**Assignment 5**

**Programming Techniques**

Stream Processing Application

**Student:**Boian Maria Andreea

**Group**:30423

Teacher: Ioan Salomie

Lab assistant: Claudia Pop

2 nd semester 2017-2018

1.**Assignment objective**

**1. 1. Task description**

Consider the task of 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\_label), 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, taken from [1,2] and downloadable from the file Activities.txt located in this folder. Write a Java 1.8 program using lambda expressions and stream processing to do the tasks defined below.

* Define a class MonitoredData with 3 fields: start time, end time and activity as string. 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
* Count how many days of monitored data appears in the log
* Count how many times has appeared each activity over the entire monitoring period. Return a map of type representing the mapping of activities to their count.
* Count how many times has appeared each activity for each day over the monitoring period
* For each line from the file map for the activity label the duration recorded on that line (end\_time-start\_time)
* For each activity compute the entire duration over the monitoring period
* Filter the activities that have 90% of the monitoring records with duration less than 5 minutes
* Documentation

**1.2. Application description**

This application allows the user to see a brief example of working with streams, lambda expressions and method references in Java 8. These features are applied on a set of data saved in Activities.txt . Results from the data filtering are stored in text files NumberOfDays,ActivityOccurences,OccurenceOfEachActivityEachDay,EveryDayActivityDuration and OverAllActivityDuration.

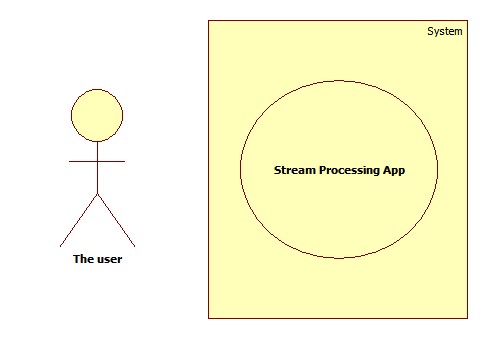
**2.Problem analysis and assumptions**

**2.1. General overview**

This application should be able to show a brief example of working with advanced Java features such as lambda expressions, method references and streams.The data over which these are performed is stored in a text file called Activities.txt.

**2.2. Assumptions**

The user is expected to have a basic knowledge of streams and lambda expressions in Java 8 in order to undersand the operations performed over the data in this application.



**3.Use cases:Simulate**

- Summary : This use case allows the user to simulate and observe the results the data stored in Activities.txt text file over the type of activity performed by the person: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare\_Time/TV, Grooming . .

- Actors: The User

- Preconditions: None

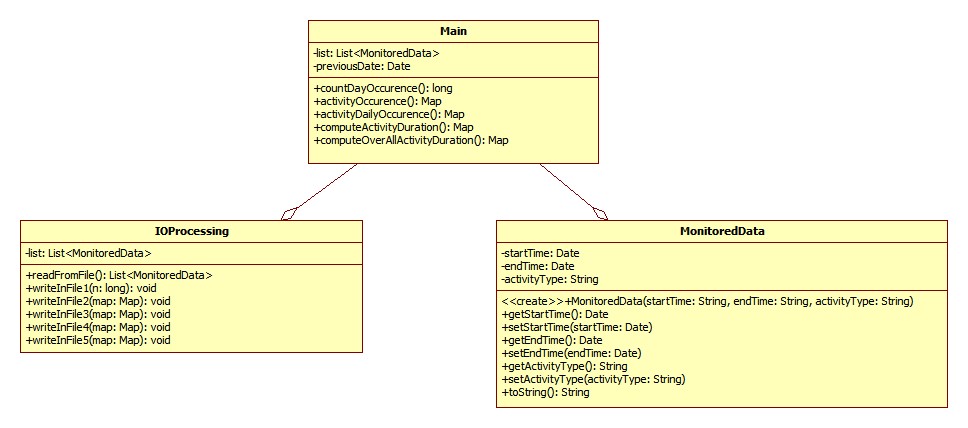
- Main success scenario : 1 . The user pushes the run button after opening the project , in the Main class . 2 . The user checks the text files located in the project directory . 3 . The users observes the resulted records . 4 . Simulation ends .

- Alternative sequences: None

- Error sequences: a) Invalid input: appears at step 2 1. The program returns an error; the use case fails .

**4.Design**

Below the uml diagram is attached . Each class will be disscused in detail .



**4.1 Class design**

**4.1.1** MonitoredData

This class is used to hold the attributes of a row of data from Activities.txt .

**4.1.2** IOProcessing

This class is used in order to read data from Activities.txt and also to write the results of the operations performed over it in separate text files .

**4.1.3** Main

Main class holds the main logic of the process which are operations performed over the activity log .

5.Implementation

**5.1** MonitoredData

This class holds represents is used to represent a row of data from the activity log, therefor it's instance variables are startTime, end time and activityType . Getters and setters are also generated besides the class's constructor.

**5.2** IOProcessing

In IOProcessing a list of type MonitoredData is used in order to save data red from text file.It consists of 6 methods:

* readFromFile() : read the data from the textFile using streams and stores it in the list.
* writeInFile1( long l) : writes in a text file the number of days from text file
* writeInFile2 ( Map < String , Long > map) : writes in file the overall activities occurences over the monitored period
* writeInFile3 ( Map < Date , Map < String,Long> > map) : writes in a text file how many times each activity takes place on every day over the monitored period
* writeInFile4 ( Map < MonitoredData , Long > map ) : writes in a text file the duration of each activity from every row of data in seconds
* writeInFile5 ( Map <String ,Long > map ) : writes in a text file the overall activity duration over the entire monitored period in seconds

**4.1.3** IOProcessing

**5.3** Main

This class represents the core of the entire project as it holds the methods for performing the desired operations over the activities stored in the input text file. It consists of the following methods:

* countDayOccurence()
* activityOccurence()
* activityDailyOccurence()
* computeActivityDuration()
* computeOverallActivityDuration()

**6.Conclusions**

**6.1 What I learned**

Trying to learn how to work with these new Java 8 features was not easy at the begining, easpecially understanding the streams concept and how lambda works , but it is from far a very "clean" and efficent way to write code . In my future projects I am looking forward for using these features in order to get more familiar with it .

BIBLIOGRAPHY

* https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html http://www.mkyong.com/tutorials/java-8-tutorials/
* https://www.codementor.io/eh3rrera/using-java-8-method-reference-du10866vx
* https://stackoverflow.com/questions/31044041/how-do-i-iterate-over-a-stream-in-java-using-for
* https://howtodoinjava.com/java-8/how-to-use-predicate-in-java-8/

Following there is a small section about Java 8 streams and lambda expressions.

**Stream processing** is a [computer programming](https://en.wikipedia.org/wiki/Computer_programming) paradigm, equivalent to [dataflow programming](https://en.wikipedia.org/wiki/Dataflow_programming), [event stream processing](https://en.wikipedia.org/wiki/Event_stream_processing), and [reactive programming](https://en.wikipedia.org/wiki/Reactive_programming),[[1]](https://en.wikipedia.org/wiki/Stream_processing#cite_note-1) that allows some applications to more easily exploit a limited form of [parallel processing](https://en.wikipedia.org/wiki/Parallel_computing). Such applications can use multiple computational units, such as the [floating point unit](https://en.wikipedia.org/wiki/Floating_point_unit) on a [graphics processing unit](https://en.wikipedia.org/wiki/Graphics_processing_unit) or [field-programmable gate arrays](https://en.wikipedia.org/wiki/Field-programmable_gate_array) (FPGAs),[[2]](https://en.wikipedia.org/wiki/Stream_processing#cite_note-2) without explicitly managing allocation, synchronization, or communication among those units.

The stream processing paradigm simplifies parallel software and hardware by restricting the parallel computation that can be performed. Given a sequence of data (a *stream*), a series of operations ([*kernel functions*](https://en.wikipedia.org/wiki/Compute_kernel)) is applied to each element in the stream. Kernel functions are usually [pipelined](https://en.wikipedia.org/wiki/Pipeline_(computing)), and optimal local on-chip memory reuse is attempted, in order to minimize the loss in bandwidth, accredited to external memory interaction. *Uniform streaming*, where one kernel function is applied to all elements in the stream, is typical. Since the kernel and stream abstractions expose data dependencies, compiler tools can fully automate and optimize on-chip management tasks. Stream processing hardware can use [scoreboarding](https://en.wikipedia.org/wiki/Scoreboarding" \o "Scoreboarding), for example, to initiate a [direct memory access](https://en.wikipedia.org/wiki/Direct_memory_access) (DMA) when dependencies become known. The elimination of manual DMA management reduces software complexity, and an associated elimination for hardware cached I/O, reduces the data area expanse that has to be involved with service by specialized computational units such as [arithmetic logic units](https://en.wikipedia.org/wiki/Arithmetic_logic_unit).

During the 1980s stream processing was explored within [dataflow programming](https://en.wikipedia.org/wiki/Dataflow_programming). An example is the language [SISAL](https://en.wikipedia.org/wiki/SISAL) (Streams and Iteration in a Single Assignment Language).

## Applications[[edit](https://en.wikipedia.org/w/index.php?title=Stream_processing&action=edit&section=1)]

Stream processing is essentially a compromise, driven by a data-centric model that works very well for traditional DSP or GPU-type applications (such as image, video and [digital signal processing](https://en.wikipedia.org/wiki/Digital_signal_processing)) but less so for general purpose processing with more randomized data access (such as databases). By sacrificing some flexibility in the model, the implications allow easier, faster and more efficient execution. Depending on the context, [processor](https://en.wikipedia.org/wiki/Central_processing_unit) design may be tuned for maximum efficiency or a trade-off for flexibility.

Stream processing is especially suitable for applications that exhibit three application characteristics:[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

* **Compute Intensity**, the number of arithmetic operations per I/O or global memory reference. In many signal processing applications today it is well over 50:1 and increasing with algorithmic complexity.
* **Data Parallelism** exists in a kernel if the same function is applied to all records of an input stream and a number of records can be processed simultaneously without waiting for results from previous records.
* **Data Locality** is a specific type of temporal locality common in signal and media processing applications where data is produced once, read once or twice later in the application, and never read again. Intermediate streams passed between kernels as well as intermediate data within kernel functions can capture this locality directly using the stream processing programming model.

Examples of records within streams include:

* In graphics, each record might be the vertex, normal, and color information for a triangle;
* In image processing, each record might be a single pixel from an image;
* In a video encoder, each record may be 256 pixels forming a macroblock of data; or
* In wireless signal processing, each record could be a sequence of samples received from an antenna.

For each record we can only read from the input, perform operations on it, and write to the output. It is permissible to have multiple inputs and multiple outputs, but never a piece of memory that is both readable and writable.

## Comparison to prior parallel paradigms[[edit](https://en.wikipedia.org/w/index.php?title=Stream_processing&action=edit&section=2)]

Basic computers started from a sequential execution paradigm. Traditional [CPUs](https://en.wikipedia.org/wiki/Central_processing_unit) are [SISD](https://en.wikipedia.org/wiki/SISD) based, which means they conceptually perform only one operation at a time. As the computing needs of the world evolved, the amount of data to be managed increased very quickly. It was obvious that the sequential programming model could not cope with the increased need for processing power. Various efforts have been spent on finding alternative ways to perform massive amounts of computations but the only solution was to exploit some level of parallel execution. The result of those efforts was [SIMD](https://en.wikipedia.org/wiki/SIMD), a programming paradigm which allowed applying one instruction to multiple instances of (different) data. Most of the time, SIMD was being used in a [SWAR](https://en.wikipedia.org/wiki/SWAR) environment. By using more complicated structures, one could also have [MIMD](https://en.wikipedia.org/wiki/MIMD) parallelism.

Although those two paradigms were efficient, real-world implementations were plagued with limitations from memory alignment problems to synchronization issues and limited parallelism. Only few SIMD processors survived as stand-alone components; most were embedded in standard CPUs.

Consider a simple program adding up two arrays containing 100 4-component [vectors](https://en.wikipedia.org/wiki/Vector_(geometric)) (i.e. 400 numbers in total).

## Stream programming libraries and languages[[edit](https://en.wikipedia.org/w/index.php?title=Stream_processing&action=edit&section=12)]

Most programming languages for stream processors start with Java, C or C++ and add extensions which provide specific instructions to allow application developers to tag kernels and/or streams. This also applies to most [shading languages](https://en.wikipedia.org/wiki/Shading_language), which can be considered stream programming languages to a certain degree.

Non-commercial examples of stream programming languages include:

* [Ateji PX](https://en.wikipedia.org/wiki/Ateji_PX) [Free Edition](http://www.ateji.com/px/index.html), enables a simple expression of stream programming, the Actor model, and the MapReduce algorithm on [JVM](https://en.wikipedia.org/wiki/JVM)
* [Auto-Pipe](http://sbs.wustl.edu/), from the Stream Based Supercomputing Lab at [Washington University in St. Louis](https://en.wikipedia.org/wiki/Washington_University_in_St._Louis), an application development environment for streaming applications that allows authoring of applications for heterogeneous systems (CPU, [GPGPU](https://en.wikipedia.org/wiki/GPGPU), FPGA). Applications can be developed in any combination of C, C++, and Java for the CPU. Verilog or VHDL for FPGAs. Cuda is currently used for Nvidia GPGPUs. Auto-Pipe also handles coordination of TCP connections between multiple machines.
* [ACOTES](http://www.hitech-projects.com/euprojects/ACOTES/) Programming Model: language from [Polytechnic University of Catalonia](https://en.wikipedia.org/wiki/Polytechnic_University_of_Catalonia) based on [OpenMP](https://en.wikipedia.org/wiki/OpenMP" \o "OpenMP)
* [Brook](http://graphics.stanford.edu/projects/brookgpu/lang.html) language from [Stanford](https://en.wikipedia.org/wiki/Stanford)
* [CAL](https://en.wikipedia.org/wiki/CAL_Actor_Language) Actor Language: a high-level programming language for writing (dataflow) actors, which are stateful operators that transform input streams of data objects (tokens) into output streams.
* [Cal2Many](https://github.com/gEssayas/Cal2Many/) a code generation framework from Halmstad University, Sweden. It takes CAL code as input and generates different target specific languages including sequential C, Chisel, parallel C targeting Epiphany architecture, ajava & astruct targeting Ambric architecture, etc..
* [DUP](http://dupsystem.org/) language from [Technical University of Munich](https://en.wikipedia.org/wiki/Technical_University_of_Munich) and [University of Denver](https://en.wikipedia.org/wiki/University_of_Denver)
* [RaftLib](http://raftlib.io/) - open source C++ stream processing template library originally from the Stream Based Supercomputing Lab at [Washington University in St. Louis](https://en.wikipedia.org/wiki/Washington_University_in_St._Louis)
* [Sh](https://en.wikipedia.org/wiki/Lib_Sh) library from the [University of Waterloo](https://en.wikipedia.org/wiki/University_of_Waterloo)
* [Shallows](http://shallows.sourceforge.net/), an open source project
* [S-Net](http://www.snet-home.org/) coordination language from the [University of Hertfordshire](https://en.wikipedia.org/wiki/University_of_Hertfordshire), which provides separation of coordination and algorithmic programming
* [StreamIt](http://www.cag.csail.mit.edu/streamit/) from [MIT](https://en.wikipedia.org/wiki/MIT)
* [Siddhi](https://wso2.github.io/siddhi/) from [WSO2](https://en.wikipedia.org/wiki/WSO2)
* [WaveScript](http://www.cs.indiana.edu/~rrnewton/wavescope/WaveScope_+_WaveScript/WaveScope_Homepage.html) Functional stream processing, also from MIT.
* [Functional reactive programming](https://en.wikipedia.org/wiki/Functional_reactive_programming) could be considered stream processing in a broad sense.

Commercial implementations are either general purpose or tied to specific hardware by a vendor. Examples of general purpose languages include:

* [AccelerEyes](https://en.wikipedia.org/wiki/AccelerEyes)' Jacket, a commercialization of a GPU engine for MATLAB
* [Ateji PX](https://en.wikipedia.org/wiki/Ateji_PX) Java extension that enables a simple expression of stream programming, the Actor model, and the MapReduce algorithm
* [Floodgate](http://www.emergent.net/Products/Gamebryo/Technical-Details/Floodgate/), a stream processor provided with the [Gamebryo](https://en.wikipedia.org/wiki/Gamebryo" \o "Gamebryo) game engine for PlayStation 3, Xbox360, Wii, and PC
* [OpenHMPP](https://en.wikipedia.org/wiki/OpenHMPP), a "directive" vision of Many-Core programming
* PeakStream,[[8]](https://en.wikipedia.org/wiki/Stream_processing#cite_note-8) a spinout of the [Brook](https://en.wikipedia.org/wiki/BrookGPU) project (acquired by [Google](https://en.wikipedia.org/wiki/List_of_Google_acquisitions) in June 2007)
* [IBM Spade - Stream Processing Application Declarative Engine (B. Gedik, et al. SPADE: the system S declarative stream processing engine. ACM SIGMOD 2008.)](http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.spade.html)
* [RapidMind](https://en.wikipedia.org/wiki/RapidMind), a commercialization of [Sh](https://en.wikipedia.org/wiki/Lib_Sh" \o "Lib Sh) (acquired by [Intel](https://en.wikipedia.org/wiki/Intel) in August 2009)
* TStreams,[[9]](https://en.wikipedia.org/wiki/Stream_processing#cite_note-9)[[10]](https://en.wikipedia.org/wiki/Stream_processing#cite_note-10) Hewlett-Packard Cambridge Research Lab

Vendor-specific languages include:

* Brook+ (AMD hardware optimized implementation of [Brook](https://en.wikipedia.org/wiki/BrookGPU)) from [AMD](https://en.wikipedia.org/wiki/AMD)/[ATI](https://en.wikipedia.org/wiki/ATI_Technologies)
* [CUDA](https://en.wikipedia.org/wiki/CUDA) (Compute Unified Device Architecture) from [Nvidia](https://en.wikipedia.org/wiki/Nvidia" \o "Nvidia)
* [Intel Ct](https://en.wikipedia.org/wiki/Intel_Ct) - C for [Throughput Computing](https://en.wikipedia.org/wiki/High-throughput_computing)
* StreamC from [Stream Processors, Inc](https://en.wikipedia.org/wiki/Stream_Processors,_Inc), a commercialization of the Imagine work at Stanford

Event-Based Processing

* [Apama](https://en.wikipedia.org/wiki/Apama_(software)) - a combined [Complex Event](https://en.wikipedia.org/wiki/Complex_event_processing) and [Stream Processing](https://en.wikipedia.org/wiki/Stream_Processing) Engine by [Software AG](https://en.wikipedia.org/wiki/Software_AG)
* [Wallaroo](https://www.wallaroolabs.com/community)
* [WSO2 Stream Processor](https://wso2.com/analytics)

Batch File Based Processing (emulates some of actual stream processing, but much lower performance in general)

* [Apache Kafka](https://en.wikipedia.org/wiki/Apache_Kafka)
* [Apache Flink](https://en.wikipedia.org/wiki/Apache_Flink)
* [Apache Storm](https://en.wikipedia.org/wiki/Apache_Storm)
* [Apache Apex](https://en.wikipedia.org/wiki/Apache_Apex)
* [Apache Spark](https://en.wikipedia.org/wiki/Apache_Spark)

Stream Processing Services:

* [Amazon Web Services - Kinesis](https://aws.amazon.com/kinesis/)
* [Google Cloud - Dataflow](https://cloud.google.com/dataflow/)
* [Microsoft Azure - Stream Analytics](https://azure.microsoft.com/en-us/services/stream-analytics/)

source: https://en.wikipedia.org/wiki/Stream\_processing