

FACULTY OF AUTOMATION AND COMPUTER SCIENCE

FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 5: PROCESSING DATA OF DAILY LIVING ACTIVITIES

Coordinator: Dr. Eng. Cristina Bianca Pop

Student: Bianca-Veronica Avram

2nd year, 2nd semester

2020

Contents

1. *Objectives*
2. *Problem analysis, modelling, scenarios, use case*
3. *Design*
4. *Implementation*
5. *Results*
6. *Conclusions*
7. *Bibliography*

*1.Objectives*

This assignment has as its main objective developing an application for processing data obtained from a smart house monitorization sensor system. Data collected by sensors is collected into a text file, then statistics are built with that data also in text files. The application is meant to keep track of daily activities of a regular person inside their house, so questions like “How many times an activity occurred each day?” or “How many different days were under test in the monitorization process?” were asked so that this application that was developed to answer them.

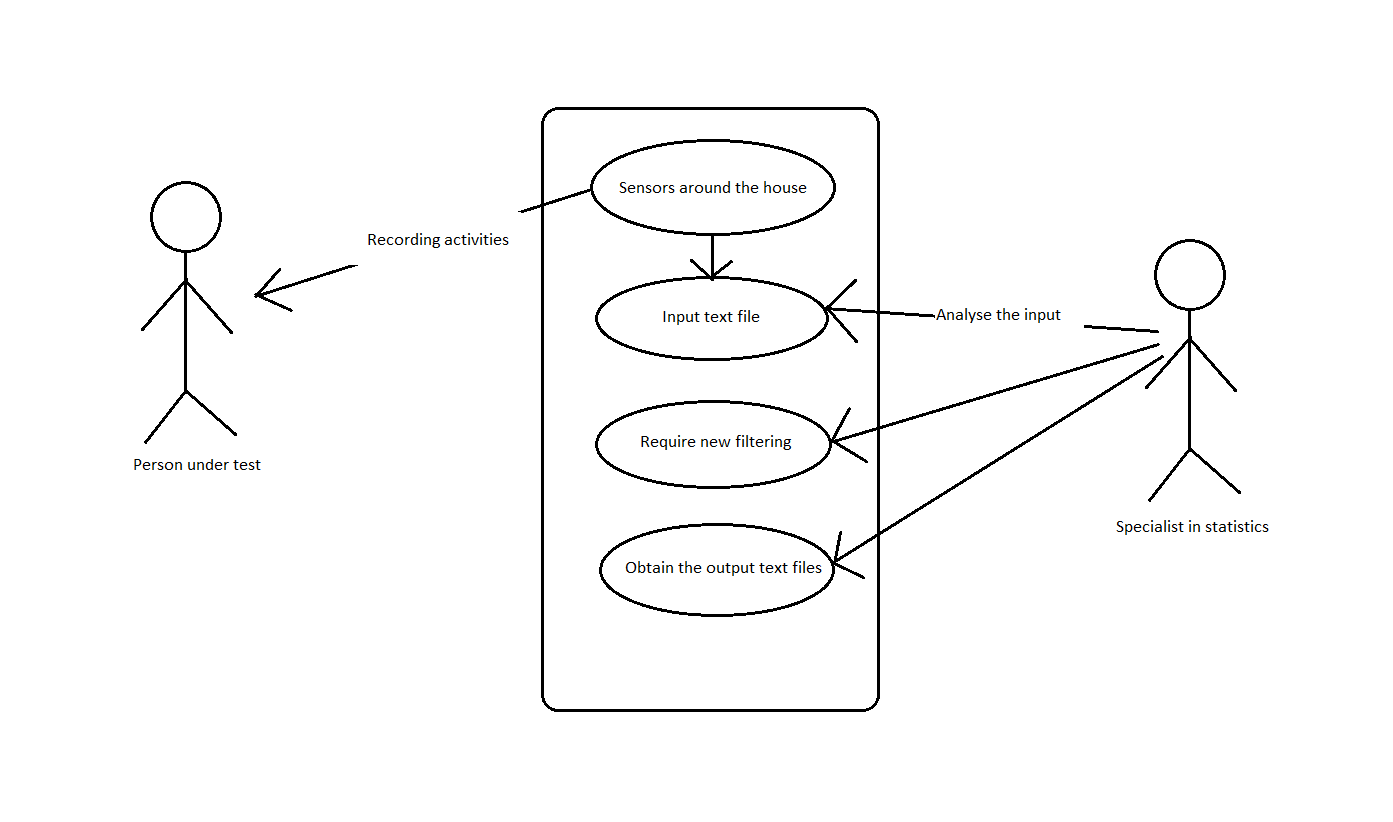
The data provided in the input file, that is based on some sensors recordings contain similar lines, each with three fields: start time of the activity, end time of the activity and the name of the activity the person is performing.

From the assignment text the secondary objectives could be stated:

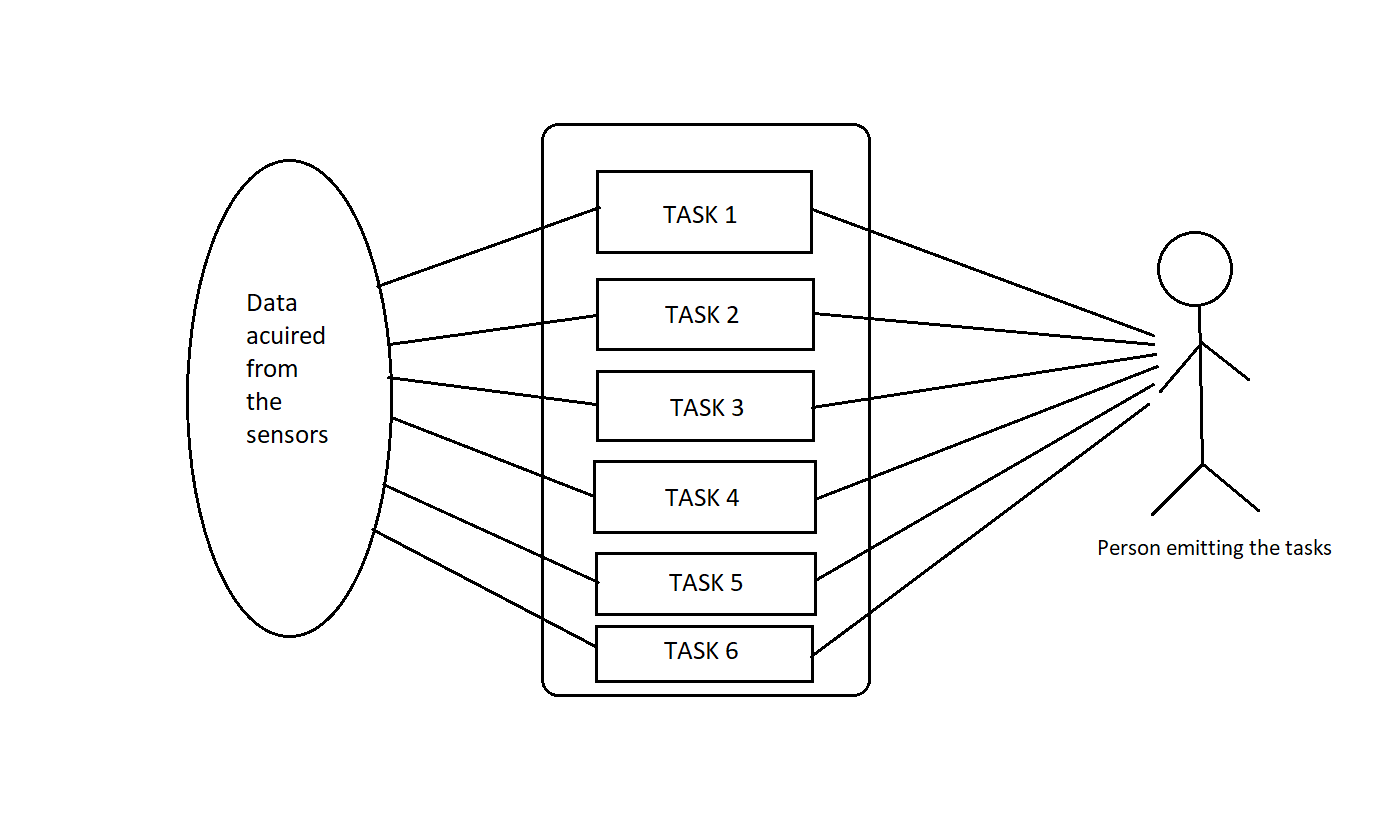
1. Reading data from file
   * This one was the most basic one and it only implies reading the input file line by line
   * It will be further discussed in the implementation section, section 4
2. Working with java streams
   * This was the newest aspect that covered this assignment
   * A set of stream methods was used and they will be presented in section 4
   * Mainly, all the tasks that follow use streams in getting a result, so this particular part will be included in describing the implementation of each of the tasks
3. TASK 1
   * This task resumed to reading the data from file, then converting it into a user defined type, class MonitoredData
   * The features of this class will be presented int section 3, but all the parts that compose it will be split and distributed to tasks
   * The task itself will be described in section 4
4. TASK 2
   * This task is meant to use method count() on streams together with some filtering criteria
   * Details about this conception can be found in section 4
5. TASK 3
   * This task required to count all the different occurrences of each activity during the monitoring process.
   * This will be presented in section 4
6. TASK 4
   * This task required to count all the different occurrences of each activity each day of the monitoring process.
   * This will be presented in section 4
7. TASK 5
   * This task asked for the total duration of each activity during the monitoring period
   * This will include implementing a new class which will be introduced in section 3
   * This task will be described in section 4
8. TASK 6
   * This task introduced a new filtering criteria for some activities that were searched
   * All the activities that appear with more than 90% of the records as lasting less than 5 minutes are wanted

*2.Problem analysis, modelling, scenarios, use case*

This application is not meant to directly interact with a regular user, as they can only observe the statistics that were obtained after monitorization. A person is placed under test inside a house full of sensors meant to register all the activities they do and the time period they do them. The sensors acquire data which is afterwards organized into a text file which can be noted by a specialist.



The specialist is the person who emits the tasks. It can be seen in such a way, even though the application itself, as it is at this moment, will not take other tasks. But at the utility level, there should be a person who wants to get some information that can be user in future research, so they simply ask for it by means of a tasks.

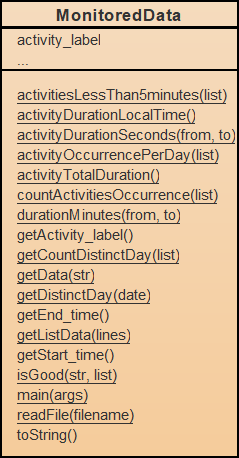


All the tasks use only the data from the sensors that are spread around the house where the tested person lives. The activities that are in the monitoring process are the following: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare\_Time/TV, Grooming. The time period for the monitoring is chosen by the specialist.

As the output is generated, the specialist can note it from the text files that were created as answers to the requirements.

*3.Design*

This assignment explicitly required the existence of one single class and that is MonitoredData.

**

This class contains three fields, as required in the assignment text, and these are: start\_time (LocalDate Time), end\_time (LocalDateTime) and activity\_label (String) and all the methods that were implemented in order to obtain the solution for the problem. There is only one constructor, the one with parameters and the getters for each field. Setters were not necessary, as there are no such sort of accesses to the fields of a MonitoredData object.

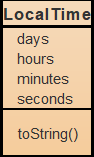
Leaving aside the methods that were implemented to solve the tasks, a toString() override was created in such a way that the aspect of the string representation of a MonitoredData object is just like its representation from the input file.

@Override  
public String toString(){  
 String str = start\_time.toString().replace('T',' ');  
 if(start\_time.toString().length() < 19)  
 str += ":00";  
 str += " "+end\_time.toString().replace('T',' ');  
 if(end\_time.toString().length() < 19)  
 str += ":00";  
 str += " "+activity\_label;  
 return str;  
}

The author had to deal with the problem that in the LocalDateTime string representation there is a ‘T’ separator between date and time, so that is the motivation for using that replace method call. Then, another problem was when attempting to visualize the result in the Task\_1.txt file, the dates were not completely aligned and that was because if the time had “:00” in the end, it was omitted, so it was put there manually, in order to preserve the aspect.

The assignment required for task number 5 to obtain the whole duration of each activity during the monitoring period. The first instinct was to use the LocalTime class from java.time, but a problem was encountered. There was an activity, Sleeping, which, during the whole monitoring period, exceeded 24 hours, so an object of type LocalTime was not enough to hold this information about an activity.

The solution came from some of the colleagues talking on the Discord server, so the decision resumed to creating a new class, with the same name, LocalTime, in order to still keep a consistency and somehow a reference to what this class actually means. This class was then designed to act as desired. It actually has only 4 fields: hours, days, minutes, seconds. These can make the format of a period of time. In this class there is only the constructor with parameters and the overridden method toString().



*4.Implementation*

The implementation of the solution for the given problem was performed gradually, task by task. First of all, it was the documenting part, reading lecture slides and some information from internet, as well as watching some tutorials about working with streams.

For each task it was made a method that gives the wanted result. In many of them, more methods were added in order to avoid putting too much work on a single method.

***TASK1***

Reading the input text file was done using a Stream<String> structure, so it was also done using streams. The result of this method was a list containing all the lines of the input file as strings.

Then, the other part of task 1 was transforming these lines that were obtained from the file into objects of type MonitoredData. This was easily done just by splitting the line using split() method and a regex that considers 2 to 10 spaces as a separator. This was decided just by analyzing the input file. The spacing between elements composing the start time and end time and the spacing between the future fields must be differentiated in a way or another.

Again streams were used and only one line of code was enough to get the desired result.

list = lines.stream().map(MonitoredData::*getData*).collect(Collectors.*toList*());

The map method call was done using method reference to getData. This method is the one that splits the line into its component and creates an object of type MonitoredData.

***TASK 2***

Task 2 asked to get the number of different days in the monitoring period. This was done also using streams, so two streams were created, one for the start time and one for the end time, taking in account only the distinct days. This seemed necessary as there might be activities that start at before midnight and end after midnight, so that would have caused an error.

Distinguishing between 2 days is done by a certain key that is calculated with the formula :

return dayOfMonth + month \* 31;

The two streams are merger using concat() method

List<Integer> merged = Stream.*concat*(endDays.stream(),startDays.stream()).collect(Collectors.*toList*());

Then the count() method is called to obtain the result.

***TASK 3***

Task 3 is implemented by only one method which first obtains the labels of all activities, then iterates through it and creates a map from that string to the number of different occurrences of that activity in the monitoring time.

List<String> activities = list.stream().map((MonitoredData p)->p.getActivity\_label()).collect(Collectors.*toList*());  
for (String str:activities) {  
 long occ = list.stream().filter(p->p.getActivity\_label().equals(str)).count();  
 map.put(str, (int) occ);  
}

***TASK 4***

Task 4 requires the number of occurrences of each activity each day. This was also done using one single method, but here it is more complicated. First all the distinct days were obtained, as a list of integers representing the code that was presented above. Then, this list was iterated and, for each day the list of activities was iterated to count the number of occurrences per day. This was done using a multiple statement filter.

long count = list.stream()  
 .filter(p->p.getActivity\_label().equals(str) && *getDistinctDay*(p.getStart\_time())==i)  
 .count();

***TASK 5***

Task 5 asked for the total duration of each activity. Here the class LocalTime was involved, but it was discussed in the previous section.

long durations = list.stream()  
 .filter(p->p.getActivity\_label().equals(str))  
 .map(p->*activityDurationSeconds*(p.getStart\_time(),p.getEnd\_time()))  
 .reduce((long) 0,(e1, e2)->e1+e2);  
LocalTime duration = *activityDurationLocalTime*(durations);

Again the activity list is iterated and then, using stream processing, the duration is obtained using a filter and also a map to obtain the duration for one occurrence of the activity at a time. This is done using the method activityDurationSeconds() which computes the number of seconds between start\_time and end\_time. The method reduce() adds all the durations, then the LocalTime object is obtained by calling activityDurationLocalTime() method, which converts the seconds into an object of type LocalTime.

***TASK 6***

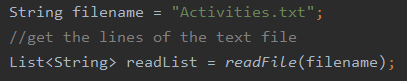
Task 6 wants to filter the activities that have more than 90% of the records lasting less than 5 minutes. This is done in two theoretical steps. Firstly, the distinct activities are obtained and secondly, each of these activities is filtered through the method isGood(). This method first obtains all the durations in minutes of all the occurrences of the activity received as parameter, than, in another stream, it filters only those that last less than 5 minutes. Like this, all the needed data for computing the percent is obtained and it can be returned.

List<Integer> durations = list.stream()  
 .filter(p->p.getActivity\_label().equals(str))  
 .map(p->*durationMinutes*(p.getStart\_time(),p.getEnd\_time()))  
 .collect(Collectors.*toList*()); //this list contains all the durations in minutes for activity str  
int count = durations.size();  
long lessThan5 = durations.stream().filter(p->p<=5).count();  
float percent = ((((float) lessThan5)/count)\*100);

*5.Results*

No testing framework was used in this assignment. The results of the application consist of a set of text files that contain the information required by each task, represented under String format. The application was run multiple times and, as expected, the results remained consistent, even with the jar file that was created.

Moving to the jar file, the assignment text only said that the application should be run by running the command java -jar PT2020\_30421\_Avram\_Bianca\_Assignment\_5.jar. The problem here come in what concerns including the input file with the data, so the author decided to place it inside the code, like reading implicitly from a text file called “Activities.txt”



So, when it is run, the user needs to make sure that this text file they want as input is called “Activities.txt” and is placed in the same directory as the jar file.

The format of the input text file is the one specified in the lab text and the format of the output is slightly different for each task. Note that there is a separate “Task\_X.txt” for each of the 6 tasks. Each of these files start with the number of the task and a very short description of the requirement for that specific task, then, the information that needs to be printed is separated by new lines.

*6.Conclusions*

The most obvious conclusion that can be drawn from this assignment is that working with streams in a declarative manner is way more efficient regarding the time spent coding than using declarative programming. Java collections and java streams work hand in hand for making the programmer’s life easier when writing code, but maybe a little bit more difficult when thinking about the code they need to write. The practice is everything, as in any situation, so, by practice, the thinking might become a releasing task too.

Streams were introduced to be able to write little, but to do a lot. The machine actually knows what it has to do with the data provided, so the programmer only has to give the correct set of instructions for that.

Then, working with lambda expressions did not mean such an improvement, but many of the stream methods required a lambda expression as parameter (predicate). In all the cases, this expression was a very simple one, like a simple return or a sum.

The author was surprised that most of the methods that were implemented using streams succeeded from the very beginning and did not need a debugging procedure, in contrast with all the other assignments.

*7.Bibliography*

1. <https://mkyong.com/java8/java-8-stream-read-a-file-line-by-line/>
2. <https://www.baeldung.com/java-stream-filter-lambda>
3. <https://www.baeldung.com/java-streams-distinct-by>
4. <https://www.baeldung.com/java-combine-multiple-collections>
5. <https://howtodoinjava.com/java8/java-stream-distinct-examples/>
6. <https://www.geeksforgeeks.org/stream-reduce-java-examples/>
7. <https://stackoverflow.com/questions/25747499/java-8-difference-between-two-localdatetime-in-multiple-units>
8. <https://www.baeldung.com/java-date-difference>
9. <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>
10. <https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>
11. <https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html>
12. <https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>