*Universitatea Tehnica din Cluj Napoca*

*Facultatea de Automatica si Calculatoare* 

***Stream Processing using Lambda Expressions***

*Nume : Luca Andrada Aurelia*

*Grupa: 30228*

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

*Cuprins:*

1. Enuntul problemei . . . . . . . . . . . . . . . . . . . . . . . . . . . .
2. Modelare, scenarii . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
3. Proiectare . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
4. Implementare . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
5. Rezultat . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
6. Dezvoltari ulterioare . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
7. Concluzii, ce s-a invatat din tema . . . . . . . . . . . . . . . . . .
8. Bibliografie . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

1 . Enuntul Problemei

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. 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 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> 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 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 containing only the distinct activity names and write the result in a text file

Java 8, aduce impreuna cu dezvoltarea sa, aparitia unor imbunatatiri si noutati. Tema are in prim plan un sistem inteligent de monitorizare a unei locuinte, care ofera informatii despre diferitele activitati zilnice, acestea fiind reprezentate printr-un fisier text.  
Rezolvarea cerintelor acestui proiect, are ca scop procesarea datelor din fisierul text respectiv, utilizand noutatile oferite de Java 8, care permit evidenţierea unor aspecte de performanţă, productivitate şi de reducere a dimensiunii codului scris.

3.Proiectare

Am ales sa abordez problema in mod simplu si direct, modeland aplicatia in doar cateva clase simple si esentiale: clasa MonitorData, pentru a reprezenta timpul si data inceperii si terminarii actiunilor, clasa Operatii, unde am scris metodele pentru rezolvarea cerintelor, clasa Test, unde lansez aplicatia propriu-zisa  
  
-Clasa MonitoredData:

**private** **String** startTime;

**private** **String** endTime;

**String** activityLabel;

**public** **MonitoredData**(**String** d1, **String** d2, **String** d3)

{

this.startTime = d1;

this.endTime = d2;

**this**.activityLabel = d3;

}

Este alcatuita din 3 field-uri, de tipul String, care reprezinta momentul in care incepe o actiune si momentul in care se termina, si o variabila de tip String prin care se evidentiaza numele activitatii.  
Constructorul clasei aduce cele trei field-uri la valorile lor initale.

Exercitiul 1:

Set<Integer> **s** = m.stream().map(**t**->t.getZi()).collect(**Collectors**.*toSet*());

Acesta are ca obiectiv numararea zilelor distincte care apar in fisier. Am folosit pentru rezolvarea acestui exercitiu, expresii lambda si stream-uri.a

Metoda primeste ca parametru o lista de tipul MonitoredData, introducand mai apoi zilele inceperii activitatilor intr-o structura de date Set, pentru a avea siguranta ca aceasta memoreaza doar elemente dinstincte. Size-ul set-ului reprezinta numarul totala de zile distincte din fisier.

Exercitiul 2:

List<String> **result1** = list.stream().sorted(Comparator.*comparing*(**n** -> n.toString()))

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

Map<String, Long> **result** = result1.stream().collect(**Collectors**.*toList*()).stream()

.sorted((**e1**, **e2**) -> e2.compareTo(e1))

.collect(**Collectors**.*groupingBy*(Function.*identity*(), **Collectors**.*counting*()));

Exercitiul doi, grupeaza activitatiile intr-un map, determinand numarul de aparitii ale acestora. Map-ul are un key de tip String, reprenzentat fiind de numele activiatii, iar value de tip long, valoarea returnata de metoda Collectors.counting(), care numara numarul de aparitii ale activitatiilor.

Exercitiul 3:

Map<String, Long> **result2** = mm.stream()

.collect(**Collectors**.*groupingBy*(MonitoredData::getActivity, **Collectors**.*counting*()));

// System.out.println(result2);

Map<Integer, Map<String, Long>> **r** = mm.stream().collect(**Collectors**.*groupingBy*(MonitoredData::getZi,

**Collectors**.*groupingBy*(MonitoredData::getActivity, **Collectors**.*counting*())));

Exercitiul trei, presupune aceeasi cerinta precum cel anterior, dar numarul de aparitii al activitatiilor este acum grupat in functie de zile.  
Acesta returneaza un Map care are un Key de tip Integer, ce reprezinta ziua, si value reprezentat de un alt Map obtinut la fel ca in exercitiu anterior.

Exercitiu 4:

Map<String, Double> **d** = m.stream().collect(

(**Collectors**.*groupingBy*(MonitoredData::getActivity, **Collectors**.*summingDouble*(MonitoredData::getTimp))));

Map<String, Double> **dd** = d.entrySet().stream().filter(**map** -> map.getValue() >= 36000)

.collect(**Collectors**.*toMap*(**p** -> p.getKey(), **p** -> p.getValue()));

Exercitiu numarul 4 determina durata totala a fiecarei activitati si filtreaza map-ul astfel incat acesta sa contina doar activitatile care dureaza mai mult de 10 ore.  
Am construit initial Map-ul, grupand in functie de numele activitatilor si folosind metoda sum, pentru a determina suma tuturor duratelor regasite in fisier.  
Mai apoi am salvat intr-un alt map, primul map filtrat.

Exercitiu 5:

Map<String, Integer> **d** = m.stream().collect(

(**Collectors**.*groupingBy*(MonitoredData::getActivity, **Collectors**.*summingInt*(MonitoredData::sub))));

**int** **a** = durataPeste(m);

**System**.***out***.println(a);

result = d.entrySet().stream().filter(**x** -> x.getValue() == durataPeste(m)).map(**x** -> x.getKey())

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

**Concepte OOP folosite:**

**Incapsularea** (engleza: encapsulation) este proprietatea obiectelor de a-si ascunde o parte din date si metode. Din exteriorul obiectului sunt accesibile ("vizibile") numai datele si metodele publice. Putem deci sa ne imaginam obiectul ca fiind format din doua straturi.

Obiectul se comporta ca si cand ar avea doua "invelisuri": unul "transparent", care permite accesul la datele si metodele publice ale obiectului, si un al doilea invelis "opac", care cuprinde datele si metodele invizibile (inaccesibile) din exterior. Starea obiectului depinde atat de datele publice, cat si de cele incapsulate. Metodele publice ale obiectului au acces la datele si metodele incapsulate (ascunse) ale acestuia. In consecinta, starea obiectului poate fi modificata atat prin modificarea directa, din exterior, a valorilor variabilelor publice, fie prin utilizarea unor metode publice care modifica valorile variabilelor incapsulate. In mod similar, valorile variabilelor incapsulate pot fi obtinute numai utilizand metode publice ale obiectului respectiv.

Incapsularea obiectelor prezinta avantaje importante in programare, deoarece mareste siguranta si fiabilitatea programelor, prin eliminarea posibilitatii modificarii accidentale a valorilor acestora ca urmare a accesului neautorizat din exterior. Din aceasta cauza, programatorii evita in general sa prevada intr-un obiect date publice, preferand ca accesul la date sa se faca numai prin metode.

Partea vizibila (publica) a obiectului constituie interfata acestuia cu "lumea exterioara". Este posibil ca doua obiecte diferite sa aiba interfete identice, adica sa prezinte in exterior aceleasi date si metode. Datorita faptului ca partea incapsulata difera, astfel de obiecte pot avea comportament diferit.

**Polimorfismul** (engleza: polymorphism) permite ca aceeasi operatie sa se realizeze in mod diferit in clase diferite. Acesta fiind folosit in cazul celor trei clase care opereaza asupra tabelelor, si anume executa query-uri.

**Java 8**

Java8 este considerată și promovată ca cea mai importantă lansare Java din istorie, marcând momentul în care este unificat ecosistemul Java embedded. Odată cu Java 8, Java ME devine un subset pentru Java SE și cele două platforme vor avea cicluri de lansare sincronizate.

Un alt element important în cazul Java SE8 e legat de optimizarea codului, excelentă pentru utilizarea în medii multi-core și multi CPU. Mi-a atras atenția și o altă declarație interesantă: codul Java pe mașina JVM are acum performanțe egale cu cel compilat nativ, pe anumite configurații hardware.

**Expresii lambda**

Unul dintre cele mai importante concepte introduse de Java 8 este cel al expresiilor functionale. Acestea deschid calea catre posiblitatea implementarii conceptelor de [programare functionala](https://en.wikipedia.org/wiki/Functional_programming" \o "https://en.wikipedia.org/wiki/Functional_programming) in java.

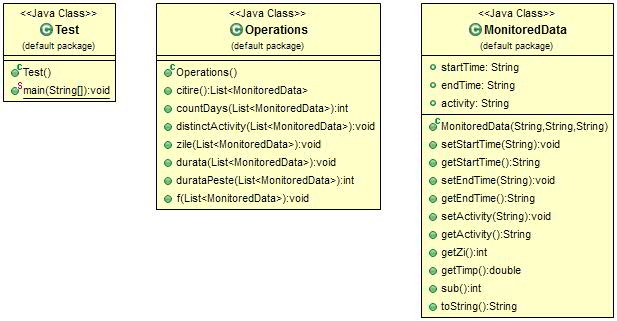
O expresie lambda este o secventa de cod care poate fi transmisa pentru executie imediat sau la un moment ulterior de timp.

Acestea reprezintă cele mai importante feature-uri ale versiunii actuale.

**Stream**

Pe scurt, stream-urile reprezintă o secvenţă de date de lungime nedeterminată. Aşa cum sugerează termenul de stream, este vorba despre un flux de date. Conceptual secvenţa sau fluxul de date poate fi asemănat cu curgerea unui rîu. Practic curgerea unui rîu nu are vreun sfîrşit. La fel se întîmplă şi în cazul unui stream - reprezintă o curgere continuă de date.

În Java, stream-urile sînt compuse din octeţi. Aceşti octeţi pot reprezenta caractere ASCII, numere întregi sau orice altceva. Ei pot "curge" mai repede sau mai încet, astfel încît programul nostru trebuie să se adapteze la viteza lor pentru a-i capta. De multe ori se poate întîmpla ca în primele faze de implementare programul nostru să se "mişte mai încet" decît octeţii din stream. Oricum ceea ce este important este faptul că, aproape întotdeauna, stream-urile sînt procesate în cadrul unui ciclu de tip while.



Pentru exercitiul 1, rezultatul se va afisa in Consola, de asemenea si continutul fisierului. Pentru restul exercitiilor, cerinta impune salvarea acestuia intr-un fisier .txt.

**Writer** **writer** = **new** BufferedWriter(

**new** OutputStreamWriter(**new** FileOutputStream(**new** File("4.Durata.txt")), **StandardCharsets**.***UTF\_8***));

dd.forEach((**key**, **value**) -> {

**try** {value=value/3600;

writer.write(key + " - " + value.intValue()+ **System**.*lineSeparator*());

} **catch** (**IOException** **e**) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

});

**try** {

writer.flush();

writer.close();

} **catch** (**IOException** **e**) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

Rezultat  
  
Rezultatul se prezinta sub forma unei aplicatii, prin intermediul careia utilizatorul are posibilitatea de a vedea rezultatele furnziate de procesarea fisierului text.  
Aceste rezultate sunt vizibile in log text sau intr-un fisier .txt in functie de cerinta.

**Ce am invatat din tema ?**

In primul rand, consider ca mi-am imbogatit cunostintele OOP-ului, reusind acum sa aplic mult mai bine teoria si conceptele invatate de-a lungul semestrului intai din acest an.

In al doilea rand, datorita faptului ca am utilizat Java Swing, consider ca am invatat multe lucruri utile pe viitor si multe noutati bune de pus in practica.

Totodata, am invatat lucruri noi ale limbajului java. Am invatat sa folosesc concept noi cum ar fi :Stream, Lambda Expressions.

Nu in ultimul rand, odata cu acest proiect am inteles modul de gandire si de concepere a acestuia, reusind sa imi structurez ideile si sa le analizez corect. In urma argumentelor mentionate, am reusit sa duc la final cerinta temei proiectului.

**VII. Bibliografie**

**-Pentru diferite probleme intampinate in crearea aplicatiei:**

1. Java de la 0 la expert ( ediția a II – a ), Autor (i) : Ștefan Tănasa, Cristian Olaru, Ștefan Andrei, Editura : Polirom, An apariție : 2007

2. [http://stackoverflow.com](http://www.serkey.com/tag/java-polynomial-division/)

3. Introducere în universul JAVA, Autor (i) : Horia Georgescu, Editura : Tehnica

4. <http://docs.oracle.com/javase/tutorial/uiswing/>

5. <http://wikipedia.com>

6. Tutoriale Youtube

**-Surse scolare:**

1. Cursul de Tehnici de programare al domnului profesor Ioan Salomie

2. Cursul de Programare Orientata Obiect al domnului profesor Giosan Ionel

3. Documentatia laboratorului de Programare Orientata Obiect

**-Programele folosite in dezvoltarea aplicatiei au fost :**

Eclipse, utilizat pentru implementarea in Java cat si pentru crearea diagramei UMLa