Lambda Expressions and Stream Processing

~ Homework 5 ~

Papita Anda

Group 30421

Programming techniques

Table of Contents

1. Objective

The objective of this laboratory homework is 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), 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. this application is required to implement a series of tasks based on this log:

Hence, the purpose of this app is getting familiarized with lambda expressions and stream processing.

2. Problem analysis, modeling, scenarios, use-cases

In order to be able to work with the data provided by the log, one must understand the concept of lambda expression and streams.

Lambda expressions are introduced in Java 8 and are touted to be its biggest feature. Lambda expression facilitates functional programming, and simplifies the development a lot. A Java lambda expression is thus a function which can be created without belonging to any class. A Java lambda expression can be passed around as if it was an object and executed on demand.

A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements. Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result. For example, `filter`, `map` and `sorted` are intermediate operations whereas `forEach` is a terminal operation. A chain of stream operations is also known as *operation pipeline*. Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both *non-interfering* and *stateless*. Streams can be created from

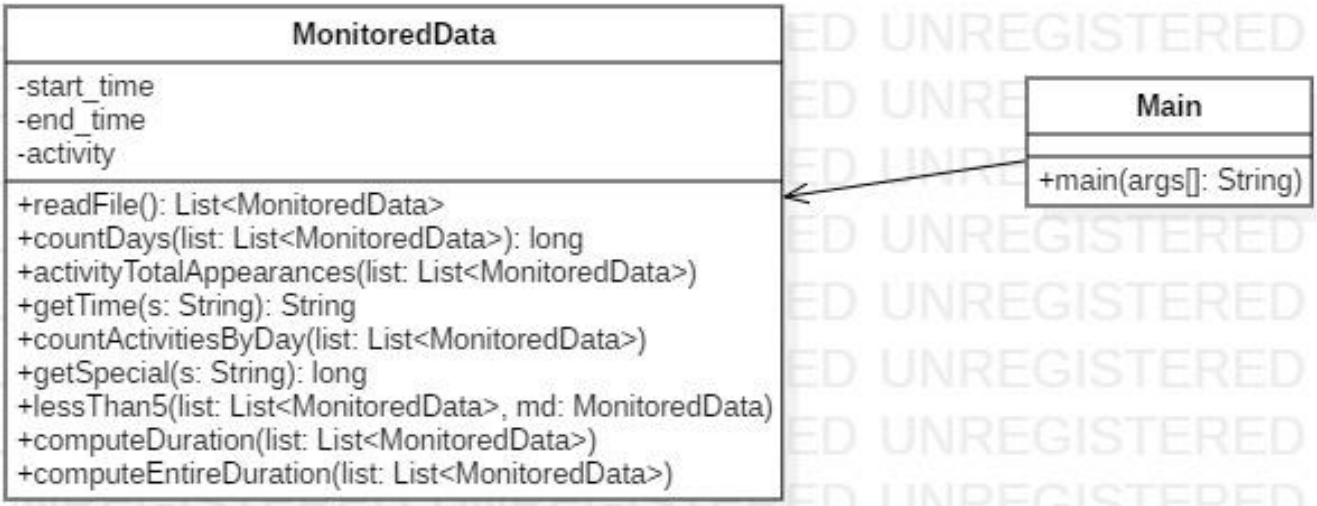
various data sources, especially collections. Lists and Sets support new methods `stream()` and `parallelStream()` to either create a sequential or a parallel stream. Parallel streams are capable of operating on multiple threads.

In this case, the application uses several lambda expressions for iterations, terminal and intermediate stream operations, and also pipelined operations. It mainly makes use of lists, which create a sequential stream.

3. Design

The design of this app follows a single-layer design architecture, consisting only of one package, main, and two classes: the model, MonitoredData, and the execution class Main.

The UML diagram of the project is the following one:



4. Implementation

MonitoredData class: this class represents the format of a line from the log Activities.txt. Its attributes consist in String start_time, representing the start of an activity, String end_time, the end of an activity and String activity, which holds the name of a specific activity. It contains simple methods as getters and setters for each field, a parameterized constructor, as well as a simple one, a toString() functions for printing the data and several methods which help implement the required tasks and are described below.

Methods:

- `public String getTime(String s)` the start_time and end_time hold a String of format `"yyyy-MM-dd HH:mm:ss"` and this method splits in half and returns the first half, the format for a date. This method is needed for another function for task 4, which counts activities by day, hence it doesn't need to know the time.
- `public List<MonitoredData> readFile()` this method implements the ***first task:*** it needs to 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. Firstly, it uses a stream to print the information from the file and then puts it in List, which it then returns.

```
public List<MonitoredData> readFile() { //task1
    String fileName = "D://Activities.txt";
    List<MonitoredData> list = new ArrayList<MonitoredData>();

    try (Stream<String> stream = Files.lines(Paths.get(fileName))) {
        stream.forEach(System.out::println);
    } catch (IOException e) {
        e.printStackTrace();
    }

    try (Stream<String> stream = Files.lines(Paths.get(fileName))) {
        list = stream
                .map(line -> line.split("\t\t"))
                .map(a -> new MonitoredData(a[0], a[1], a[2]))
                .collect(Collectors.toList());

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

second task: it needs to count the number of days that are recorded in the log. The stream uses a terminal operation which filters the data by the

number of sleep activities, then the method count() which returns the desires number of the days.

```
public long countDays(List<MonitoredData> list) { //task2
    long d = list.stream()
            .filter(s -> s.activity.equals("Sleeping"))
            .count();
    return d;
}
```

- `public Map<String, Long> ActivityTotalAppearances(List<MonitoredData> list)` this method, for ***task 3***, counts how many times each activity has appeared over the entire monitoring period. It returns a map of type representing the mapping of activities to their count. The method uses the operation collect for mapping the number of occurrences to each activity.

```
public void ActivityTotalAppearances(List<MonitoredData> list){ //task3
    Map<String, Long> counts = list.stream()
                                   .collect(Collectors.groupingBy(s->s.activity,
                                           Collectors.counting()));

    counts.entrySet().stream()
        .forEach(pair -> System.out.println(pair.getKey()
                + " - " + pair.getValue() + " times"));
}
```

- `public void countActivitiesByDay(List<MonitoredData> list)` this method, for ***task 4***, counts how many times each activity has appeared for each day over the monitoring period. Instead of creating a simple mapping, it creates one of this structure Map<T, Map<T, T>> , as for each day, each activity with the number of occurrences must be displayed. Like all other methods, it then prints the result.

```
public void countActivitiesByDay(List<MonitoredData> list) { //task4
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

    Map<LocalDate, Map<String, Long>> map = list.stream()
        .collect(Collectors.groupingBy(s->LocalDate.parse(s.getTime(s.start_time), formatter),
                Collectors.groupingBy(s->s.activity, Collectors.counting())));

    map.entrySet().stream()
    .forEach(pair -> System.out.println(pair.getKey() + " - " + pair.getValue()));

}
```

- `public Long getSpecial(Map<String, Long> counts, String s)` this method is a helper function for ***task 5***, which simply returns the label for an activity, if it exists, else it returns 0. It helps in computing the percentage.
- `public void computeDuration(List<MonitoredData> list)` this method for task 5 computes for each line from the file map for the activity label the duration recorded on that line (end_time-start_time). It uses the Duration class which has functions which help calculate the difference between LocalDateTime instances. It maps each line from the log with the duration of its activity, pretty straightforward.

```
public void computeDuration(List<MonitoredData> list) { //task5
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    Map <String, Duration>  duration = list.stream()
            .collect(Collectors.toMap(s->s.toString(),
                    s->Duration.between(LocalDateTime.parse(s.start_time, formatter),
                            LocalDateTime.parse(s.end_time, formatter)).abs()));
    duration.entrySet().stream()
        .forEach(pair -> System.out.println(pair.getKey() + " - " + pair.getValue()));
}
```

- `public void computeEntireDuration(List<MonitoredData> list)` this method, implemented for task 6, computes for each activity the entire duration over the monitoring period. It is quite similar to the previous method. The only difference is that by selecting to group the results by each specific line, it groups the result by each activity. Moreover, it has been implemented with different methods inside the collector operation to observe their functionality.

```
public void computeEntireDuration(List<MonitoredData> list) { //task6
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
    Map <String, Duration>  duration =
            list.stream()
            .collect(Collectors.groupingBy(s->s.activity,
                    Collectors.reducing(Duration.ZERO,
                            s -> Duration.between(LocalDateTime.parse(s.start_time, formatt
                                    LocalDateTime.parse(s.end_time, formatter)).abs(),
                                    Duration::plus
                                    )
                            ));
    duration.entrySet().stream()
        .forEach(pair -> System.out.println(pair.getKey() + " - " + pair.getValue()));

}
```

- `public void lessThan5(List<MonitoredData> list, MonitoredData md)` this method was implemented for ***task 7***. It filters the activities that have 90% of the monitoring records with duration less than 5 minutes, consisting of three steps. Firstly, it maps the total map, which calculates the total appearances for each activity. Secondly, for the counts map, it

returns the activities that take less than 5 minutes and their appearances in the log as well. Thirdly, in the last map, it filters the activities by percentage, which is calculated based on the two previous maps.

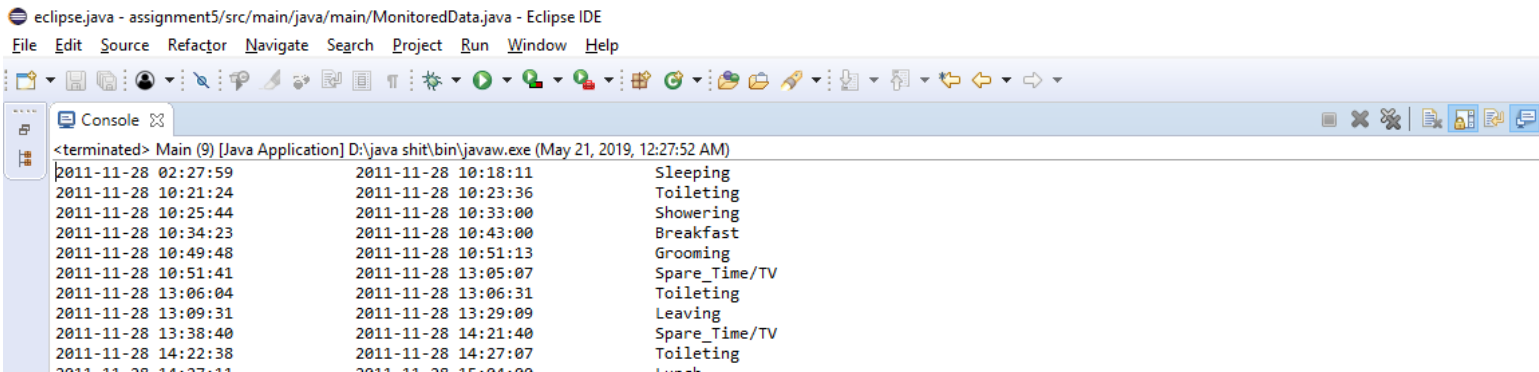
```
public void lessThan5(List<MonitoredData> list, MonitoredData md) {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

    Map<String, Long> total = list.stream()
        .collect(Collectors.groupingBy(s->s.activity, Collectors.counting()));

    Map<String, Long> counts = list.stream()
            .filter( s->Duration.between(LocalDateTime.parse(s.start_time, formatter),
                    LocalDateTime.parse(s.end_time, formatter)).abs().toMinutes() < 5)
            .collect(Collectors.groupingBy(s->s.activity,
            Collectors.counting()));

    Map<String, Long> newlist = list.stream()
            .filter(s ->  (md.getSpecial(counts, s.activity) * 100 / total.get(s.activity)  >
            .collect(Collectors.groupingBy(s->s.activity, Collectors.counting()));

    newlist.entrySet().stream().forEach(pair -> System.out.println(pair.getKey() + " - " + "l
}
```

Main class: this class executes the methods implemented in the previous class. It consists only in the main method.

5. Results

As we can see in the following picture, this is the result displayed by the console for each task.

Task 1 – reading the file, split its lines in 3 parts and creating a list with its objects.

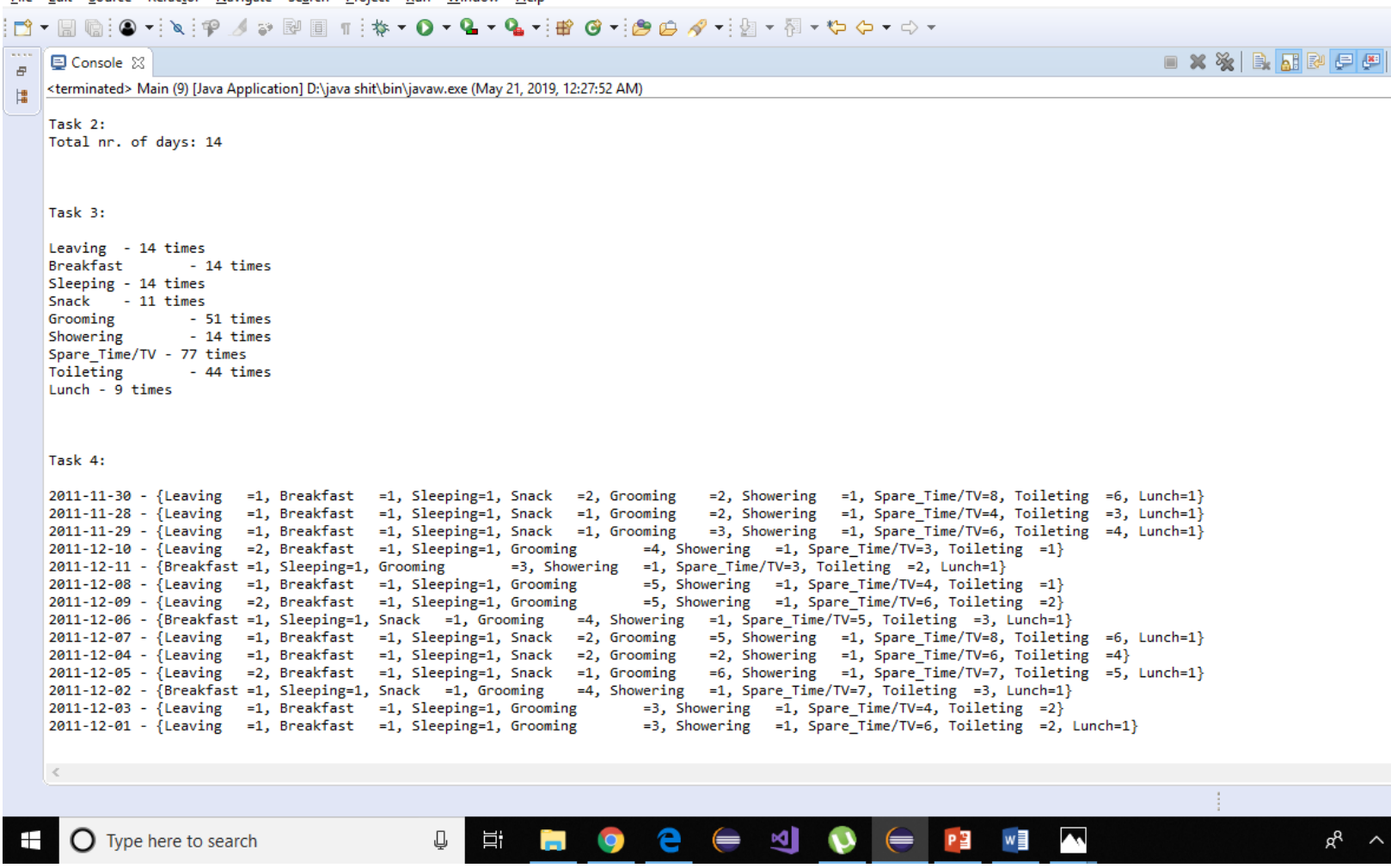


Task 2 – computing the number of days recorded

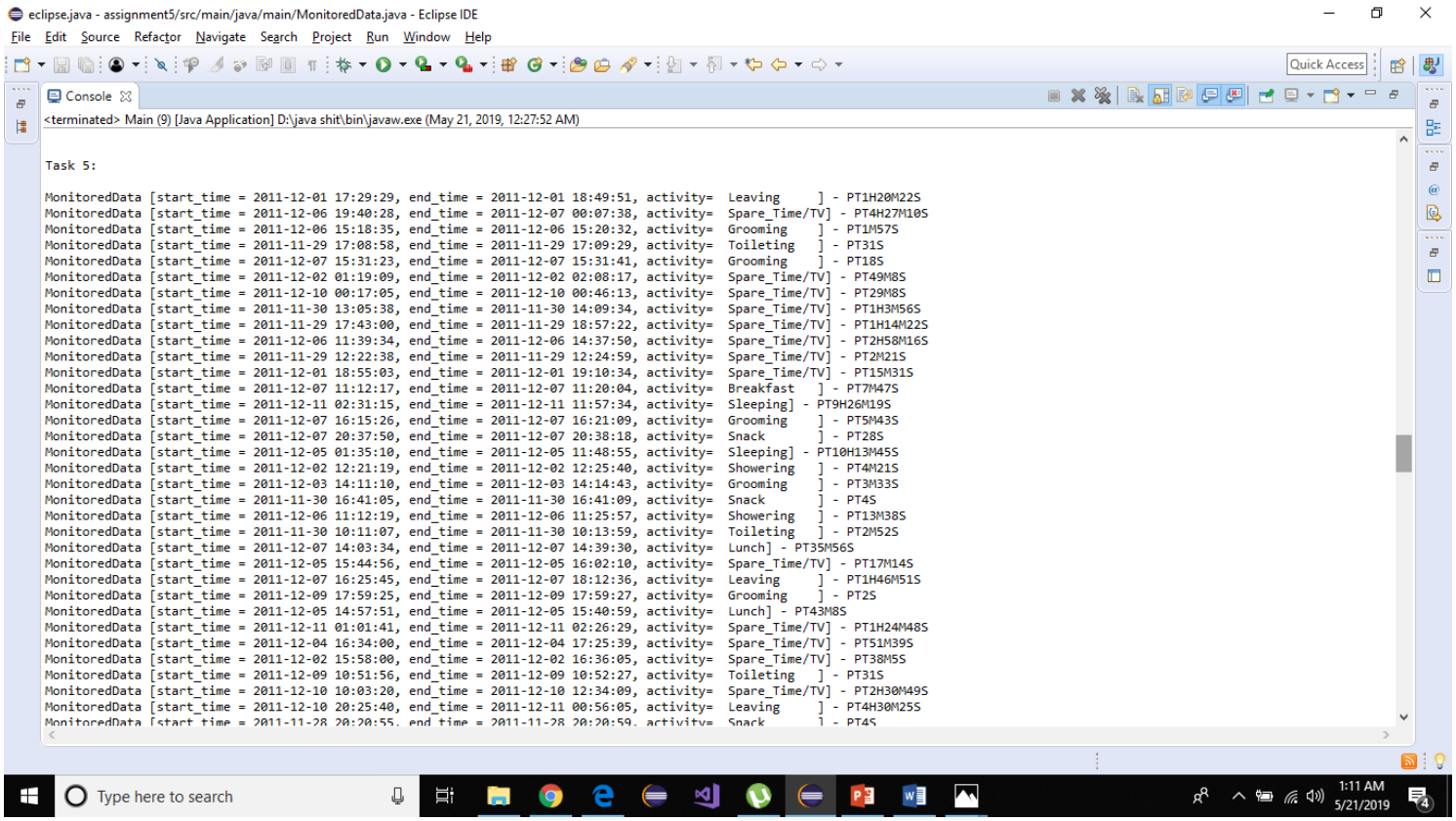
Task 3 - Counting how many times each activity has appeared over the entire monitoring period. Return a map of type representing the mapping of activities to their count.

Task 4 - Count how many times has appeared each activity for each day over the monitoring period

See results:



Task 5 - Compute for each line from the file map the duration recorded on that line (end_time-start_time)



Task 6 - For each activity compute the entire duration over the monitoring period

Task 7 - Filter the activities that have 90% of the monitoring records with duration less than 5 minutes

```
Task 6:

Leaving  - PT27H44M44S
Breakfast        - PT2H58M8S
Sleeping - PT131H3M31S
Snack    - PT6M1S
Grooming         - PT2H40M42S
Showering        - PT1H34M9S
Spare_Time/TV - PT142H28M55S
Toileting        - PT2H20M34S
Lunch - PT5H13M31S


Task 7:

Snack    - lasts 11 times (90% of the monitoring records) less than 5 mins
```

6. Conclusions

In conclusion, this assignment was highly beneficial for documenting and getting a good understanding of lambda expressions and streams. It has shown that many expressions and calculus can be simplified with a correct use of these features.

7. Bibliography

1. http://tutorials.jenkov.com/java/lambda-expressions.html
2. https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/
3. https://archive.ics.uci.edu/ml/datasets/Activities+of+Daily+Living+(ADLs)+Recognition+Using+Binary+Sensors