00:00:00. This form of the date is used for comparison with other dates not taking in consideration the exact time.

```
public Date getDateWithoutTime() throws ParseException {
            SimpleDateFormat dFormat = new SimpleDateFormat("yyyy-MM-dd");
            Date result = dFormat.parse(dFormat.format(startTime));
            return result;
      }
```

A simple date format is defined, and using parse the date is formed according to the simple date format.

Another method is getDateDiff() which returns a long representing the duration between start time and end time in minutes. The two dates are converted to milliseconds, they are subtracted and converted back to minutes.

```
public long getDateDiff() {
          long diffInMillies = endTime.getTime() - startTime.getTime();
          return TimeUnit.MINUTES.convert(diffInMillies,TimeUnit.MILLISECONDS);
      }
```

HashCode and equals methods are implemented also using Eclipse's predefined methods.

The class Statistics has two fields:

```
private List<MonitoredData> data;
private FileWriter fileWriter;
```

where data is a list of monitored data for storing the information read from the file and fileWriter is for opening or creating a file and writing in this file.

In the constructor the list is initialized as an ArrayList, the output file is created and the fileWriter is initialized, the constructor has no parameters.

There is a method read() for reading the data from the file. It is done with Streams, every line is processed from the "Activities.txt" text file, the lines are split into three at every two tabs ("\t\t"), this way we have a string for each field from monitored data. The dates are parsed, and the data is created using the .map method of streams and collected as a list at the end.

The countDays() methods counts the number of days that are monitored. From the start times the date without hour (hour set to 0) are extracted and stored in a set, and a set has unique elements, duplicates does not appear in the set. The number of days is given by the size of the set.

```
public void countDays() {
		Set<Date> days = data.stream().map(d -> {
			try {
				return d.getDateWithoutTime();
			} catch (ParseException e) {
				e.printStackTrace();
			}
			return null;
		}).collect(Collectors.toSet());
		writeResult("Number of days is: " + days.size());
		//System.out.println("Number of days is: " + days.size());
	}
```

The countActivities() method counts how many times has appeared each activity over the entire monitoring period. Here a map is used, that has as a key the name of the activity and as value a Long representing the number of appearances. The method groupingBy from Collectors is used for creating the map, map is grouped by activity names, which will be the key, and for the value the counting() method is used.

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

writeResult("\r\nTimes each activity appeared:");
activities.entrySet().stream().forEach(d -> {writeResult(d.getKey() + "	" +
d.getValue());
		//System.out.println(d.getKey() + "	" + d.getValue());	});
	}
```

The countEachDayActivities() method counts how many times has appeared each activity for each day over the monitoring period working on the same principle as the countActivites() method.

The Map defined here is of type Map<Date, Map<String, Long>>, so for every different day (key) a map with activities and their number of appearances is mapped.

In the countDuration() method every line is printed adding their duration calculated by using the getDateDiff() method from MoitoredData class.

The countTotalDuration() method computes for each activity the entire duration over the monitoring period. A map is created, where the key is the name of the activity and the value is a Long representing the total duration in minutes. It is calculated using the summingLong method from Collectors that sums the getDateDiff() for every line where the key is the same.

```
public void countTotalDuration() {
Map<String, Long> result =
data.stream().collect(Collectors.groupingBy(MonitoredData::getActivity,
Collectors.summingLong(MonitoredData::getDateDiff)));

writeResult("\r\nTotal duration for every activity:");
result.entrySet().stream().forEach(d -> writeResult(d.getKey() + "       " +
d.getValue() + " minutes"));
       }
```

The activityFrequency() method filters the activities that have 90% of the monitoring records with duration less than 5 minutes. Creates a map with the name of the activity as a string and as a value the number each activity was done in less than 5 minutes (if the duration is less than 5 minutes adds 1, else adds 0). Another map is created for counting each activity how many times appeared during the monitored period, after that the data are extracted to lists, and we iterate through the lists in the same time to get the probabilities, and if the probability is above 0.9 (90 %) than the name of the activity is printed.

5. Usage and Testing

For the correct usage of the application, a data set is needed stored in a file. After the program run, the results appear in an output text file.

For testing intermediary result were displayed on the console.

6. Conclusions

Using streams allow us to write shorter code, and to solve some problems in an easier way with the stream and Collectors function. Iterating is much simpler then before Java 8, construction of

collection like List, Set or Map is easier with Streams. Combining Streams with lambda functions gives us a better performance and results.

This assignment helped me to be more familiar with the Java 8 concepts, but I still need exercise to understand better its power and to be able to use it in every situations that becomes simpler to solve with streams or lambda functions.

Future developments:

- Some generalizations regarding the filenames for input or output
- More elegant way to use streams
- Nicer printing of a data in the file

7. Bibliography

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

https://www.mkyong.com/java8/java-8-flatmap-example/

http://users.utcluj.ro/~cviorica/PT2019/

https://www.mkyong.com/java8/java-8-streams-filter-examples/

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

https://stackoverflow.com/questions/40826431/whats-the-difference-between-groupingby-and-mapping-in-collectors-java

https://www.baeldung.com/java-8-collectors

https://stackoverflow.com/questions/30515792/collect-stream-with-grouping-counting-and-filtering-operations

https://www.geeksforgeeks.org/stream-map-java-examples/

https://docs.oracle.com/javase/8/docs/api/index.html?java/io/FileWriter.html

https://www.baeldung.com/java-date-without-time