Technical University of Cluj-Napoca

Year 2016-2017

Department of Computer Science

Programming Techniques

5th Assignment

Student: Gyarmathy Tímea

Group: 30423

Table of Contents

[Specification of the problem 3](#_Toc482948898)

[Description 3](#_Toc482948899)

[The analysis of the problem and modeling 4](#_Toc482948900)

[Reading the input file 4](#_Toc482948901)

[Class Control 4](#_Toc482948902)

[Implementation and testing 5](#_Toc482948903)

[Conclusions 5](#_Toc482948904)

[What have I learned? 5](#_Toc482948905)

[Bibliography 5](#_Toc482948906)

# Specification of the problem

A smart house features a set of sensors that may be used to record the behavior of a person living in the house. The historical log of the person’s activity is stored as tuples (*startTime, endTime, activityLabel),* where *startTime* and *endTime* 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 attached log file *Activities.txt* contains a set of activity records over a certain period of time.

Define a class *MonitoredData* having *startTime, endTime* and *activityLabel* as instance variables and read the input file data into the data structure *monitoredData* of type *List<MonitoredData>*. Using stream processing techniques and lambda expressions introduced by Java 8, write the following set of short programs for processing the *monitoredData*.

1. Count the distinct days that appear in the monitoring data.

2. Determine a map of type <String, Integer> that maps to each distinct action type the number of occurrences in the log. Write the resulting map into a text file.

3. Generates a data structure of type Map<Integer, Map<String, Integer>> that contains the activity count for each day of the log (task number 2 applied for each day of the log) and writes the result in a text file.

4. Determine a data structure of the form Map<String, DateTime> that maps for each activity the total duration computed over the monitoring period. Filter the activities with total duration larger than 10 hours. Write the result in a text file.

5. Filter the activities that have 90% of the monitoring samples with duration less than 5 minutes, collect the results in a List<String> containing only the distinct activity names and write the result in a text file.

# Description

In [computer programming](https://en.wikipedia.org/wiki/Computer_programming), an anonymous function (function literal, lambda abstraction) is a [function](https://en.wikipedia.org/wiki/Function_(computer_science)) definition that is not [bound](https://en.wikipedia.org/wiki/Name_binding) to an [identifier](https://en.wikipedia.org/wiki/Name_(computer_science)). Anonymous functions are often:

1. arguments being passed to [higher-order functions](https://en.wikipedia.org/wiki/Higher-order_function), or
2. used for constructing the result of a higher-order function that needs to return a function.

If the function is only used once, or a limited number of times, an anonymous function may be syntactically lighter than using a named function. Anonymous functions are ubiquitous in [functional programming languages](https://en.wikipedia.org/wiki/Functional_programming_language) and other languages with [first-class functions](https://en.wikipedia.org/wiki/First-class_function), where they fulfill the same role for the [function type](https://en.wikipedia.org/wiki/Function_type) as [literals](https://en.wikipedia.org/wiki/Literal_(computer_programming)) do for other [data types](https://en.wikipedia.org/wiki/Data_type).

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream −

* Sequence of elements − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
* Source − Stream takes Collections, Arrays, or I/O resources as input source.
* Aggregate operations − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
* Pipelining − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
* Automatic iterations − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

# The analysis of the problem and modeling

To properly model the smart house, as the specification asks, we'll create the class MonitoredData, to which we'll add the 3 specified attributes. For startTime and endTime we'll use LocalDateTime, as this class has a wide variety of useful methods. The attribute activityLabel will be stored as a String. This class contains getters and setters to these properties and besides, two helper methods, one rrturns only the date of the monitored activity, while the other the duration in seconds of monitorization.

The whole application consists of three classes, one of which was described above, and the other two are: Main and Control. The Main class only contains the main() method which runs the application, but within this, the reading and parsing of the source text file is done.

## Reading the input file

It is done with the help of streams, and data is directly converted into MonitoredData objects:

list = stream .map(line -> {

String[] lines = line.split(" ");

DateTimeFormatter format = DateTimeFormatter.*ofPattern*("yyyy-MM-dd HH:mm:ss");

**return** **new** MonitoredData(LocalDateTime.*parse*(lines[0], format),

LocalDateTime.*parse*(lines[1], format), lines[2]);

}).collect(Collectors.*toList*());

Within the lambda expression, strings are parsed and converted into objects. At the end, the stream is collected into a list.

## Class Control

It contains all the implementations of the methods which solve the requirements of the assignment.

It has an attribute, a list of MonitoredData objects, which is passed to it when created. The methods are implemented in the shortest way possible, but each of them writes the result in their specificly named text file, from within the method. Thus, most of these methods do not return a value, but directly write into a text file, as the specification says. In some cases, the result is directly computed when writing it to the file. To take an example of implementation, Let’s look at request number 4.

To be able to work with time, I took the units in seconds, so whenever we see an unit of time of type long, the value is converted in seconds. Here, in this method, the following happens:

1. We convert the requested filter (number of hours) into seconds.
2. We take a map, which counts for each activity the total duration, by mapping them to their specific activity label, grouping by it, and the summing the value returned of the method getDuration from class MonitoredData, which returns the duration in seconds.
3. A path is specified for the output text file.
4. The resulted map is written into the output file, but the filter is applied, thus, data only with duration greater than h hours is passed. We can see this condition in the following sequence: ().filter(t -> t.getValue() > hours)

The code of the method looks like the following:

**public** **void** activityDurations(**int** h) {

**long** hours = h \* 60 \* 60;

Map<String, Long> myMap = list.stream().collect(Collectors.*groupingBy*(MonitoredData::getActivityLabel,

Collectors.*summingLong*(MonitoredData::getDuration)));

Path myPath = Paths.*get*("activityDurations.txt");

**try** {

Files.*write*(myPath, () -> myMap.entrySet().stream().filter(t -> t.getValue() > hours).<CharSequence>map(e -> e.getKey() + ": " + e.getValue()).iterator());

} **catch** (IOException e) {

e.printStackTrace();

}

}

# Implementation and testing

Implementation started from modeling the monitored data class, then reading the input file and creating a list of objects based on it. All the functions from the requirement were implemented in order of appearance. Testing was done by printing the expected data and verifying its correctness. Throughout development when an error occurred, I resolved it using the IDE’s debugger and tracing the error source.

# Conclusions

## What have I learned?

I have learned the use of lambda expressions and streams.

# Bibliography

1. For programming issues: <https://docs.oracle.com/javase/8/docs/api/overview-summary.html>
2. For more programming issues: <http://stackoverflow.com/>
3. Lambda expressions: <https://en.wikipedia.org/wiki/Anonymous_function>
4. Java streams: <https://www.tutorialspoint.com/java8/java8_streams.htm>
5. Last semester’s OOP code on how to use Swing to create user interface, and to realize the MVC.