**Tehnici de programare fundamentale**

**Tema 5**

**Analyzing the behavior of a person recorded by a set of sensors**

Pentek Tamas

Grupa: 30228

**1. Obiectivul temei**

Obiectul principal al temei este crearea unui program care foloseste expresii Lambda si Stream-uri. Programul porneste de la un fisier care contine activitatile unei persoane preluate de niste senzori. Fiecare activitate are un timp de inceput, timp de sfarsit si o denumire si pe baza acestor informatii, programul calculeaza anumite lucruri din care putem sa aflam comportamentul persoanei, adica cat timp petrece cu diferite activitati pe parcursul mai multor zile. Aceasta aplicatie nu are o interfata grafica, rezultatele sunt afisate in consola. Utilizatorul trebuie sa ruleze o data aplicatia si dupa aceea toate rezultatele apar in consola.

**2. Analiza problemei, modelare, scenarii, cazuri de utilizare**

Aceasta aplicatie nu este foarte complexa, utilizatorul este limitat la o singura operatie si anume sa deschide aplicatie si sa ruleze o data, dupa care toate rezultatele sunt afisate in consola de unde poate sa obtina niste informatii despre persoana a carei comportament a fost analizata pe baza unui fisier, care contin date preluate de la niste senzori. Daca apare ceva dupa rularea programului, utilizatorul este ajutat prin niste informatii care sunt afisate tot in consola. De exemplu daca programul nu a reusit sa deschida fisierul care contine informatia sau nu a reusit sa citeasca de acolo, un mesaj corespunzator va aparea in consola. O alta problema ce poate apare este conversia formatul de data, dar utilizatorul este ajutat si aici, daca ceva nu functioneaza corect, un mesaj de eroare va fi afisat in consola.

Fisierul respectiv contine informatii preluate de la niste senzori, mai exact fiecare activitate este caracterizat prin: data si timpul de inceput, data si timpul de sfarsit si denumirea activitatii(Lunch, Dinner, Snack, Spare\_Time/TV, Grooming). Un rand din acest fisier arata in felul urmator:

2011-11-28 20:20:55 2011-11-28 20:20:59 Snack

Prima coloana reprezinta timpul de inceput a doau coloana timpul de sfarsit si a treia coloana denumirea activitatii. Senzorii au adunat informatii pe baza mai multor zile, de aceea putem sa spunem ca avem o viziune destul de buna asupra comportamentului persoanei respective.

**3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)**

Acest proiect se afla intr-un singur pachet cu nume Lambda.AnalyzingTask si acest pachet contine doua clase. Avem doar doua clase din cauza faptului ca folosim expresii Lambda si Stream-uri care ne ajuta sa avem mai putin cod, dar totusi avem o complexitate destul de importanta pentru ca lucram cu multe date si facem operatii pe volum mare de date. Prima clasa este MainClass care contine metoda main, metoda din care este pornita aplicatia. In aceasta clasa avem doar apelul metodelor din cealalta clasa si niste metode prin care afisam in consola informatii utile si rezultatul. Cealalta clasa contine mai multe metode, dar are o dimensiune destul de redusa.

Pentru fiecare cerinta gasim o metoda, in care folosim expresii Lambda prin care reducem dimeniunea clasei cu multe randuri, prin acesta scrierea codului fiind mult mai simpla. Langa metodele care folosim pentru rezolvarea cerintelor, mai avem niste metode getter si setter. Clasa MonitoredData are trei atribute: startTime, endTime, activity; fiecare dintre acestea reprezinta o bucata de informatie din fisierul respectiv, iar cele trei atribute impreuna reprezinta un rand din fisier. Clasa are un singur constructor, care are trei parametrii, aceste fiind utilizate pentru cele trei atribute ale clasei.

**4. Implementare**

Pe baza acestui proiect se afla expresiile Lambda si Stream-uri. O functie Lambda este o functie definita si apelata fara a fi legata de un identificator. Functiiel Lambda sunt o forma de functii „incuibate” (nested functions) in sensul ca permit accesul la variablile din domeniul functiei in care sunt continute. O expresie Lambda consta dintr-o lista de parametri formali, separati prin virgula si cuprinsi eventual intre paranteze rotunde; dintr-o sageata directionala si un bloc, un body ce consta dint-ro expresie sau un bloc de instructiuni. O expresie Lambda are ca si tip o interfata functionala (Functional Interface), expresia nu are tip de returnat, nu are nume. Interfata functionala se refera a o interfata care contine o singura metoda abstracta. Tipul exact a unei expresie Lambda este determinat de compilatorul Java.

Stream-ul este o secventa de elemente dintr-o sursa care sustine oepratii agregete. Prin folosirea Stream-urilor operatiile efectuate pe o colectie pot fi mult mai complexe si anume: filtrarea dupa un predicat de selectie, maparea obiectului filtrat, respectiv executarea unei actiuni pe fiecare obiect mapat. In contrast cu Collections, care sunt iterate explicit, operatiile Stream fac acest lucru implicit, intern.

Prima metoda din clasa MonitoredData este getData() care este o metoda statica. Aceasta returneaza o lista de obiecte de MonitoredData care este creat citind informatiile din fisier. Prima data in aceasta metoda este create un obiect Stream din fisierul respectiv, dupa care Streamul respectiv este convertit intr-o lista de String-uri, fiecare String reprezinta o sigura linie din fisier. Dupa aceea este creata o expresie Labda care este de tipul interfetei functionala numita Function. Aceasta este o interfata functionala implicita, care are metoda abstracta apply cu un singur parametru. Practic prin aceasta expresie Lambda este impartita, divizita o linia din fisierul respectiv in trei bucati: startTime, endTime si activity. La finalul metodei avem lista de tip MonitoredData care contine tot fisierul. Metoda getData() arata in felul urmator:

**public** **static** List<MonitoredData> getData() {

List<MonitoredData> list = **new** ArrayList<MonitoredData>();

String fileName = "HW5\_Activities.txt";

Stream<String> stream = **null**;

**try** {

stream = Files.*lines*(Paths.*get*(fileName));

} **catch** (IOException e) {

System.***out***.println("Error while creating the stream form the file!");

}

List<String> start = stream.collect(Collectors.*toList*());

Function<String, String[]> split = x -> x.split("[\\t]");

**for** (String s : start) {

String[] newString = split.apply(s);

list.add(**new** MonitoredData(newString[0], newString[2], newString[4]));

}

**return** list;

}

Folosim try catch pentru ca poate sa apara o exceptie de tip IOException, daca nu reusim sa deschidem fisierul sau daca nu reusim sa citim informatia. In acest caz in consola apare un mesaj de eroare.

O alta metoda este countActivities care are un parametru: o lista de obiecte de tip MonitoredData si returneaza un Map de tip <String, Integer>. Aceasta metoda este folosita pentru a calcula numarul total de aparitie a fiecarei activitate in fisierul respectiv. Prima data este declarat un Map, apoi o lista de String-uri care este creat cu ajutorului unui Stream; este aplicat un Stream pe o lista, in care sunt mapate toate activitatile cu nume distincte, deci la final avem o lista cu fiecare activitate care apare in fisier. Dupa aceea este numarat fiecare activitate de cate ori apare in fisierul intreg, aici folosim tot un stream, care filtreaza dupa numele activitatii. Pe baza numele activitatii si numarului de cate ori apare, este construit Map-ul si la final este returnat de aceasta metoda. In clasa Main este afisat un mesaj corespunzator, iar dupa care este afisat si Map-ul, adica perechea <cheia, valoare> folosind un stream, unde cheia reprezinta denumirea activitatii, iar valoarea este numarul de cate ori apare in fisier activitatea respectiva. In ceea ce urmeaza este prezentat metoda:

**public** **static** Map<String, Integer> countActivities(List<MonitoredData> list) {

Map<String, Integer> map = **new** TreeMap<String, Integer>();

List<String> activities = list.stream().map(m -> m.getActivity()).distinct().collect(Collectors.*toList*());

**long**[] count = **new** **long**[activities.size()];

**int** i = 0;

**for** (i = 0; i < activities.size(); i++) {

**int** j = i;

count[i] = list.stream().filter(x -> x.activity.equals(activities.get(j))).count();

map.put(activities.get(i), (**int**) count[i]);

}

**return** map;

}

Metoda folosita pentru ultima cerinta a problemei este getFilteredActivities care este o metoda statica de tip void. Prima data se creeaza o lista de tip Integer cu ajutorul unui stream, prin care este calculat 90% din numarul fiecarei activitate, aceste numere fiind stocate in aceasta lista. Dupa aceea este creat o list de tip String care contine numele fiecarei activitati si dupa aceea lista este sortata, adica elementele apar in ordinea alfabetica in lista respectiva. Int-ro bucla este numarat fiecare activitate de cate ori apare in fisier si daca are o durata mai mica decat 300 de secunde, adica 5 minute, atunci este incrementat un contor. La finalul metodei este verificat daca numarul de aparitii este mai mare sau egal cu 90% din numarul de aparitii a activitatii respectiva. Daca se indeplineste aceasta conditie, atunci activitatea este adaugata intr-o lista iar la final este afisat folosind un stream. Codul metodei arata in felul urmator:

**public** **static** **void** getFilteredActivities(List<MonitoredData> list) {

Map<String, Integer> map = MonitoredData.*countActivities*(list);

List<Integer> filteredValues = map.values().stream().map(x -> Math.*round*((x \* 90f / 100)))

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

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

List<String> activities = list.stream().map(m -> m.getActivity()).distinct().collect(Collectors.*toList*());

List<String> filteredActivities = **new** ArrayList<String>();

activities = activities.stream().sorted().collect(Collectors.*toList*());

**for** (**int** k = 0; k < activities.size(); k++) {

**int** nr = 0;

**for** (MonitoredData m : list) {

**if** (m.getActivity().equals(activities.get(k))) {

LocalDateTime now = LocalDateTime.*parse*(m.getStartTime(), formatter);

LocalDateTime then = LocalDateTime.*parse*(m.getEndTime(), formatter);

**long** sec = ChronoUnit.***SECONDS***.between(now, then);

**if** (sec < 300)

nr++;

}

}

**if** (nr >= filteredValues.get(k))

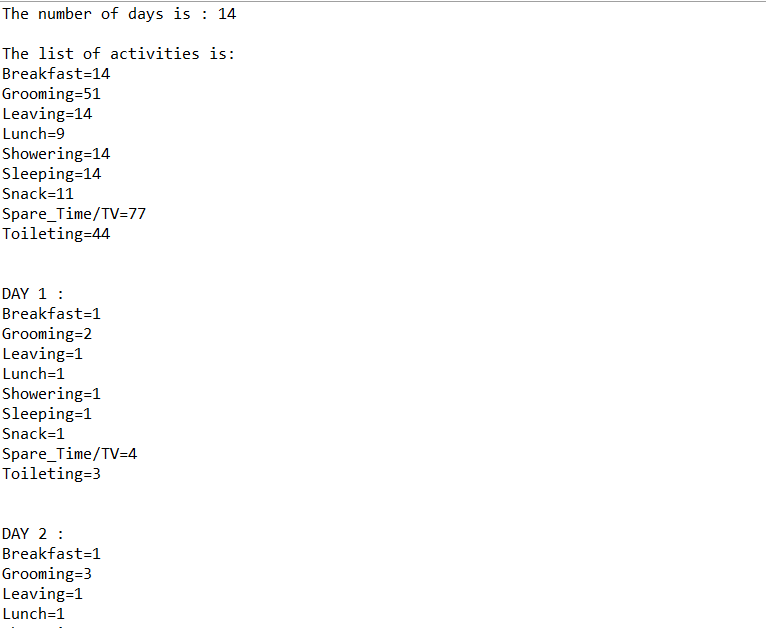
filteredActivities.add(activities.get(k));

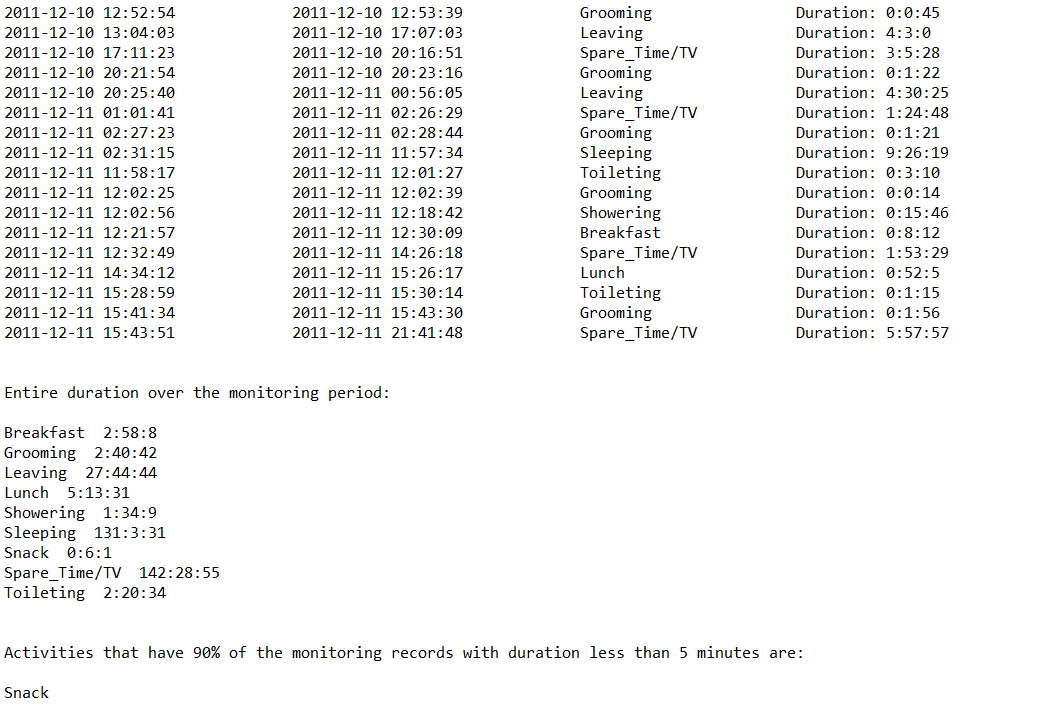
}

filteredActivities.stream().forEach(System.***out***::println);

}

Dupa rularea programului rezultatul apare in consola. O parte din rezultat arata in felul urmator:





**5. Concluzii**

Acest program este destul de folositoare, deoarece in zilele de azi toate lumea se intalneste cu foarte multe senzori, le foloseste niste senzori fara sa stie. Este important sa prelucram datele de la aceste senzori si fiindca avem multe informatie de prelucrat este foarte important sa facem acest lucram eficient si rapid, mai ales daca este vorba o prelucrare real-time. Aceasta tema a fost folositoare deoarece am invatat lucruri noi, am inteles cum functioneaza expresiile Lambda si cum trebuie sa folosesc un Stream, prin aceasta salvand mult timp. Aceast aplicatie se poate dezvolta, se mai pot adauga alte cerinte, alte operatii, putem sa aduagam si alte informatii in fisierul respectiv si putem sa facem niste statistice pe baza acestor informatii. O alta inbunatatire are fi sa facem o interfata grafica, sa facem niste digrame pe baza datelor sau sa salvam rezultatele intr-un fisier.

**6. Bibliografie**

* HW5\_Indications
* https://www.geeksforgeeks.org/stream-in-java/
* https://www.tutorialspoint.com/java8/java8\_lambda\_expressions.htm
* https://docs.oracle.com/javase/8/docs/api/java/time/temporal/ChronoUnit.html
* https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html
* https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html