MINISTRY OF EDUCATION AND RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 5
SOLUTION DESCRIPTION DOCUMENT

PROCESSING SENSOR DATA OF DAILY LIVING ACTIVITIES

Student: Mihai Cristina-Mădălina
Teaching assistant: Chifu Viorica

Faculty of Automation and Computer Science
Computer Science Department (En)
2nd year of study, gr. 30424

Content:

1. Objective

- **The main objective** of this laboratory work is to **design, implement and test an application for analyzing the behaviour of a person** recorded by a set of sensors installed in its house. The historical log the person's activity is stored in tuples: start_time, end_time and activity_label.
- The project will implemented as a **Maven Project** using **Intellij IDEA application**.

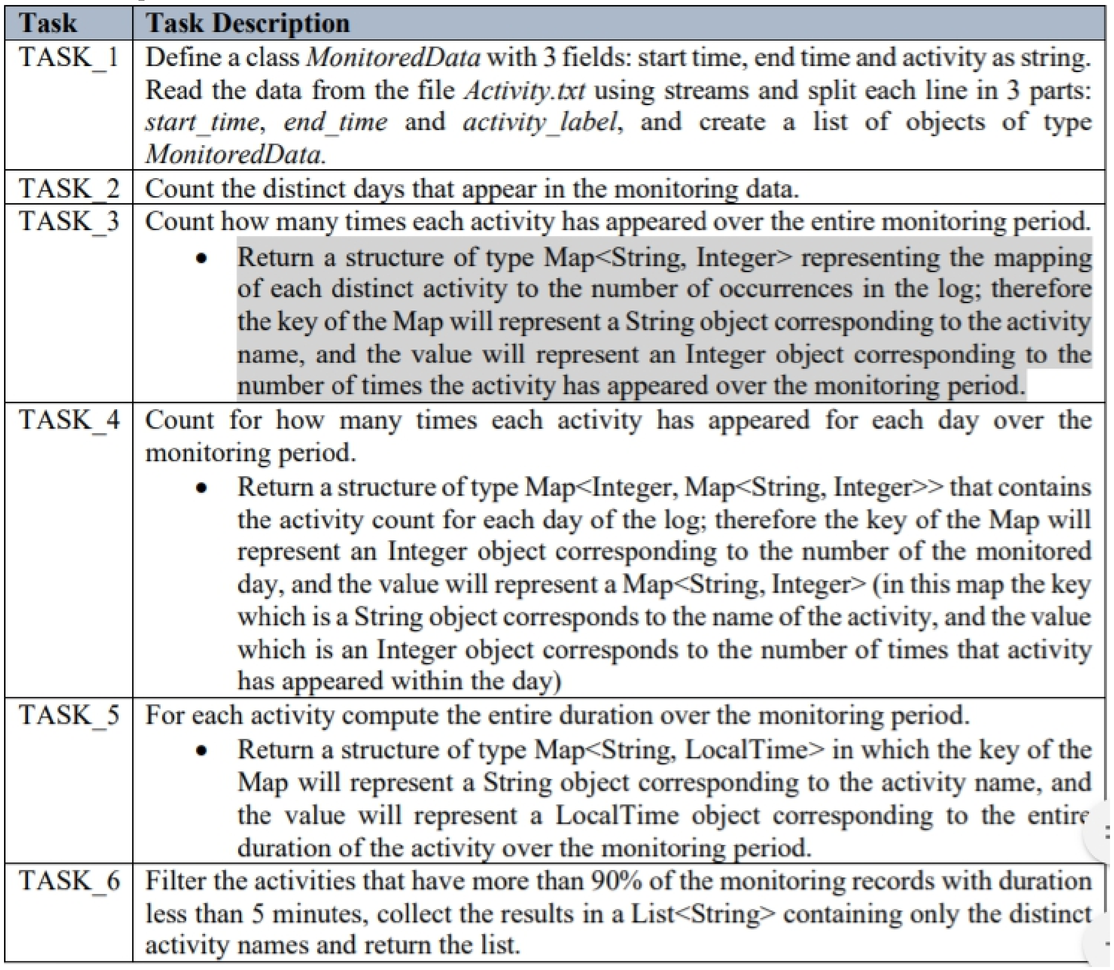
Secondary objective:

- to use functional programming in Java with lambda expressions and stream processing;
- to create a jar file for executing the application;
- to generate a different txt file for each task required.

2. Analysis of the laboratory work, scenarios, use-cases

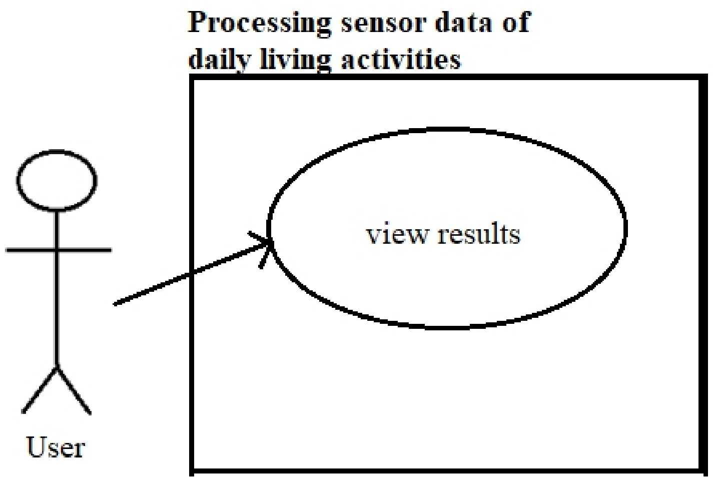
Analysis:

The historical log of the person's activity is stored as tuples, 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 http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_5/
The tasks required as detailed in the table below:



Task	Task Description
TASK_1	Define a class *MonitoredData* with 3 fields: start time, end time and activity as string. Read the data from the file *Activity.txt* using streams and split each line in 3 parts: *start_time*, *end_time* and *activity_label*, and create a list of objects of type *MonitoredData*.
TASK_2	Count the distinct days that appear in the monitoring data.
TASK_3	Count how many times each activity has appeared over the entire monitoring period. • Return a structure of type Map<String, Integer> representing the mapping of each distinct activity to the number of occurrences in the log; therefore the key of the Map will represent a String object corresponding to the activity name, and the value will represent an Integer object corresponding to the number of times the activity has appeared over the monitoring period.
TASK_4	Count for how many times each activity has appeared for each day over the monitoring period. • Return a structure of type Map<Integer, Map<String, Integer>> that contains the activity count for each day of the log; therefore the key of the Map will represent an Integer object corresponding to the number of the monitored day, and the value will represent a Map<String, Integer> (in this map the key which is a String object corresponds to the name of the activity, and the value which is an Integer object corresponds to the number of times that activity has appeared within the day)
TASK_5	For each activity compute the entire duration over the monitoring period. • Return a structure of type Map<String, LocalTime> in which the key of the Map will represent a String object corresponding to the activity name, and the value will represent a LocalTime object corresponding to the entire duration of the activity over the monitoring period.
TASK_6	Filter the activities that have more than 90% of the monitoring records with duration less than 5 minutes, collect the results in a List<String> containing only the distinct activity names and return the list.

Use-cases:



- Title: view results
  Resume: The user can view the filtered and analyzed data by opening the txt files after running the application.
  Actor: User

Scenarios:

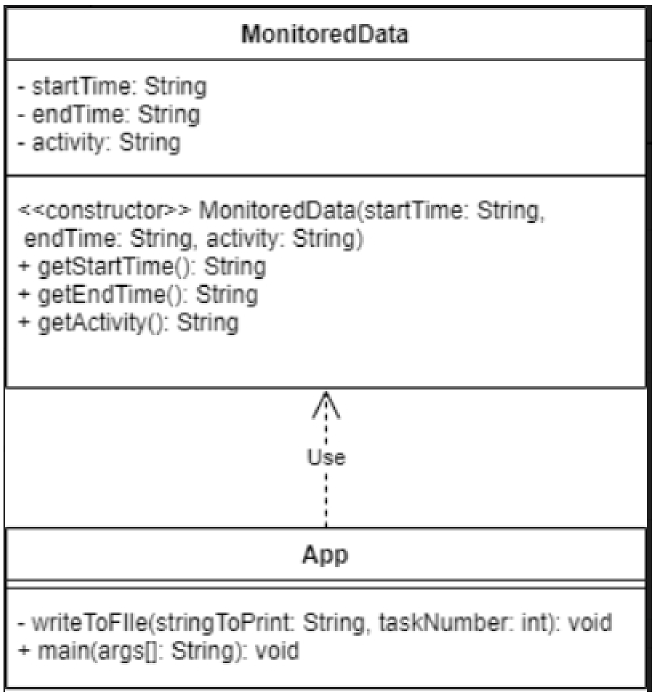
- Preconditions: The user has to introduce the following command line in the terminal: java -jar PT2020_30424_Cristina_Mihai_Assignment_5.jar
- Normal Scenario: The user has perfectly run the command. After this, he/she can open the generated txt files named *Task_x*, where *x* is the number of task and can vary between 1 and 6.
- Alternative Scenario: If the user does not introduce the command line as predefined, an error will occur.

3. Design

3.1 Design approach

The processing data system has only one package due to the simplicity of the requirements, namely *main_package.* It contains two classes which represent the model of the application and the processing part.

3.1 Class diagram


MonitoredData
- startTime: String
- endTime: String
- activity: String
<<constructor>> MonitoredData(startTime: String, endTime: String, activity: String)
+ getStartTime(): String
+ getEndTime(): String
+ getActivity(): String
Use
App
- writeToFIle(stringToPrint: String, taskNumber: int): void
+ main(args[]: String): void

3.1 Data structures

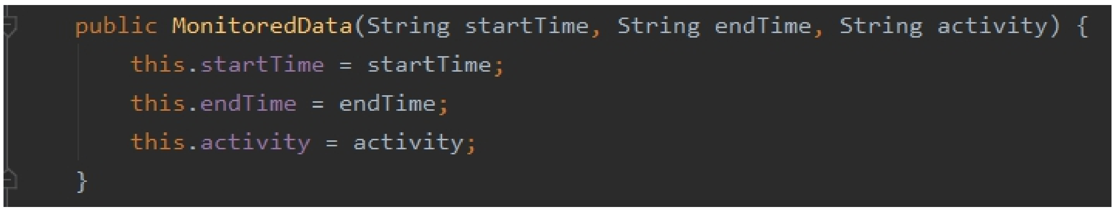
- List<E> : I used this collection during implementation because it is an ordered one and I could easily keep track of the activities given. “E” was replaced either by *MonitoredData* to have access to any activity, either by *String* to store some particular elements of the activities such as the distinct days.
- Map<K,V> : The advantage of this data structure is the uniqueness of the value provided by the key. In our case, the keys were always the name of the activity, whereas the value was changing from task to task. It was either the number of occurrences of the activities during the monitoring period or the duration of each one.
- Map<K, Map<K’,V’>> : This data structure was used to provide a safe and high-quality storage of the frequency of the activity during each day. So, I chose the key of the outer map to be the day of the month, the key of the inner map, namely K’, to be the activity label and the value of the inner map, namely V’, to be the number of occurrences of each activity in the corresponding day.

4. Implementation

MonitoreData class:

Instance variables:
- startTime: String
- endTime: String
- activity: String

All the variables are private and have the following meaning: *startTime* and *endTime* represent the date and time when each activity has started and ended while the *activity* represents the type of activity performed by the person: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare_Time/TV, Grooming. There has been implemented a constructor where all three variables are instantiated.

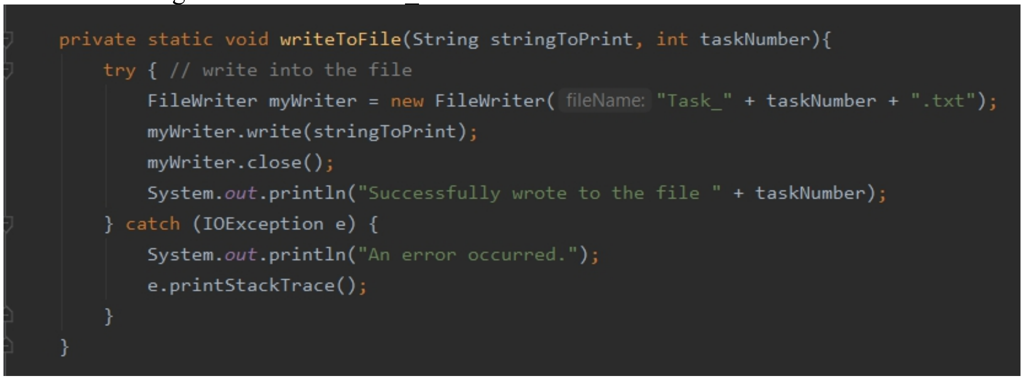


```java
public MonitoredData(String startTime, String endTime, String activity) {
    this.startTime = startTime;
    this.endTime = endTime;
    this.activity = activity;
}
```

Also, the standard getters have been implemented since we want to have access to these variables. The setters we not implemented because the *MonitoredData* elements are read from the text file given, they are supposed to be correct and we are not advised to make any changes.

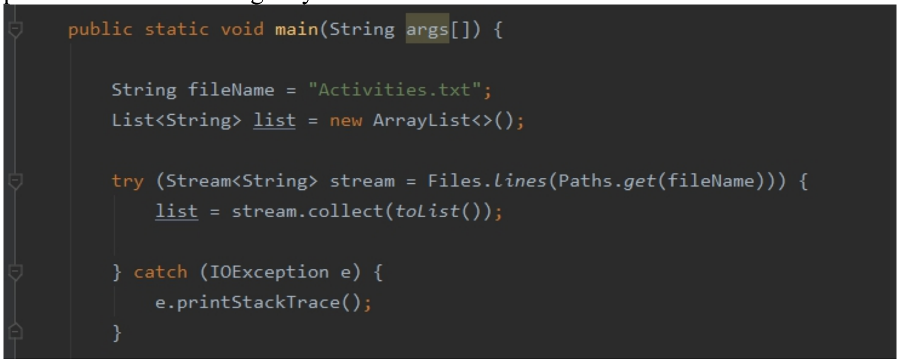
App class:

There was no need for variables since we perform all the processing in the main class.
The auxiliary method *writeToFile(stringToPrint: String, taskNumber: int): void* (available below) is helpful for creating the necessary .txt files and writing to them. It is given as parameters a string which is desired to be written in the file created and an integer which will be part of the file name, since the files have to be given the name "Task_number.txt".



```java
private static void writeToFile(String stringToPrint, int taskNumber){
    try { // write into the file
        FileWriter myWriter = new FileWriter( fileName: "Task_" + taskNumber + ".txt");
        myWriter.write(stringToPrint);
        myWriter.close();
        System.out.println("Successfully wrote to the file " + taskNumber);
    } catch (IOException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
```

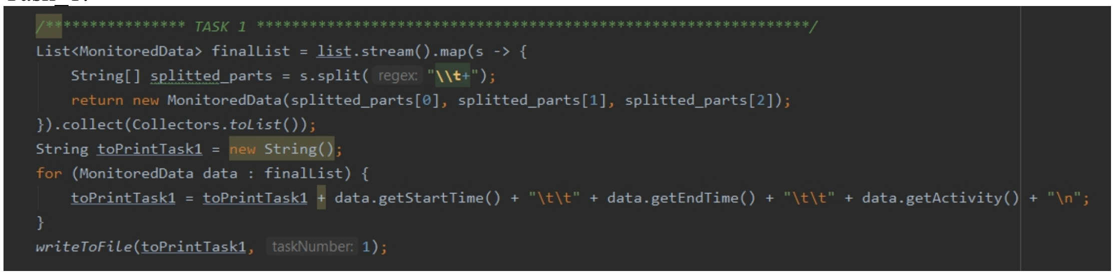
In the main static method, the data from the supplied .txt file is read and the requirements are performed the following way:



```java
public static void main(String args[]) {

    String fileName = "Activities.txt";
    List<String> list = new ArrayList<>();

    try (Stream<String> stream = Files.lines(Paths.get(fileName))) {
        list = stream.collect(toList());

    } catch (IOException e) {
        e.printStackTrace();
    }
```

First, we read the data using stream and store each line from the file in a list of Strings.
Next, we perform the tasks one by one:

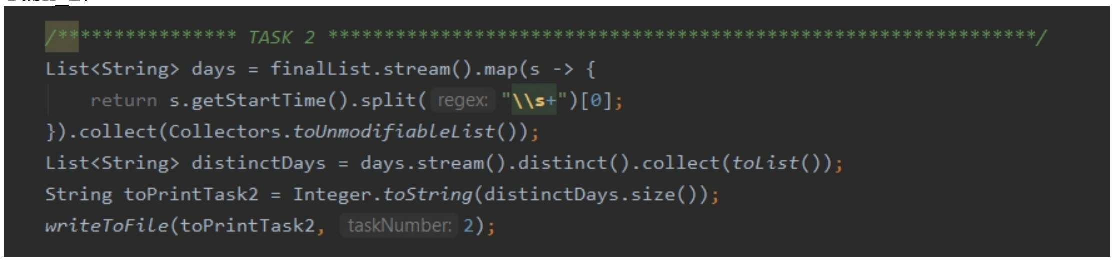
Task_1:



```
/*************** TASK 1 *****************************************************************/
List<MonitoredData> finalList = list.stream().map(s -> {
    String[] splitted_parts = s.split( regex: "\\t+");
    return new MonitoredData(splitted_parts[0], splitted_parts[1], splitted_parts[2]);
}).collect(Collectors.toList());
String toPrintTask1 = new String();
for (MonitoredData data : finalList) {
    toPrintTask1 = toPrintTask1 + data.getStartTime() + "\t\t" + data.getEndTime() + "\t\t" + data.getActivity() + "\n";
}
writeToFile(toPrintTask1, taskNumber: 1);
```

- declare a list of monitoredData elements;
- map each element of the previous list;
- split each string of the previous list and create an monitoredData element out of them;
- collect the final result;
- create a string with the desired output and call the method *writeToFile* to create and fill in the *Task_1.txt*

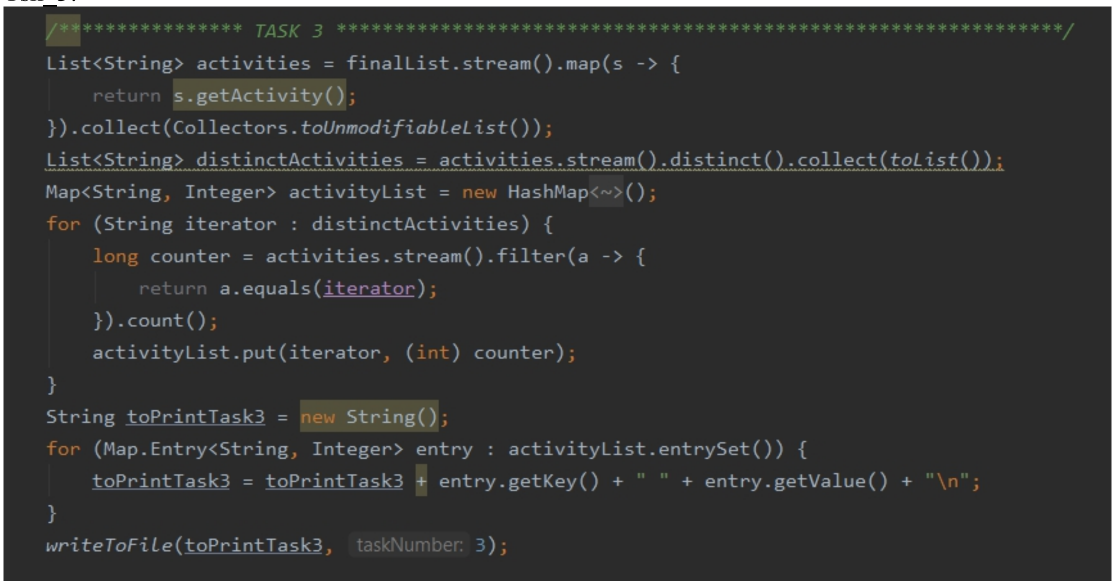
Task_2:



```
/*************** TASK 2 *****************************************************************/
List<String> days = finalList.stream().map(s -> {
    return s.getStartTime().split( regex: "\\s+")[0];
}).collect(Collectors.toUnmodifiableList());
List<String> distinctDays = days.stream().distinct().collect(toList());
String toPrintTask2 = Integer.toString(distinctDays.size());
writeToFile(toPrintTask2, taskNumber: 2);
```

- map just through each startTime in the list of monitoredData (the endTime will have the same number of distinct days, so I chose to iterate just one of them);
- split it in order to get the day of form yyyy-mm-dd;
- collect the data resulted in a list of strings;
- send to *writeToFile* method the size of the created list converted to string.
The list of distinct days will be used later as well.

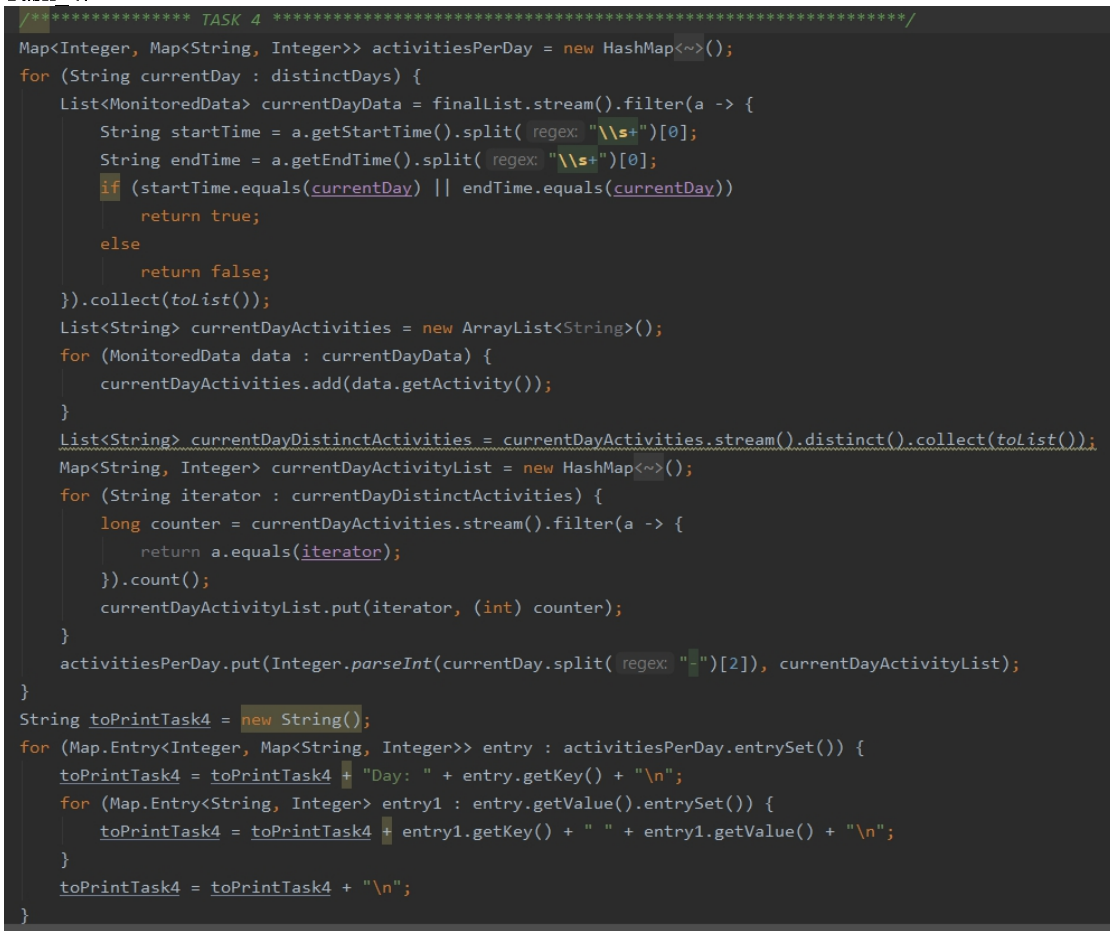
Tsk_3:



```
/*************** TASK 3 *****************************************************************/
List<String> activities = finalList.stream().map(s -> {
    return s.getActivity();
}).collect(Collectors.toUnmodifiableList());
List<String> distinctActivities = activities.stream().distinct().collect(toList());
Map<String, Integer> activityList = new HashMap<~>();
for (String iterator : distinctActivities) {
    long counter = activities.stream().filter(a -> {
        return a.equals(iterator);
    }).count();
    activityList.put(iterator, (int) counter);
}
String toPrintTask3 = new String();
for (Map.Entry<String, Integer> entry : activityList.entrySet()) {
    toPrintTask3 = toPrintTask3 + entry.getKey() + " " + entry.getValue() + "\n";
}
writeToFile(toPrintTask3, taskNumber: 3);
```

- apply the same logic to create a list of string with all the activities encountered in the file;
- apply *distinct* on all the activities created at the previous step and create a new list of distinct activities;
- create a map with the activity name as a key and the number of occurrences of each activity on the entire monitoring period as a value.;
- iterate trough the list of distinct activities in the file, filter them using streams and count the occurrence of each one;
- add the result in the map;
- build a suitable string and send it to the *writeToFile* method.

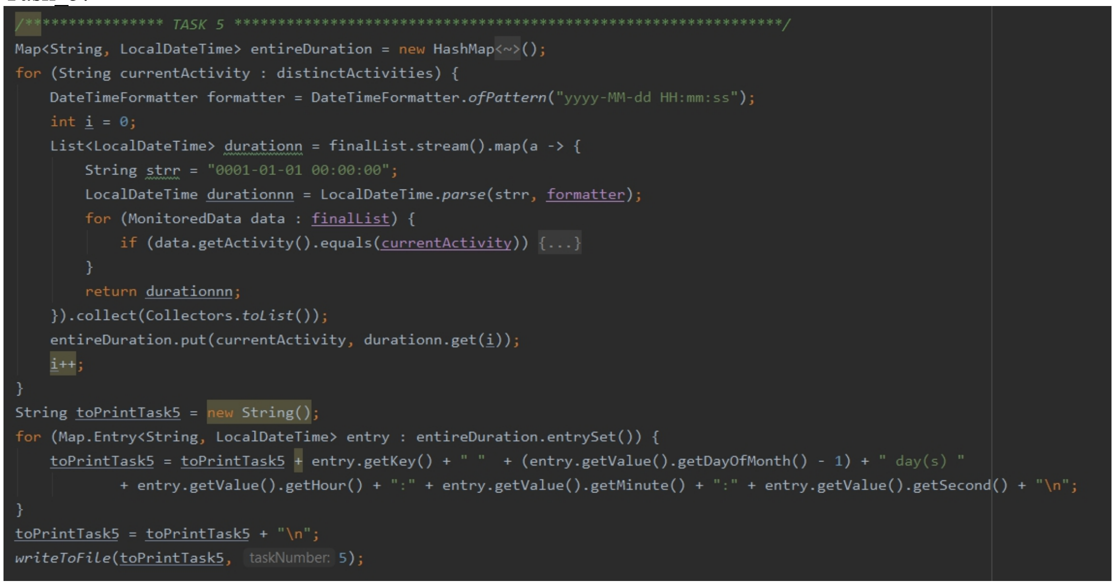
Task_4:



```
/**************** TASK 4 *******************************************************************/
Map<Integer, Map<String, Integer>> activitiesPerDay = new HashMap<~>();
for (String currentDay : distinctDays) {
    List<MonitoredData> currentDayData = finalList.stream().filter(a -> {
        String startTime = a.getStartTime().split( regex: "\\s+")[0];
        String endTime = a.getEndTime().split( regex: "\\s+")[0];
        if (startTime.equals(currentDay) || endTime.equals(currentDay))
            return true;
        else
            return false;
    }).collect(toList());
    List<String> currentDayActivities = new ArrayList<String>();
    for (MonitoredData data : currentDayData) {
        currentDayActivities.add(data.getActivity());
    }
    List<String> currentDayDistinctActivities = currentDayActivities.stream().distinct().collect(toList());
    Map<String, Integer> currentDayActivityList = new HashMap<~>();
    for (String iterator : currentDayDistinctActivities) {
        long counter = currentDayActivities.stream().filter(a -> {
            return a.equals(iterator);
        }).count();
        currentDayActivityList.put(iterator, (int) counter);
    }
    activitiesPerDay.put(Integer.parseInt(currentDay.split( regex: "-")[2]), currentDayActivityList);
}
String toPrintTask4 = new String();
for (Map.Entry<Integer, Map<String, Integer>> entry : activitiesPerDay.entrySet()) {
    toPrintTask4 = toPrintTask4 + "Day: " + entry.getKey() + "\n";
    for (Map.Entry<String, Integer> entry1 : entry.getValue().entrySet()) {
        toPrintTask4 = toPrintTask4 + entry1.getKey() + " " + entry1.getValue() + "\n";
    }
    toPrintTask4 = toPrintTask4 + "\n";
}
```

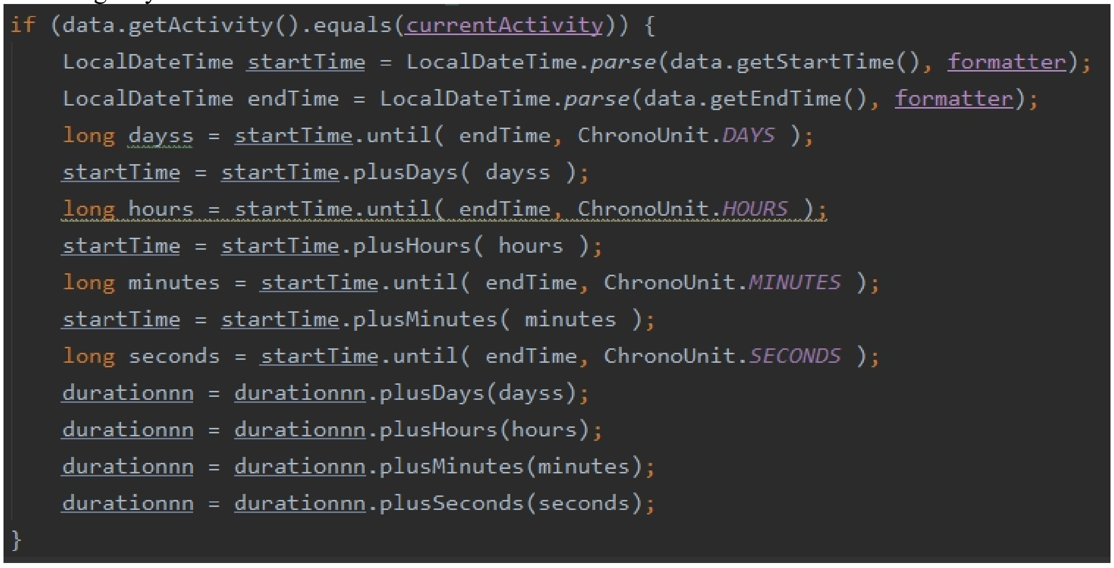
- declare a map of form Map<Integer, Map<String, Integer>> as required;
- iterate through each of the distinct days of the month and do the following;
- use streams and lambda expression to filter the given list of monitored data (finalList);
- collect into the currentDayData list the data for each day. Since some activities start in a day and end in other day, I considered that particular activity happening in both days;
- create and populate a new list of string (currentDayActivities) containing just the activity labels of the current day;
- create and fill in a map representing in the following step the value of the initial map;
- the map currentDayActivityList will represent having as key each distinct activity of the current day and as value the number of occurrences of that particular activity;
- finally, in the map activitiesPerDay we will add the day of the month, taken from the current day format and converted to string, as key and the previously explained map as value;
- then we build a suitable string and send it to the *writeToFile* method.

Task_5:



```java
/*************** TASK 5 ***************************************************************/
Map<String, LocalDateTime> entireDuration = new HashMap<~>();
for (String currentActivity : distinctActivities) {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    int i = 0;
    List<LocalDateTime> durationn = finalList.stream().map(a -> {
        String strr = "0001-01-01 00:00:00";
        LocalDateTime durationnn = LocalDateTime.parse(strr, formatter);
        for (MonitoredData data : finalList) {
            if (data.getActivity().equals(currentActivity)) {...}
        }
        return durationnn;
    }).collect(Collectors.toList());
    entireDuration.put(currentActivity, durationn.get(i));
    i++;
}
String toPrintTask5 = new String();
for (Map.Entry<String, LocalDateTime> entry : entireDuration.entrySet()) {
    toPrintTask5 = toPrintTask5 + entry.getKey() + " "  + (entry.getValue().getDayOfMonth() - 1) + " day(s) "
            + entry.getValue().getHour() + ":" + entry.getValue().getMinute() + ":" + entry.getValue().getSecond() + "\n";
}
toPrintTask5 = toPrintTask5 + "\n";
writeToFile(toPrintTask5, taskNumber: 5);
```

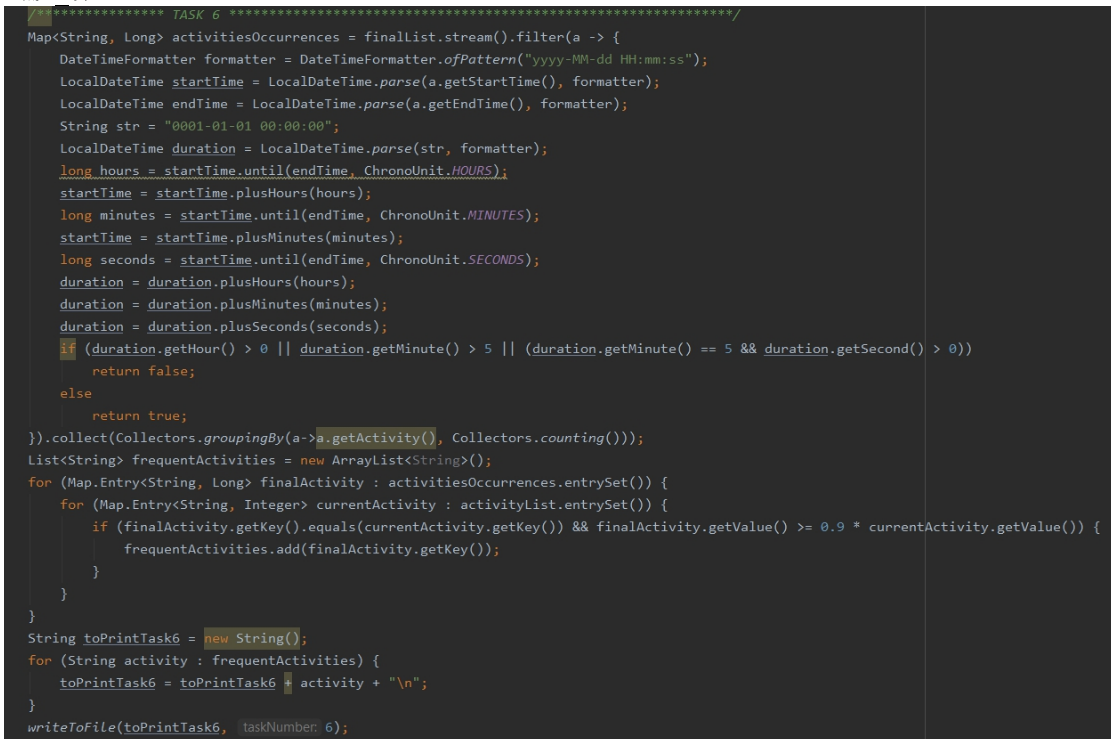
- declare a map with String as key and LocalDateTime as value, which will represent the activity label and duration in local date time format of the corresponding activity;
- iterate through each distinct activity;
- initialize each time a duration of the activity of 1 day, 1 month and 1 year, because it is no possible to initialize them with 0 as well as for the hours, minutes and seconds;
- each time we encounter the current activity we are working on, we compute the duration the following way:



```java
if (data.getActivity().equals(currentActivity)) {
    LocalDateTime startTime = LocalDateTime.parse(data.getStartTime(), formatter);
    LocalDateTime endTime = LocalDateTime.parse(data.getEndTime(), formatter);
    long dayss = startTime.until( endTime, ChronoUnit.DAYS );
    startTime = startTime.plusDays( dayss );
    long hours = startTime.until( endTime, ChronoUnit.HOURS );
    startTime = startTime.plusHours( hours );
    long minutes = startTime.until( endTime, ChronoUnit.MINUTES );
    startTime = startTime.plusMinutes( minutes );
    long seconds = startTime.until( endTime, ChronoUnit.SECONDS );
    durationnn = durationnn.plusDays(dayss);
    durationnn = durationnn.plusHours(hours);
    durationnn = durationnn.plusMinutes(minutes);
    durationnn = durationnn.plusSeconds(seconds);
}
```

- use the method "until" available in the LocalDateTime class to compute days, hours, minutes and seconds from the start time until the end time and then add each of them to the initialized variable duration;
- collect the duration of each activity in a list of LocalDateTime elements;
- add in the initial map the current activity as key and duration as value;
- finally build a suitable string and send it to the *writeToFile* method.

Task_6:

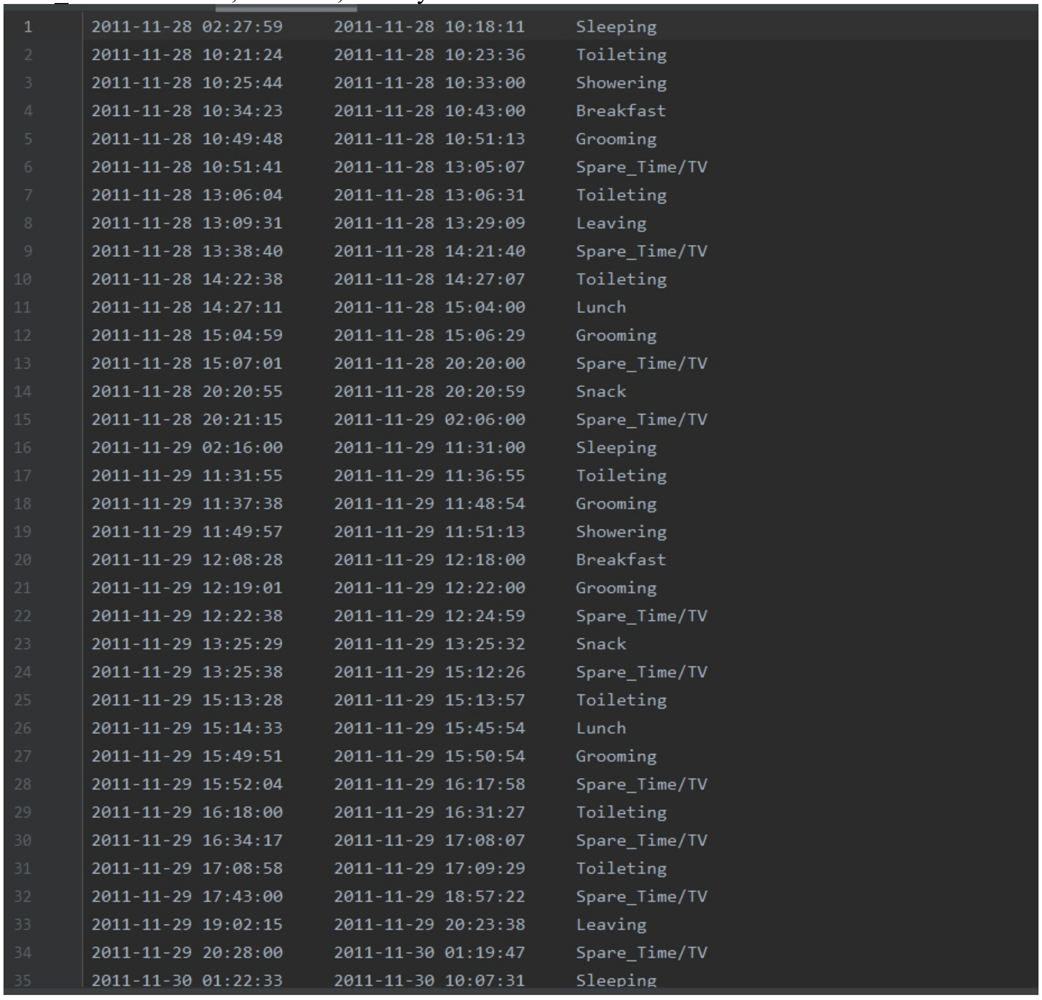


```
/*************** TASK 6 ***************************************************************/
Map<String, Long> activitiesOccurrences = finalList.stream().filter(a -> {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    LocalDateTime startTime = LocalDateTime.parse(a.getStartTime(), formatter);
    LocalDateTime endTime = LocalDateTime.parse(a.getEndTime(), formatter);
    String str = "0001-01-01 00:00:00";
    LocalDateTime duration = LocalDateTime.parse(str, formatter);
    long hours = startTime.until(endTime, ChronoUnit.HOURS);
    startTime = startTime.plusHours(hours);
    long minutes = startTime.until(endTime, ChronoUnit.MINUTES);
    startTime = startTime.plusMinutes(minutes);
    long seconds = startTime.until(endTime, ChronoUnit.SECONDS);
    duration = duration.plusHours(hours);
    duration = duration.plusMinutes(minutes);
    duration = duration.plusSeconds(seconds);
    if (duration.getHour() > 0 || duration.getMinute() > 5 || (duration.getMinute() == 5 && duration.getSecond() > 0))
        return false;
    else
        return true;
}).collect(Collectors.groupingBy(a->a.getActivity(), Collectors.counting()));
List<String> frequentActivities = new ArrayList<String>();
for (Map.Entry<String, Long> finalActivity : activitiesOccurrences.entrySet()) {
    for (Map.Entry<String, Integer> currentActivity : activityList.entrySet()) {
        if (finalActivity.getKey().equals(currentActivity.getKey()) && finalActivity.getValue() >= 0.9 * currentActivity.getValue()) {
            frequentActivities.add(finalActivity.getKey());
        }
    }
}
String toPrintTask6 = new String();
for (String activity : frequentActivities) {
    toPrintTask6 = toPrintTask6 + activity + "\n";
}
writeToFile(toPrintTask6, taskNumber: 6);
```

- create a map having as key a string representing the activity label and as value a long representing the number of occurrences of the corresponding activity through the entire monitoring period;
- stream through each element in the list of monitoredData elements;
- compute the duration of each activity as in the previous task;
- choose (through filtering) just the activities that have a duration less or equal to 5 minutes;
- collect the results in the initially declared map, putting as key the activity label and as value the number of times that particular activity that happened to have less than 5 minutes duration;
- iterate though the map of activity labels and their occurrences of duration less than 5 minutes and the map of activity labels and their number of occurrences regardless of the duration;
- chose the activities from the first map for which the values are with at least 90% greater than the ones in the second map;
- finally build a suitable string and send it to the *writeToFile* method.

5. Results

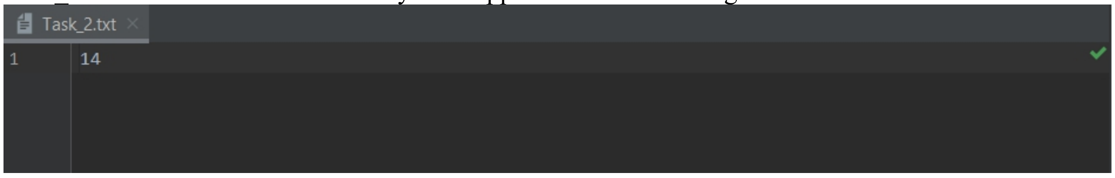
Task_1.txt: start time, end time, activity label



```
2011-11-28 02:27:59	2011-11-28 10:18:11	Sleeping
2011-11-28 10:21:24	2011-11-28 10:23:36	Toileting
2011-11-28 10:25:44	2011-11-28 10:33:00	Showering
2011-11-28 10:34:23	2011-11-28 10:43:00	Breakfast
2011-11-28 10:49:48	2011-11-28 10:51:13	Grooming
2011-11-28 10:51:41	2011-11-28 13:05:07	Spare_Time/TV
2011-11-28 13:06:04	2011-11-28 13:06:31	Toileting
2011-11-28 13:09:31	2011-11-28 13:29:09	Leaving
2011-11-28 13:38:40	2011-11-28 14:21:40	Spare_Time/TV
2011-11-28 14:22:38	2011-11-28 14:27:07	Toileting
2011-11-28 14:27:11	2011-11-28 15:04:00	Lunch
2011-11-28 15:04:59	2011-11-28 15:06:29	Grooming
2011-11-28 15:07:01	2011-11-28 20:20:00	Spare_Time/TV
2011-11-28 20:20:55	2011-11-28 20:20:59	Snack
2011-11-28 20:21:15	2011-11-29 02:06:00	Spare_Time/TV
2011-11-29 02:16:00	2011-11-29 11:31:00	Sleeping
2011-11-29 11:31:55	2011-11-29 11:36:55	Toileting
2011-11-29 11:37:38	2011-11-29 11:48:54	Grooming
2011-11-29 11:49:57	2011-11-29 11:51:13	Showering
2011-11-29 12:08:28	2011-11-29 12:18:00	Breakfast
2011-11-29 12:19:01	2011-11-29 12:22:00	Grooming
2011-11-29 12:22:38	2011-11-29 12:24:59	Spare_Time/TV
2011-11-29 13:25:29	2011-11-29 13:25:32	Snack
2011-11-29 13:25:38	2011-11-29 15:12:26	Spare_Time/TV
2011-11-29 15:13:28	2011-11-29 15:13:57	Toileting
2011-11-29 15:14:33	2011-11-29 15:45:54	Lunch
2011-11-29 15:49:51	2011-11-29 15:50:54	Grooming
2011-11-29 15:52:04	2011-11-29 16:17:58	Spare_Time/TV
2011-11-29 16:18:00	2011-11-29 16:31:27	Toileting
2011-11-29 16:34:17	2011-11-29 17:08:07	Spare_Time/TV
2011-11-29 17:08:58	2011-11-29 17:09:29	Toileting
2011-11-29 17:43:00	2011-11-29 18:57:22	Spare_Time/TV
2011-11-29 19:02:15	2011-11-29 20:23:38	Leaving
2011-11-29 20:28:00	2011-11-30 01:19:47	Spare_Time/TV
2011-11-30 01:22:33	2011-11-30 10:07:31	Sleeping
```

The list goes on until 248.

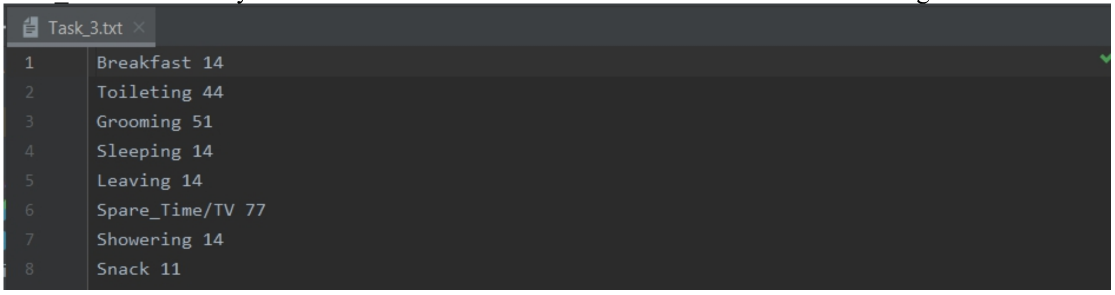
Task_2.txt: the number of distinct days that appear in the monitoring data



Task_2.txt

```
14
```

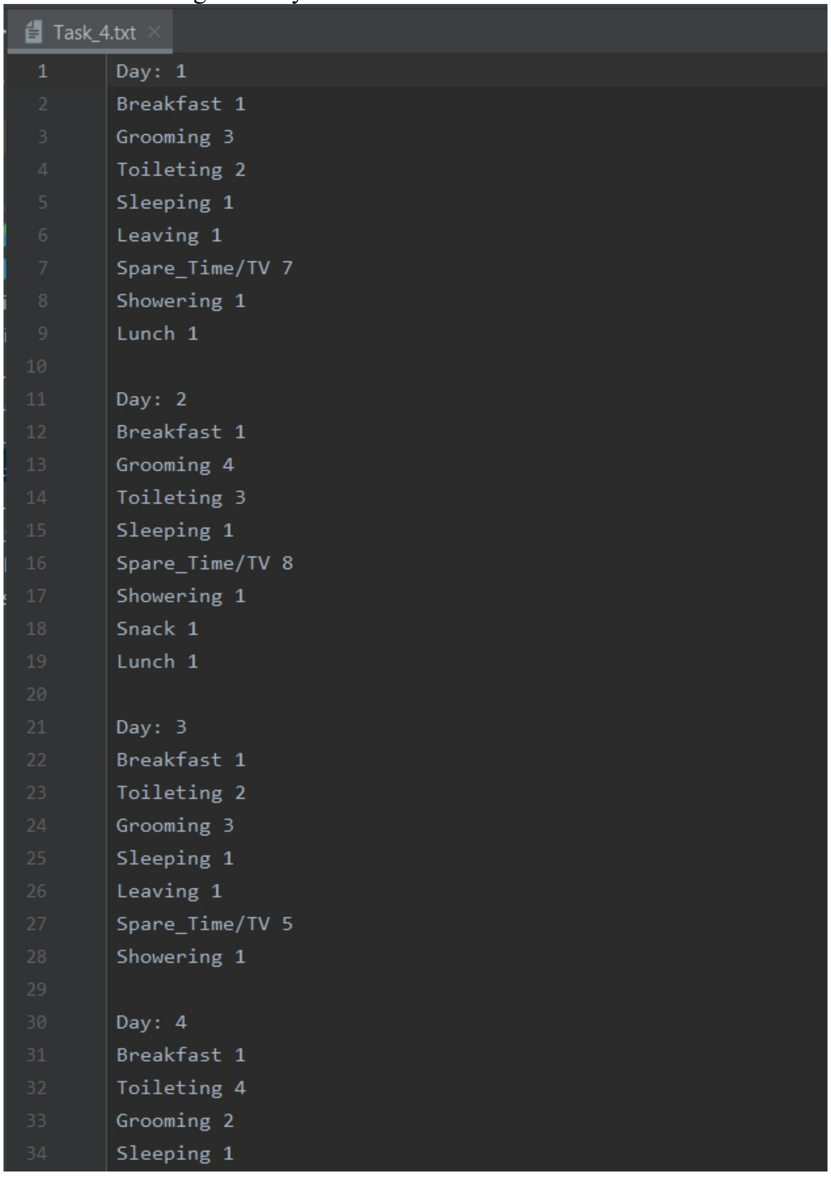
Task_3.txt: the activity label and the number of occurrences over the entire monitoring data



Task_3.txt

```
Breakfast 14
Toileting 44
Grooming 51
Sleeping 14
Leaving 14
Spare_Time/TV 77
Showering 14
Snack 11
```

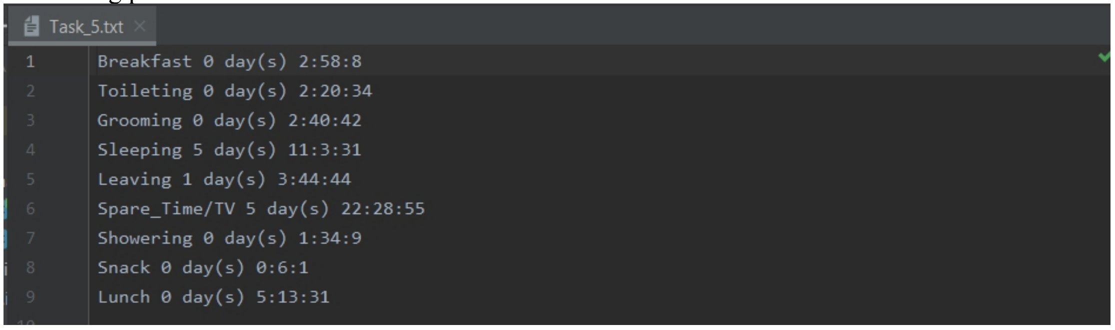
Task_4.txt: for each day of the month, there has been displayed the activities involved and their occurrences through the day



```
Task_4.txt
Day: 1
Breakfast 1
Grooming 3
Toileting 2
Sleeping 1
Leaving 1
Spare_Time/TV 7
Showering 1
Lunch 1

Day: 2
Breakfast 1
Grooming 4
Toileting 3
Sleeping 1
Spare_Time/TV 8
Showering 1
Snack 1
Lunch 1

Day: 3
Breakfast 1
Toileting 2
Grooming 3
Sleeping 1
Leaving 1
Spare_Time/TV 5
Showering 1

Day: 4
Breakfast 1
Toileting 4
Grooming 2
Sleeping 1
```

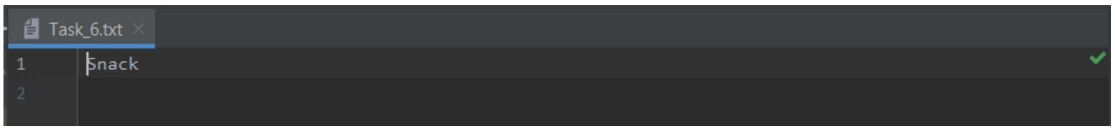
Days are between 1 to 11, then from 28 to 30.

Task_5.txt: the activity label and their duration in days, hours, minutes and seconds over the entire monitoring period



```
Task_5.txt
Breakfast 0 day(s) 2:58:8
Toileting 0 day(s) 2:20:34
Grooming 0 day(s) 2:40:42
Sleeping 5 day(s) 11:3:31
Leaving 1 day(s) 3:44:44
Spare_Time/TV 5 day(s) 22:28:55
Showering 0 day(s) 1:34:9
Snack 0 day(s) 0:6:1
Lunch 0 day(s) 5:13:31
```

Task_6.txt: Activities that have more than 90% of the monitoring records with duration less than 5 minutes



```
Snack
```

6. Conclusions

Lessons learned:

- how to work with lambda expressions and streams
- better understanding of the reading, generating and writing from/to a .txt file
- many statistics can be easily implemented using streams and lambda expression.

Possible feature improvements:

- to define functional interfaces for each task;
- implement classes for each task;
- many other statistics required by the user.

7. Bibliography

[1] List:
https://docs.oracle.com/javase/8/docs/api/java/util/List.html
[2] Map:
https://docs.oracle.com/javase/8/docs/api/java/util/Map.html
[3] Lambda expressions and streams:
https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html
[4] Reading and writing from/to a text file:
https://mkyong.com/java8/java-8-stream-read-a-file-line-by-line/