Todea Tudor-Stefan
Group 30424

Programming Techniques
May 2020

Processing Sensor Data Of Daily Living Activities

1. Purpose of the laboratory work 2
2. Analysis, modelling and use-case scenarios 2
3. Development 3
 3.1 Package Diagram 3
 3.2 Class Diagram 3
4. Implementation 4
 4.1 Model Package 4
  4.1.1 MonitoredData 4
 4.2 Start Package 4
  4.2.1 Start 4
 4.3 Tasks Package 4
  4.3.1 Task1 4
  4.3.2 Task2 5
  4.3.3 Task3 5
  4.3.4 Task4 5
  4.3.5 Task5 6
  4.3.6 Task6 6
5. Conclusions 7
6. Bibliography 7

1. Purpose of the laboratory work

We were required to develop an application that processes information provided by a sensor regarding daily activities. The implementation of this application must be done using Java functional programming techniques, such as utilising streams, lambda expressions and making use of the predefined Functional Interfaces such as Predicate, Function, Supplier, Client etc.

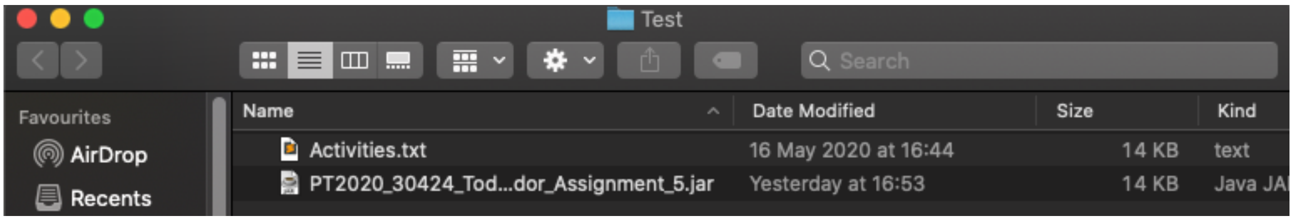
We must implement 6 different tasks and print their keys into a text file with the format "**Task_nb.txt**". Each task is implemented in a separate class. We will discus this further during the 3rd and 4th chapter.

2. Analysis, modelling and use-case scenarios

We were given a text file named "Activities.txt" containing a list of activities, as well as their start time and end time. We were required to handle the data such that all the presented tasks could be completed.

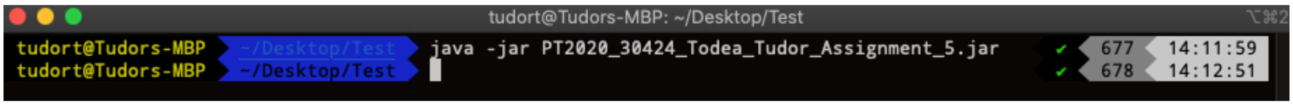
In order to run our application, the input file must have the name "Activities.txt" and be placed in the folder of the project. We were only required to handle the given file, but any text file that is given and has the same format could be used in order to filter the data through the 6 tasks that were implemented.

We place the .jar file and the "Activities.txt" file in the same folder as in the example below:



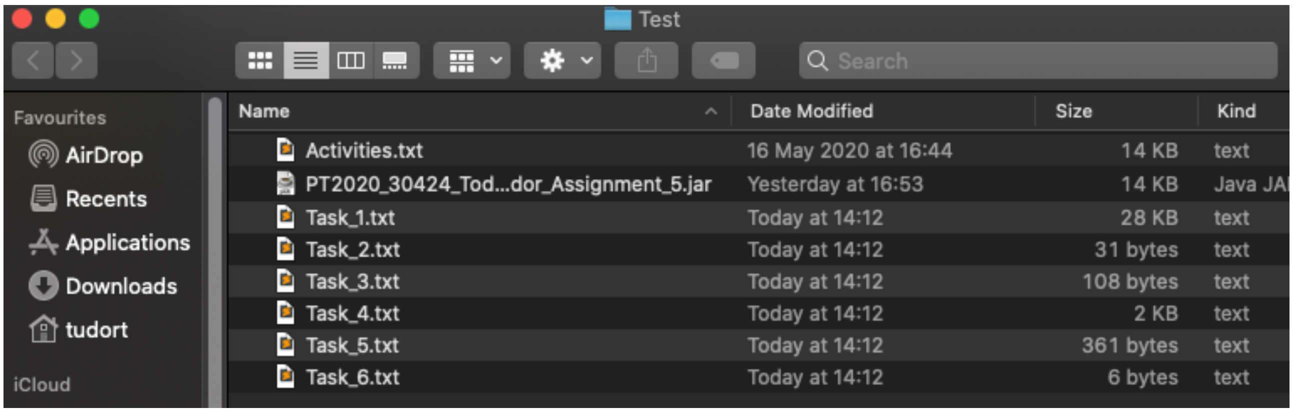
Name	Date Modified	Size	Kind
Activities.txt	16 May 2020 at 16:44	14 KB	text
PT2020_30424_Tod...dor_Assignment_5.jar	Yesterday at 16:53	14 KB	Java JA

We then run the following command from the terminal:



```
tudort@Tudors-MBP:  ~/Desktop/Test
tudort@Tudors-MBP  ~/Desktop/Test  java -jar PT2020_30424_Todea_Tudor_Assignment_5.jar   ✔  677  14:11:59
tudort@Tudors-MBP  ~/Desktop/Test  ▌                                                     ✔  678  14:12:51
```

After running the application we should see the results placed in the same folder as our two previous files:



Name	Date Modified	Size	Kind
Activities.txt	16 May 2020 at 16:44	14 KB	text
PT2020_30424_Tod...dor_Assignment_5.jar	Yesterday at 16:53	14 KB	Java JA
Task_1.txt	Today at 14:12	28 KB	text
Task_2.txt	Today at 14:12	31 bytes	text
Task_3.txt	Today at 14:12	108 bytes	text
Task_4.txt	Today at 14:12	2 KB	text
Task_5.txt	Today at 14:12	361 bytes	text
Task_6.txt	Today at 14:12	6 bytes	text

Finally, we have our results for each task as text files, which can be then analysed.

Since there is no alternative running scenario, a use-case list or diagram is not needed. There is no other way of manipulating our application.

3. Development

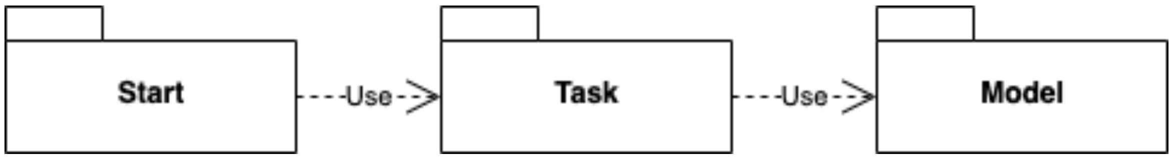
The code of the application is split into three packages: Model, Start and Tasks.

The Model package contains the MonitoredData class which represents an activity containing the name of the activity, its start time and its ending time.

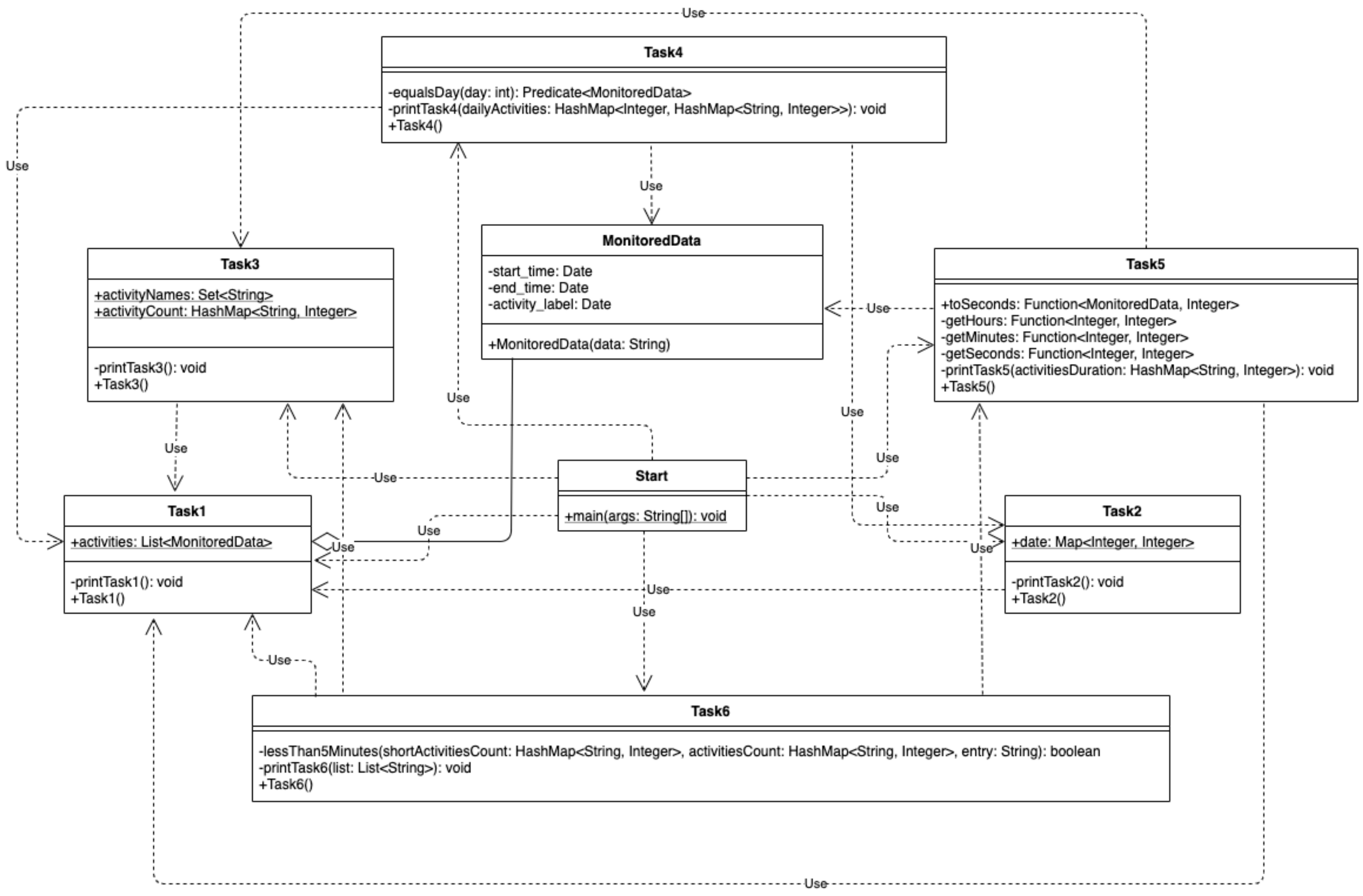
The Start package contains the Start class which is made up of the main method of our application, in which we create 6 new objects, one for each task that we have to complete.

The Tasks package contains 6 classes, which represent each individual task. Each constructor solves its specific task. The printing method for each task is also present in their respective class.

3.1 Package Diagram



3.2 Class Diagram


Task4
-equalsDay(day: int): Predicate<MonitoredData>
-printTask4(dailyActivities: HashMap<Integer, HashMap<String, Integer>>): void
+Task4()
MonitoredData
-start_time: Date
-end_time: Date
-activity_label: Date
+MonitoredData(data: String)
Task3
+activityNames: Set<String>
+activityCount: HashMap<String, Integer>
-printTask3(): void
+Task3()
Task5
+toSeconds: Function<MonitoredData, Integer>
-getHours: Function<Integer, Integer>
-getMinutes: Function<Integer, Integer>
-getSeconds: Function<Integer, Integer>
-printTask5(activitiesDuration: HashMap<String, Integer>): void
+Task5()
Start
+main(args: String[]): void
Task1
+activities: List<MonitoredData>
-printTask1(): void
+Task1()
Task2
+date: Map<Integer, Integer>
-printTask2(): void
+Task2()
Task6
-lessThan5Minutes(shortActivitiesCount: HashMap<String, Integer>, activitiesCount: HashMap<String, Integer>, entry: String): boolean
-printTask6(list: List<String>): void
+Task6()
Use

The constructor of each class contains the functional code that does the work. Everything has been implemented, as much as possible, using predefined Functional Interfaces and lambda expressions.

4. Implementation

4.1 Model Package

4.1.1 MonitoredData

The MonitoredData class represents the only Model class we have in this project. It contains three class fields: start_time: Date, end_time: Date and activity_label: String.

The constructor takes a line of data as an input and parses it in the following way:

```
String[] tokens = data.split("\t\t");
```

This is done so because the end_time and start_time and end_time and activity_label are separated by two tabs inside our input file. The format of the Dates is given as:

```
SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss”);
```

The start_time gets the firs token parsed as a Date with the above mentioned format, the end_time gets the second token parsed in the same way, and the activity_label gets the third token after it has been stripped of tabs, since some activity names may have a tab or two at the end of their names, before the new line.

```
this.activity_label = tokens[2].replace("\t", "");
```

4.2 Start Package

4.2.1 Start

The Start class contains the main method which creates a new instance for each one of the 6 Task classes:

```
public static void main(String[] args) throws IOException {
        new Task1();
        new Task2();
        new Task3();
        new Task4();
        new Task5();
        new Task6();
}
```

4.3 Tasks Package

4.3.1 Task1

For the first task we were required to parse the input file using streams and to create a MonitoredData object for each activity.

We read the lines of the file into a Stream object, then placed all the lines inside a List:

```
Stream<String> fileStream =  Files.lines(Paths.get("Activities.txt"));
List<String> list = fileStream.collect(Collectors.toList());
```

Then, for each line of the list we created and stored in the *activities* field a new MonitoredData object. The *activities* field is an ArrayList of MonitoredData. It is public and static because we will use this list for more tasks:

```
list.forEach(item -> activities.add(new MonitoredData(item));
```

At the end, we print the results in “Task_1.txt”.

4.3.2 Task2

For the second task we had to count the number of individual days that occur in the data set. We have done that by creating a Map with the day as the key and the month as the value (we will use that in one of the next tasks), so that we will not have duplicates, and the final size of the Map will be the number of individual days:

```
date = Task1.activities
        .stream()
        .collect(Collectors.toMap(k -> k.getStart_time().getDate(), v -> v.getStart_time().getMonth(), (oldV,
newV) -> oldV));
```

We take the stream of activities from the first task and we collect them in a map. Every time we encounter a new value (an entry that is in the same day) we replace the value with either the old value or the new one. It does not matter in our case since the days do not repeat in two consecutive months (we don't have 1st of December and 1st of November). This could also have been done with a Set. This Map is held in a public static variable called *date* since we will use it further in the next tasks.

4.3.3 Task3

For the third task we had to count how many times each activity has appeared during the monitoring period. We created a public static list called *activityNames* (we'll use it later) containing all the names of individual activities. We then take each activity name and search for all the activities with the same name in the list of MonitoredData objects. When we find such an activity, we place it in a map which has the name of the activity as the key and the number of occurrences as the value. We keep this map as a public static variable since we will need it later:

```
activityNames.forEach(activityName ->
        Task1.activities
                .stream()
                .filter(activity -> activity.getActivity_label().equals(activityName))
                .forEach(activity -> {
                        activityCount.put(activityName, activityCount.get(activityName) + 1);
                }));
```

At the end we print the result in the "Task_3.txt" file.

4.3.4 Task4

For the fourth task we had to count how many times each activity occurs during every single day of our sample data. This means we have to design a HashMap which takes the day (an Integer) as key, and as the value a HashMap which has the activity name as the key and the number of occurrences that day as the value. We solve this problem by taking every day (using the Map from Task 2) and for each day we iterate over the whole list of activities. When we find an activity that happens during that day, we add it to the *dailyActivityCount* Hash Map. After we are done going through the whole list of activities, we map this *dailyActivityCount* to the larger HashMap, *dailyActivities*, and move on to the next day:

```
Task2.date
    .entrySet()
    .stream()
    .map(Map.Entry::getKey)
    .collect(Collectors.toList())
    .forEach(day -> {
        dailyActivityCount.clear();
        Task1.activities
            .stream()
            .filter(equalsDay(day))
            .forEach(activity -> {
                try {
                    dailyActivityCount.put(activity.getActivity_label(),
dailyActivityCount.get(activity.getActivity_label()) + 1);
                }catch (NullPointerException e) {
                    dailyActivityCount.put(activity.getActivity_label(), 1);
                }
            });
        HashMap<String, Integer> dailyActivityCopy = new HashMap<>(dailyActivityCount);
        dailyActivities.put(day, dailyActivityCopy);
```

```
                                });
```

Wen we print the results of Task4, we also print the month of the respective day, this is why we kept track of the month in Task2.

4.3.5 Task5

The 5th task requirements prompt us to create a Map using the activity name as the key and the total duration of an activity as the value. The requirement states that we should use LocalTime for counting the time, but LocalTime can only hold up to 23 hours and 59 minutes, and we have activities that have a total time well over 24 hours, so we used Integers for keeping track of the time in seconds.

The conversion of the duration of an activity to seconds is done in the following way:

```
Function<MonitoredData, Integer> toSeconds = data -> {
   int startSeconds = data.getStart_time().getHours()*3600 + data.getStart_time().getMinutes()*60 +
data.getStart_time().getSeconds();
   int endSeconds = data.getEnd_time().getHours()*3600 + data.getEnd_time().getMinutes()*60 +
data.getEnd_time().getSeconds();
   int finalTime = endSeconds - startSeconds;

   if (finalTime < 0) {
      finalTime += 24 * 3600;
   }

   return finalTime;
};
```

We take the end_time converted in seconds and subtract the start_time converted in seconds. If the result is less than 0, that means that the activity starts in a day and ends in the other, so we add 24 hours converted in seconds.
We go through every activity name and we create a HashMap called *activitiesDuration* which takes the name as the key and the duration in seconds as the value.
When we print the result in the “Task_5.txt” file, we convert the result into hours, minutes and seconds using the following Functions:

```
Function<Integer, Integer> getHours = data -> data / 3600;
Function<Integer, Integer> getMinutes = data -> (data%3600) / 60;
Function<Integer, Integer> getSeconds = data -> (data%3600) % 60;
```

4.3.6 Task6

The 6th task requires us to list all the activities that are shorter than 5 minutes 90% of the time. To achieve this we started by creating a HashMap with the name of the activity as the key and the number of occurrences of that activity for an interval of time shorter than 5 minutes:

```
  Task3.activityNames.forEach(activityName ->
     Task1.activities
           .stream()
           .filter(activity -> activity.getActivity_label().equals(activityName) &&
Task5.toSeconds.apply(activity) < 300)
           .forEach(activity -> {
              try {
                 shortActivitiesCount.put(activityName, shortActivitiesCount.get(activityName) + 1);
              }catch (NullPointerException e) {
                 shortActivitiesCount.put(activityName, 1);
              }
           }));
```

After that, we created a List of activities that meet the requirements of the task. To do this, we have taken the *activityCount* map from Task3 and filtered it such that the number present in the *shortActivitiesCount* is at least 90% of the number present in *activityCount* for the same activity_label:

```
private boolean lessThan5Minutes(HashMap<String, Integer> shortActivitiesCount, HashMap<String, Integer>
activitiesCount, String entry) {
            try {
                return shortActivitiesCount.get(entry) >= (int)(0.9 * activitiesCount.get(entry));
            }catch (NullPointerException e) {
                return false;
            }
    }

        List<String> shortActivities = Task3.activityCount.keySet()
                .stream()
                .filter(entry -> lessThan5Minutes(shortActivitiesCount, Task3.activityCount, entry))
                .collect(Collectors.toList());
```

5. Conclusions

During the development of this application we have learned how to make use of the Java 8 Functional Interfaces and lambda expressions to handle the non-OOP part of our code through declarative programming. As far as this application goes, with small changes to how we keep track of the days, it could be scaled to analyse up to a year of data, supposing we are starting from the 1st of January.

6. Bibliography

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

http://tutorials.jenkov.com/java-functional-programming/functional-interfaces.html

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

Lecture slides