**DOCUMENTATIE TEMA 5**

**Stream Processing using Lambda Expressions**

**Nume prenume Man Larisa Silvana**

**Grupa 30224**

**Profesor Laborator Assist Pop Claudia**

Contents

[1. Cerinte Functionale 3](#_Toc483437676)

[2. Obiective 4](#_Toc483437677)

[2.1. Obiectiv Principal: 4](#_Toc483437678)

[2.2. Obective Secundare: 4](#_Toc483437679)

[3. Analiza Problemei 4](#_Toc483437680)

[4. Implementare 5](#_Toc483437681)

[5. Concluzii si Dezvoltari Ulterioare 7](#_Toc483437682)

[6. Bibliografie 7](#_Toc483437683)

# Cerinte Functionale

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.

# Obiective

## Obiectiv Principal:

Principalul obiectiv este invatarea folosirii stream-urilor si lambda expressions, pe care le gasim in Java 8.

## Obective Secundare:

Am avut o serie de cerinte care trebuie implemetate intr-un mod cat mai simplu folosind pachetul Java 8.

# Analiza Problemei

O funcţie lambda (funcţie anonimă) este o funcţie definită şi apelată fără a fi legată de un identificator. Funcţiile lambda sunt o formă de funcţii ,,incuibate” (nested functions) în sensul că permit accesul la variabilele din domeniul funcţiei în care sunt conţinute.

Funcţiile anonime au fost introduse de către Alonzo Church în anul 1936, în teoria sa despre calculele lambda.

În limbajele de programare, funcţiile anonime sunt implementate din anul 1958 ca parte a limbajului Lisp. În unele limbajele orientate pe obiect, precum Java, apar concepte similare, precum clasele anonime. Abia în versiunea 8 a limbajului Java sunt adăugate şi funcţiile anonime. Alte limbaje, precum C#, JavaScript, Perl, Python, Ruby ofereau demult suport pentru acest concept.

Lambda expresiile ne permit să creăm instanţe ale claselor cu o singură metodă într-un mod mult mai compact.

O lambda expresie constă:

dintr-o listă de parametri formali, separaţi prin virgulă şi cuprinşi eventual între paranteze rotunde,

săgeata direcţională ->,

un body ce constă dintr-o expresie sau un bloc de instrucţiuni.

O interfaţă funcţională (functional interface) este orice interfaţă ce conţine doar o metodă abstractă. Din această cauză putem omite numele metodei atunci când implementăm interfaţa şi putem elimina folosirea claselor anonime. În locul lor vom avea lambda expresii. O interfaţă funcţională este anotată cu @FunctionalInterface.

# Implementare

Proiectul meu consta din doua clase :MonitoredData si clasa Main.

Clasa MonitoredData este reprezentata de o variabila de tip string – activityLabel si doua de tip DateTime, din pachetul JODA, - startTime si endTime. Mai gasim o lista monitoredData, de tipul clasei respective. In aceasta clasa, pe langa constructor, gettere si settere, mai gasim implementarea metodelor de care avem nevoie pentru rezolvarea cerintelor, respectiv: nrOfDays() – care calculeaza numarul zilelor diferite din fisierul sursa, getActiDist() – care pune intr-un map activitatile distincte, getnrAod() – care gaseste activitatile din fiecare zi, actDuration() – calculeaza durata fiecarei activitati si le filtreaza pe cele care tin mai multe de 10 ore, getFilteredAct() - filtreaza activitatile cu cel putin 90% din samples cu durata mai mica de 5 minute. Toate metodele au fost implementate stream-uri. Stream-ul este un concept abstract care înglobează orice flux de date de la o sursă (canal de intrare) la o destinatie (canal de ieșiere). Bibliotecile I/O folosesc conceptul de **stream** (*flux*), ce reprezinta orice **sursa** sau **consumator** de date care este capabil sa produca sau sa primeasca **unitati** de date, intr-o maniera **secventiala**. Stream-ul **ascunde** detaliile a ceea ce se intampla cu datele in interiorul entitatii I/O, care poate fi:

* **fisier** de pe disc
* **program** (care poseda cele 3 stream-uri standard: in, out, err)
* etc

Clasele Java pentru I/O sunt separate din punct de vedere al operatiei: **input** si **output**. Exista doua linii mari de lucru cu stream-uri, si, anume, la nivel de:

* **octet**: clasele derivate din InputStream si OutputStream. Sunt asociate adesea cu fisierele **binare**.
* **caracter**: clasele derivate din Reader si Writer. Sunt asociate adesea cu fisierele **text**. Acest lucru nu inseamna ca un fisier text nu poate fi privit ca un flux **binar**, dar, daca informatia se doreste a fi interpretata sub forma sirurilor de caractere, este necesara perspectiva **caracter**.

Majoritatea functiilor din aceste clase pot arunca **exceptii** de tip IOException sau derivate din aceasta, cum este FileNotFoundException. Acestea sunt **checked**, deci vor trebui prinse.

*public static int nrOfDays(List<MonitoredData> list){  
 return (int) list  
 .stream()  
 .map(m -> m.getStartTime().toDate().getDate())  
 .distinct()  
 .count();  
}  
  
public static Map<String, Long> getActiDist(List<MonitoredData> list) {  
  
 return list  
 .stream()  
 .map(m -> m.getActivityLabel())  
 .collect(groupingBy(Function.identity(), Collectors.counting()));  
}  
  
public static Map<Integer,Map<String,Long>> getnrAod(List<MonitoredData> list){  
 return list  
 .stream()  
 .collect(groupingBy(m->m.getStartTime().getDayOfMonth(),groupingBy(m->m.getActivityLabel(),counting())));  
  
}  
  
public static Map<String, Long> actDuration(List<MonitoredData> list){  
 Map<String, Long> rez= list  
 .stream()  
 .collect(Collectors.groupingBy(p -> p.getActivityLabel(),  
 Collectors.summingLong(  
 p -> (p.getEndTime().getMillis() - p.getStartTime().getMillis())  
 / 3600000)))  
 .entrySet().stream().filter(m -> m.getValue() > 10)  
 .collect(Collectors.toMap(m->m.getKey(), m->m.getValue()));  
 return rez;  
  
 }  
  
public static List<String> getFilteredAct(List<MonitoredData> list) {  
  
 Map<String,Long> a=getActiDist(list);  
 Map<String, Long> b = list.stream().filter(  
 m -> m.getEndTime().getSecondOfDay() - m.getStartTime().getSecondOfDay() < 300)  
 .map(m -> m.getActivityLabel())  
 .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));  
 return list.stream()  
 .filter(m -> b.get(m.getActivityLabel()) != null  
 && b.get(m.getActivityLabel()) >= 0.9 \* a.get(m.getActivityLabel()))  
 .map(m -> m.getActivityLabel()).distinct().collect(Collectors.toList());  
  
}*

In clasa Main am citi cu ajutorul stream-ului datele dintr-un fisier text, linie cu linie, spart in bucati, pe care le-am pus in variabilele porivite. Tot aici am apelat metodele din clasa MonitoredData si rezultatele lor le-am pus in fisiere txt cu ajutorul PrintWriter-ului.

# Concluzii si Dezvoltari Ulterioare

Acest proiect mi-a imbogatit cunostintele in folosirea a Java 8. Am invatat sa folosesc inerfete functionale, stream-uri si expresii lambda. Ca si dezvoltari ulterioare as mentiona imbunatatirea eficientei codului si crearea unei interfete grafice cu un meniu care sa selecteze operatia dorita.

# Bibliografie

* <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>
* <http://www.mkyong.com/tutorials/java-8-tutorials/>
* <https://www.tutorialspoint.com/java8/java8_lambda_expressions.htm>
* <https://stackoverflow.com/>
* <http://cursuri.cs.pub.ro/~poo/wiki/index.php/Input/Output>