Technical University of Cluj-Napoca

Faculty of Automation and Computer Science

2nd Semester 2016-2017

Stream processing

Programming Techniques

Homework 5

Student Ieremiasș Viorel

Group 30422

Contents

[1. Problem specification 2](#_Toc483879840)

[1.1. Task description 2](#_Toc483879841)

# Problem specification

## Task description

**Description**

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.

# Objectives

## Main objective

Develop an application that processes a set of data representing the records of a person’s activities, as recorded by the sensors in the smart house, that monitors the behavior of the person.

The processing of the data means applying some operations on the raw data, in order to extract meaningful information and statistics, that could be used for a wide range of purpose, from making the house more energetically efficient, to monitoring the activities that take the most amount of time in daily life and where time could be spared and used elsewhere.

The operations mentioned above include filtering the activities based on some useful criteria, counting the occurrences of some events or the number of different activities in one day, summing the total time of an activity and some others.

# Theoretical foundations

## Streams

In this example, widgets are a Collection<Widget>. We create a stream of Widget objects via Collection.stream(), filter it to produce a stream containing only the red widgets, and then transform it into a stream of int values representing the weight of each red widget. Then this stream is summed to produce a total weight.

In addition to Stream, which is a stream of object references, there are primitive specializations for IntStream, LongStream, and DoubleStream, all of which are referred to as "streams" and conform to the characteristics and restrictions described here.

To perform a computation, stream operations are composed into a stream pipeline. A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc), zero or more intermediate operations (which transform a stream into another stream, such as filter(Predicate)), and a terminal operation (which produces a result or side-effect, such as count() or forEach(Consumer)). Streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

Collections and streams, while bearing some superficial similarities, have different goals. Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source. However, if the provided stream operations do not offer the desired functionality, the BaseStream.iterator() and BaseStream.spliterator() operations can be used to perform a controlled traversal.

A stream pipeline, like the "widgets" example above, can be viewed as a query on the stream source. Unless the source was explicitly designed for concurrent modification (such as a ConcurrentHashMap), unpredictable or erroneous behavior may result from modifying the stream source while it is being queried.

Most stream operations accept parameters that describe user-specified behavior, such as the lambda expression w -> w.getWeight() passed to mapToInt in the example above. To preserve correct behavior, these behavioral parameters:

* must be non-interfering (they do not modify the stream source); and
* in most cases must be stateless (their result should not depend on any state that might change during execution of the stream pipeline).

Such parameters are always instances of a functional interface such as Function, and are often lambda expressions or method references. Unless otherwise specified these parameters must be non-null.

A stream should be operated on (invoking an intermediate or terminal stream operation) only once. This rules out, for example, "forked" streams, where the same source feeds two or more pipelines, or multiple traversals of the same stream. A stream implementation may throw IllegalStateException if it detects that the stream is being reused. However, since some stream operations may return their receiver rather than a new stream object, it may not be possible to detect reuse in all cases.

Streams have a BaseStream.close() method and implement AutoCloseable, but nearly all stream instances do not actually need to be closed after use. Generally, only streams whose source is an IO channel (such as those returned by Files.lines(Path, Charset)) will require closing. Most streams are backed by collections, arrays, or generating functions, which require no special resource management. (If a stream does require closing, it can be declared as a resource in a try-with-resources statement.)

Stream pipelines may execute either sequentially or in parallel. This execution mode is a property of the stream. Streams are created with an initial choice of sequential or parallel execution. (For example, Collection.stream() creates a sequential stream, and Collection.parallelStream() creates a parallel one.) This choice of execution mode may be modified by the BaseStream.sequential() or BaseStream.parallel() methods, and may be queried with the BaseStream.isParallel() method.