TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA
ROMANIA

ASSIGNMENT 5

PROCESSING SENSOR DATA OF DAILY LIVING ACTIVITIES

Student: Mitrica Livia-Maria

Group: 30424

Techer Assistant: Chifu Viorica

Contents

Objective

The main objective of this assignment is designing, implementing and testing an application for analysing 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_timeand 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.

Problem analysis

The problem is divided into 6 smaller tasks which must be accomplished as it follows:

1. Defining a class MonitoredData with 3 fields (start time, end time and activity) and create a list of objects of type MonitoredData.
2. Counting the distinct days that appear in the monitoring data.
3. Counting the number of times each activityhas appeared over the entire monitoring period
4. Counting for how many times each activity has appeared for each day over the monitoring period.
5. For each activity computing the entire duration over the monitoring period.
6. Filtering the activities that have more than 90% of the monitoring records with duration less than 5 minutes.

The user will run the application given as a jar file and obtain the requested results for the 6 tasks in files named accordingly to the task.

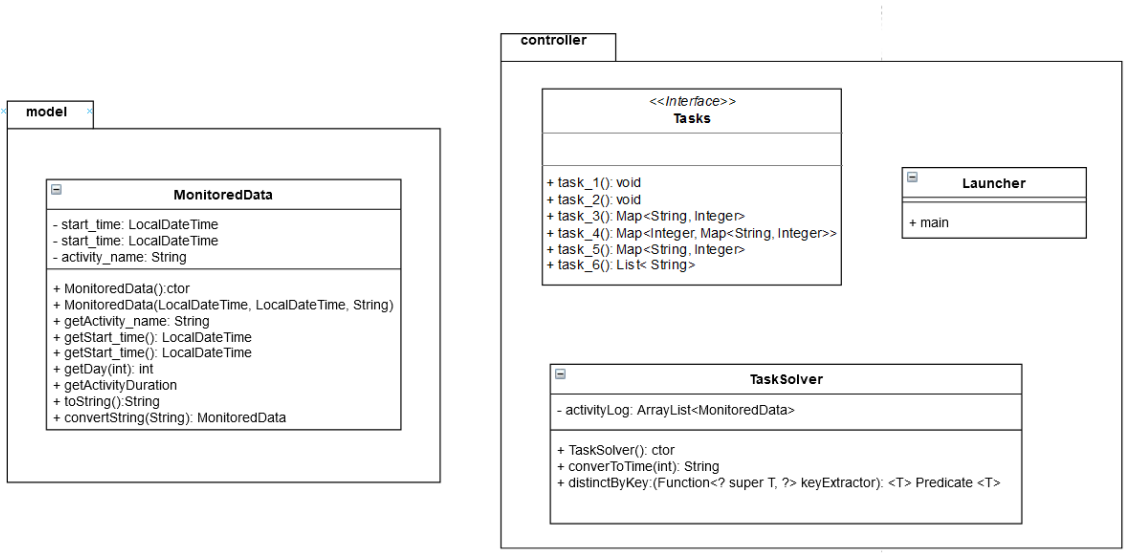
Problem design

The implementation will be done using Java 1.8 specific techniques, which are Stream Processing and Lambda Expressions.

Taking into account that the user is not requested to enter himself the file where the activity log is stored, we will only assume that the path must be given in the program and that the file must also be found in the same folder as the application.

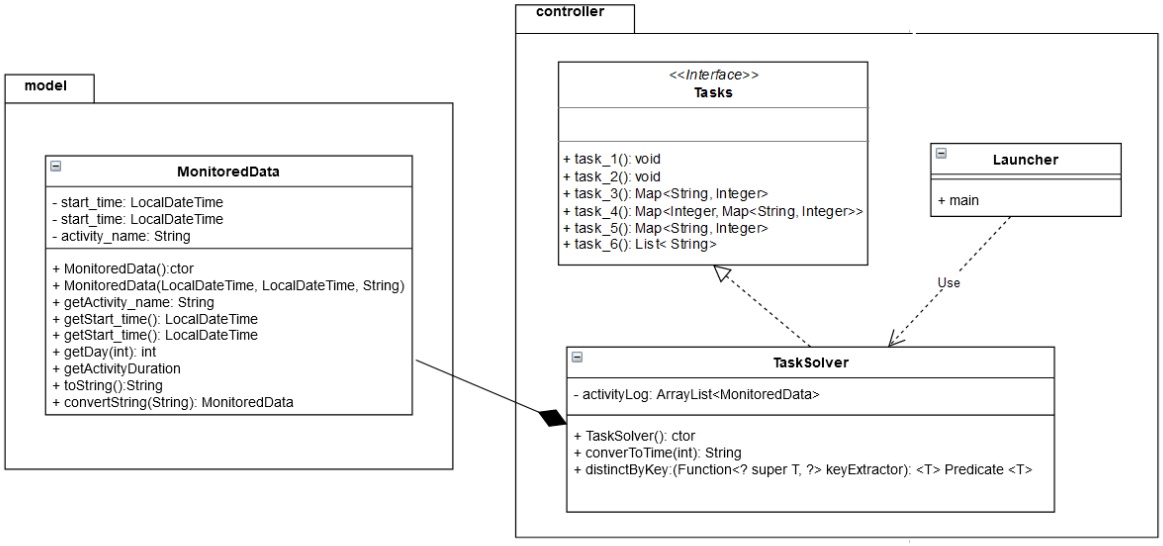
I have decided to divide the problem into 2 packages, model and controller, similarly to MVC design pattern, but this time, no user interface was required.

The following picture illustrates the classes I have used and the packages they belong to.



The first package model, contains the class MonitoredData, is designed for modelling the special type of data needed in this application, the data registered by the sensors, which are the start time, the end time and the name of the activity.

The second package, controller contains the interface with all the tasks requested, a class designed for solving all the tasks – TaskSolver and a Launcher, where the application is started and the information is printed in the corresponding files.



In this picture the relations that establish between packages can be noticed.

The interface, implemented by the TaskSolver represents a collection of all the tasks requested to be performed by the application.

For this application I have decided to use as data types mainly LocalDateTime, String, Integer, Duration, ArrayList, List and int.

LocalDateTime seemed to be the best for recording the beginning and end of each activity, since it allowed simple access for both date and time.

String was mainly used for storing the name of the performed activities.

Integer was mostly used for counting activity frequency, similarly for primitive int.

Duration was used for computing the time difference between the start time and end time of an activity, since it was the most suitable for saving the subtraction of two variables of type LocalDateTime.

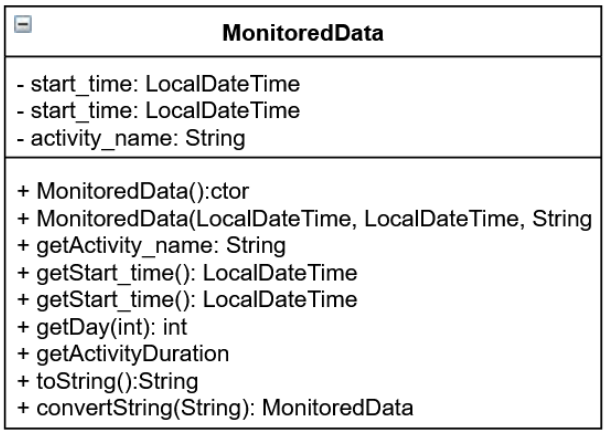
ArrayList and List were both used for storing the activity names complying to the requested filters.

Problem Implementation

In this section I will detail each class and its important methods.

Package model:

MonitoredData



- The main attributes are the fields requested, namely, start_time, end_time, activity_name.
- The methods include:
  - Contrusctors
  - Getters for the fields
  - getDay: returns the day computed from the beginning of the log period, for example the history begins on 20/01/2020, therefore 21/01/2020 represents the second day; the function takes as argument the first day in the activity history. It takes as reference the day of year (not of the month)

```
public int getDay(int first_day){
    return this.getStart_time().getDayOfYear()-first_day+1;
}
```

  - getActivityDuration: computes the duration of an activity in seconds, using the class Duration, extremely suitable in situations that measure machine-based time

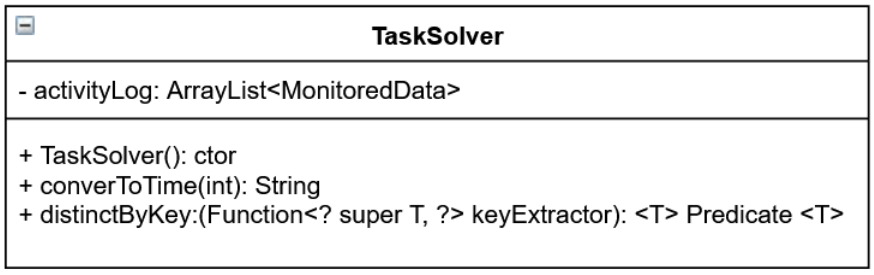
```
public int activityDuration(){
    return (int) Math.abs(Duration.between(this.getStart_time(),this.getEnd_time()).toSeconds());
}
```

- convertString: transforms a line of data in the activity liog file into an entry of type MonitoredData. Since there are multiple spaces between the date and the activity, these are cleared until only one remains, the string that results is split according to the empty spaces left.
  A formatter for the LocalDateTime is created, in order to specify the pattern we want our start_time and end_time to have. After obtaining the correct string corresponding to the beginnig and end of each activity we parse it to the newly created variable of type LocalDateTime and eventually to the constructor of an entry of type MonitoredData.

```
public MonitoredData convertString (String s){
    s.trim().replaceAll("\\s+", "");
    String[] words=s.split("\\s+");

    String date_time_start = words[0]+" "+words[1];
    String date_time_end = words[2]+" "+words[3];

    String activity = words[4];
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    LocalDateTime start = LocalDateTime.parse(date_time_start, formatter);
    LocalDateTime end = LocalDateTime.parse(date_time_end, formatter);
    MonitoredData data = new MonitoredData(start, end, activity);

    return data;
}
```

Package controller

TaskSolver



- The main and only attribute is the ArrayList of MonitoredData entries
- This class implements the interface Tasks, therefore apart from the 2 methods represented in the picture, it also implements the methods task_1, task_2, task3, task_4, task_5, task_6
- The methods include:
  - A constructor which creates an empty list of MonitoredData items
  - Task_1 – this method is the first to be performed when starting the application. It reads line by line the data in the Activities.txt file and converts them into an object of type MonitoredData which is inserted in the list ActivityLog afterwords

```
@Override
public void task_1() throws FileNotFoundException {

    String fileName = "Activities.txt";
    String outputFile = "Task_1.txt";
    FileOutputStream fileOutputStream = new FileOutputStream(outputFile, true);
    PrintStream outFile = new PrintStream(fileOutputStream);

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

        stream.forEach(s ->
        {
            MonitoredData data = new MonitoredData();
            data = data.convertString(s);
            activityLog.add(data);
            outFile.println(data.toString());
        });
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- Task_2 – this method counts the number of distinct activities the sensors register; it uses the method distinctByKey[4], in order to ensure that an activity will not be added twice

```
@Override
public void task_2() throws FileNotFoundException {
    int count = (int) activityLog.stream().filter(distinctByKey(a ->  a.getStart_time().getDayOfYear())).count();
    String outputFile = "Task_2.txt";
    FileOutputStream fileOutputStream = new FileOutputStream(outputFile, true);
    PrintStream outFile = new PrintStream(fileOutputStream);
    outFile.println("Task2");
    outFile.println("Distinct days: "+count);
}
```

- Task_3 – this method returns the frequency over the given period of time of each activity. The elements are collected in a map, where the key is a String, the activity name and the value represents the frequency. Basically, each time a certain activity is encountered, the number of appearances of that specific activity is increased by 1.

```
@Override
public Map<String,  Integer>  task_3() {
   Map<String, Integer> countActivityFrequency = activityLog.stream().
         collect(groupingBy(MonitoredData::getActivityName, summingInt(d-> 1)));

   return countActivityFrequency;
}
```

- Task_4 – this method returns a map having the key the current day of the monitorization period and the another map where the second key is the activity and the value is the frequency of the activity in that certain day. Computing the first day of the monitoring period is done relative to the day of the year, not of the month.

```
@Override
public  Map<Integer, Map<String, Integer>> task_4() {
   int firstDay = activityLog.get(0).getStart_time().getDayOfYear();

   Map<Integer, Map<String, Integer>> countActivityPerDayFrequency = activityLog.stream().
         collect(groupingBy(a->a.getDay(firstDay),
               groupingBy(MonitoredData::getActivityName, summingInt(d-> 1))));

   return countActivityPerDayFrequency;
}
```

- Task_5 – this method returns a map where the key represents the name of the activity and the value is the total activity duration over the entire period of monitorization. The value represents the time in seconds. Although it was requested to use Map<String, LocalTime>, LocalTime can only be used for a period of maximum 24 hours (23:59:59), so, in order to obtain a correct result I stored the value in seconds in an Integer.

```
public Map<String, Integer> task_5() {
   Map<String, Integer> countActivityFrequency = activityLog.stream().
         collect(groupingBy(MonitoredData::getActivityName, summingInt(MonitoredData::activityDuration)));

   return countActivityFrequency;
}
```

- task_6 – this method filters the activities that havemore than 90% of the monitoring records with duration less than 5 minutes and returns a collection of the results in a List<String> containing only the distinct activity names.

Firstly, it computes the frequency in seconds of each activity, then the the number frequency of each activity of less than 5 minutes. Finaly, for each activity, if it has entries of less than five minutes, determine if they are at least 90% of the monitored records.

```
@Override
public  List<String> task_6() {

   Map<String, Integer> countActivityFrequency = task_3();
   Map<String, Integer> countActivityLess5Mins = activityLog.stream().filter(a->a.activityDuration()<5*30)
         .collect(groupingBy(MonitoredData::getActivityName, summingInt(d-> 1)));

   List<String> finalActivityList = activityLog.stream().filter(a->{
      if (countActivityLess5Mins.get(a.getActivityName())!=null){

if(countActivityLess5Mins.get(a.getActivityName())>=0.9*countActivityFrequency.get(a.getActivityName())){
            return true;
         }
      }
      return false;
   }).map(MonitoredData::getActivityName).distinct().collect(Collectors.toList());

   return finalActivityList;
}
```

- distinctByKey – this method enables filtering the objects considering only one property.

  The Stream API provides the *distinct()* method that returns different elements of a list based on the *equals()* method of the *Object* class. However, it becomes less flexible if we want to filter by a specific attribute. The alternative I have chosen is to write a filter that maintains the state using a stateful *Predicate:*

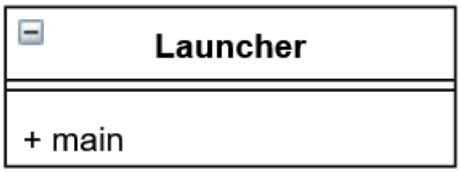
```
public static <T> Predicate<T> distinctByKey(
      Function<? super T, ?> keyExtractor) {

   Map<Object, Boolean> seen = new ConcurrentHashMap<>();
   return t -> seen.putIfAbsent(keyExtractor.apply(t), Boolean.TRUE) == null;
}
```

- convertToTime – this method receives an integer representing the total duation of an activity in seconds which is then converted into a string of format „d days hh:mm:ss” (if the duration is smaller than 1 day, this might be missing in the print)

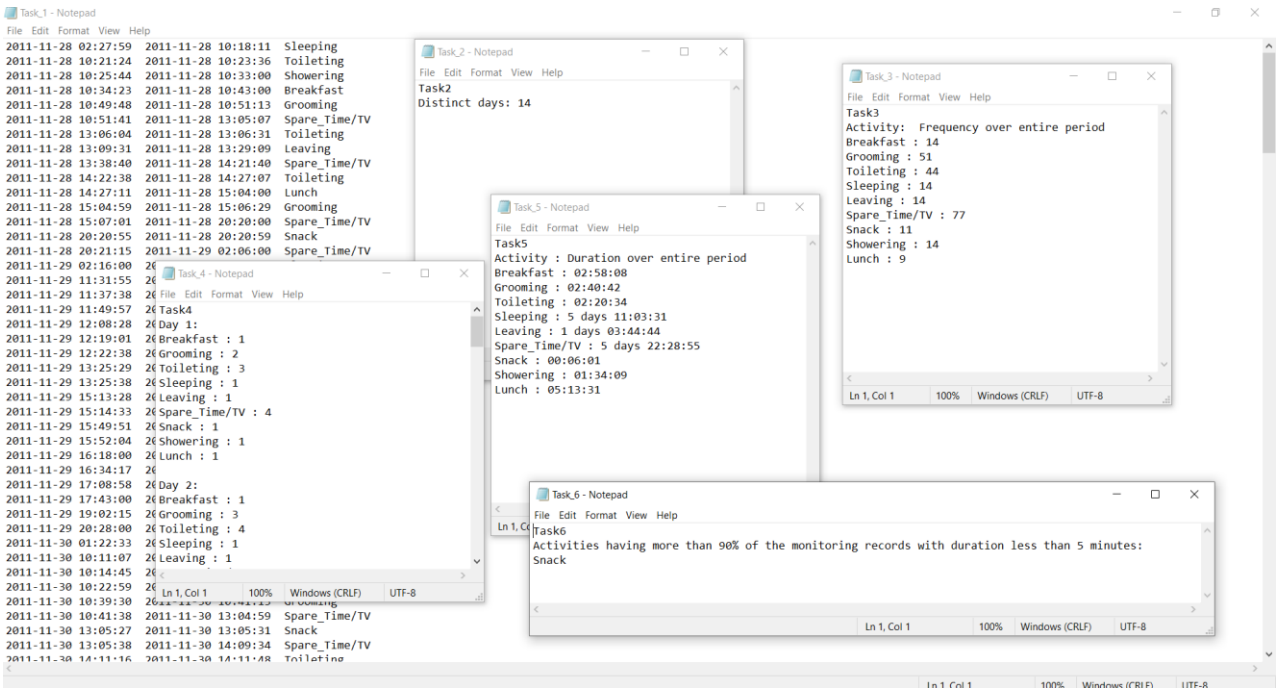
```java
public static String convertToTime(int timeDifference){
    Duration d = Duration.parse("PT"+timeDifference+"S");
    int seconds = d.toSecondsPart();
    int minutes = d.toMinutesPart();
    int hours = d.toHoursPart();
    long days = d.toDaysPart();
    String h = (hours<=9) ?  "0"+hours : String.valueOf(hours);
    String m = (minutes<=9) ?  "0"+minutes : String.valueOf(minutes);
    String s = (seconds<=9) ?  "0"+seconds : String.valueOf(seconds);
    String string = h +":"+ m +":"+ s;

    string = (days>0) ? days + " days " + string : string ;

    return string;
}
```

Launcher

- This class contains only the main method for starting the application. In this method a new TaskSolver is created and each method for solving the tasks is called. For each method the corresponding return type is printed in a file named accordingly.



Results



Conclusions

This project helped me learn more about lambda expressions and how they can help in achieving readabilty, eliminating shadow variables. It also helped me obtain a cleaner code. This assignment also gave me the opportunity to get more familiar with LocalDateTime, given the fact that it seemed to suit best the requirements for activity start and end time.

As further improvments, a user interface could be created, where the user might choose the task he wants to see the results for. Also, more filters could be applied, for example instead of choosing activities with more than 90% frequency of 5 minutes, the user could choose himself the duration.

Bibliography

[1] Materials provided at the lecture

[2] https://mkyong.com/java8/java-8-stream-read-a-file-line-by-line/

[3] https://www.logicbig.com/how-to/code-snippets/jcode-java-8-streams-stream-foreach.html

[4] https://www.baeldung.com/java-streams-distinct-by

[5] https://stackoverflow.com/questions/24491243/why-cant-i-get-a-duration-in-minutes-or-hours-in-java-time