**DOCUMENTATIE TEMA 5**

**PROCESSING SENSOR DATA OF DAILY LIVING ACTIVITIES**

Cuprins:

1. Obiectivul temei
2. Analiza problemei
3. Proiectare
4. Implementare
5. Rezlutate
6. Concluzii
7. Bibliografie

**1. Obiectivul temei**

Procesarea datelor transmise de senzorii activitatilor zilnice. Aceste date reprezinta timpul de inceput si de sfarsit al unei activitati definite printr-o eticheta specifica(de exemplu “Sleeping, Spare-Time/TV, Toileting, Showering, Breakfast, Grooming, Leaving, Lunch, Snack”). Aceasta procesare se va face cu ajutorul facilitatilor introduse de Java 8.

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

**2.1 Analiza problemei**

Pentru rezolvarea acestei probleme trebuie sa folosim facilitatile introduse odata cu lansarea Java 8. Aceste facilitate, conform documentatiei Oracle sunt:

• **Lambda expression** − Java Lambda Expressions este cea mai apropiată pe care o obțineți în Java de a avea funcții ca obiecte. Expresiile lambda Java sunt similare cu implementările anonime ale interfețelor cu o singură metodă, dar cu o sintaxă mai concisă. Expresiile lambda Java sunt excelente pentru implementarea ascultătorilor de evenimente mici, apeluri de apel sau pentru programarea funcțională în Java, de ex. cu API-ul fluxurilor Java. Adăugă capacitate funcțională de procesare la Java. Expresia Lambda facilitează programarea funcțională și simplifică mult dezvoltarea.

• **Method references** − este sintaxa scurta pentru o expresie lambda care execută doar o metodă. Se refera la funcțiile de trimitere după numele lor în loc să le invoce direct. Utilizarea funcțiilor ca parametru. Într-o referință a metodei, plasați obiectul (sau clasa) care conține metoda înaintea operatorului **::** și numele metodei, fără argumente. În primul rând, o metodă de referință nu poate fi utilizată pentru nicio metodă. Acestea pot fi utilizate numai pentru a înlocui o expresie lambda cu o singură metodă. Deci, pentru a utiliza o referință a metodei, mai întâi aveți nevoie de o expresie lambda cu o singură metodă. Pentru a folosi o expresie lambda, mai întâi ai nevoie de o interfață funcțională, o interfață cu o singură metodă abstractă.

•**Default method** − Înainte de Java 8, interfețele ar putea avea doar metode abstracte. Implementarea acestor metode trebuie asigurată într-o clasă separată. Deci, dacă se adaugă o metodă nouă într-o interfață, atunci codul de implementare trebuie furnizat în clasa care implementează aceeași interfață. Pentru a depăși această problemă, Java 8 a introdus conceptul de metode implicite care permit interfețelor să aibă metode cu implementare fără a afecta clasele care implementează interfața.

• **New tools** − Instrumentele și utilitățile de compilare sunt adăugate ca „jdeps” pentru a descoperi dependențele.

• **Stream API** − Noul flux API pentru a facilita procesarea conductelor. Introdus în Java 8, API-ul Stream este utilizat pentru procesarea colecțiilor de obiecte. Un flux este o secvență de obiecte care acceptă diferite metode care pot fi canalizate pentru a produce rezultatul dorit. Caracteristicile fluxului Java sunt:

-Un flux nu este o structură de date, ci are intrare din canalele Collections, Arrays sau I/O.

-Fluxurile nu schimbă structura de date originală, acestea furnizează doar rezultatul în conformitate cu metodele canalizate.

-Fiecare operație intermediară este executată leneș și returnează un flux ca urmare, prin urmare, diverse operații intermediare pot fi canalizate. Operațiunile terminalului marchează sfârșitul fluxului și returnează rezultatul.

• **Date Time API** − Noua API pentru ora-dată este introdusă în Java 8 pentru a depăși următoarele dezavantaje ale API-ului pentru data dată:

* Fără siguranță pentru filete: Spre deosebire de vechiul java.util.Date care nu este sigur, noua API-ul datei este imuabilă și nu are metode de setare.
* Mai puține operații: în API-ul vechi există doar puține operațiuni de date, dar noul API ne oferă multe operațiuni de date.

Java 8 sub pachetul java.time a introdus o nouă API-dată pentru data, cele mai importante clase dintre ele sunt:

* Local: API simplificată pentru ora dată și fără complexe de gestionare a fusului orar.
* Zonă: API specializată pentru data pentru ora pentru a face față diverselor fusuri orare.

• **Optional** − Clasă opțională Java: fiecare programator Java este familiarizat cu NullPointerException. Îți poate prinde codul. Și este foarte greu să o evitați fără a utiliza prea multe verificări nule. Java 8 a introdus o clasă nouă opțională în pachetul java.util. Poate ajuta la scrierea unui cod îngrijit fără a utiliza prea multe verificări nule. Folosind opțional, putem specifica valori alternative pentru a returna sau cod alternativ pentru a fi rulat. Acest lucru face ca codul să fie mai lizibil, deoarece faptele care au fost ascunse sunt acum vizibile dezvoltatorului.

• **Nashorn, JavaScript Engine** − Nashorn este un motor JavaScript care este introdus în JDK 8. Cu ajutorul Nashorn, putem executa codul JavaScript la Java Virtual Machine. Nashorn este introdus în JDK 8 pentru a înlocui motorul JavaScript existent, adică Rhino. Nashorn este mult mai bun decât Rhino în termeni de performanță. Utilizarea invocării caracteristicii dinamice, conversia codului JavaScript în bytecode direct în memorie etc face ca Nashorn să fie mai celebru în JDK 8. Putem executa cod JavaScript folosind instrumentul din linia de comandă și prin încorporarea codului JavaScript în codul sursă Java.

**2.2 Modelare**

Se poate observa citind rezumatul functionarii fiecarei facilitati, ca vom avea nevoie in principal de expresii Lambda pentru a procesa datele, de Stream API pentru a citi si a schimba diferite structuri de date si de Date Time API pentru a modela timpul de inceput si de sfarsit citit din fisier. Pentru asta vom avea nevoie de o descriere mai detaliata a acestor functionalitati.

**Stream**: o secvență de elemente dintr-o sursă care acceptă operații agregate.

**Sequence of elements**: Un flux oferă o interfață cu un set secvențial de valori ale unui tip de element specific. Cu toate acestea, fluxurile nu stochează de fapt elemente; ele sunt calculate la cerere.

**Source**: Fluxurile consumă dintr-o sursă care furnizează date, cum ar fi colecții, tablouri sau resurse de I / O.

**Aggregate operations**: Fluxurile acceptă operațiuni asemănătoare SQL și operații comune din limbaje de programare funcționale, cum ar fi filtrul, harta, reduce, găsi, potrivi, sorta etc.

**Expresii Lambda**:

- Expresiile Lambda sunt o caracteristică nouă și importantă inclusă în Java SE 8. Ele oferă un mod clar și concis de a reprezenta o interfață de metodă folosind o expresie. Expresiile Lambda îmbunătățesc, de asemenea, bibliotecile Colecției, facilitând iterarea, filtrarea și extragerea datelor dintr-o colecție. În plus, noile funcții de concurență îmbunătățesc performanțele în medii multicore.

- Expresiile Lambda exprimă practic instanțele interfețelor funcționale (o interfață cu o metodă abstractă unică se numește interfață funcțională. Un exemplu este java.lang.Runnable). Expresiile lambda implementează singura funcție abstractă și, prin urmare, implementează interfețe funcționale. Expresiile lambda sunt adăugate în Java 8 și oferă functionalitatiile de mai jos:

* Permite tratarea funcționalității ca un argument de metodă sau codul ca date.
* Funcție care poate fi creată fără a aparține vreunei clase.
* Expresia lambda poate fi transmisă ca și cum ar fi un obiect și executată la cerere.

**2.3 Scenarii**

Diferite scenarii de utilizare pot aparea odata cu schimbarea fisierului din care sunt citite datele, adica diferitele valori returnate de senzori. Aplicatia este construita sa functioneaza pe cazuri generale astfel ca oricare ar fi datele de intrare, datele de iesire sunt deterministe.

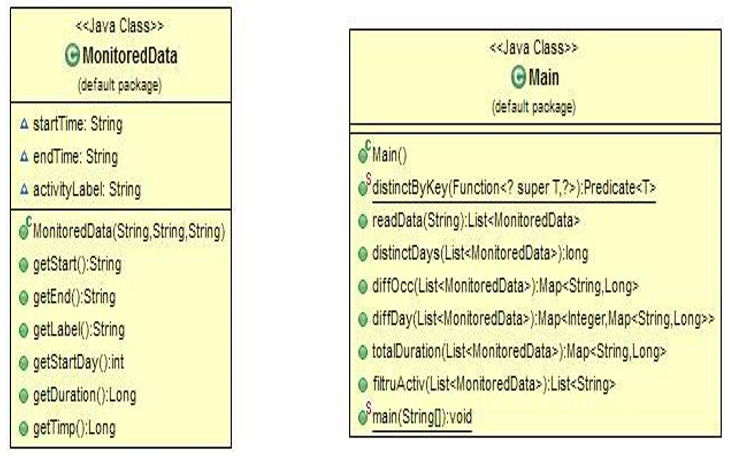
Din fisier se citeste data de intrare si data de iesire, un timp de inceput si un timp de sfarsit ale unor activitati pe care persoanele la au in programul lor zilnic.

**2.3 Cazuri de utilizare**

Pentru acest proiect nu exista cazuri de utilizare, aplicatia procesand orice date de intrare intr-un mod specific. Utilizatorul nu poate influenta datele de intrare care vin de la senzorii activitatilor, deci nu poate infulenta nici datele de iesire. Astfel ca daca datele transmise de senzori sunt valide atunci aplicatia le va filtra corespunzator cerintei.

Trebuie sa determinan numarul total de activitatil pe care le desfasoara o persoana in programul ei, care este cea mai lunga activitate, numarul total de zile care apare in fisierul text dat, intreaga durata a fiecarei activitati.

**3. Proiectare**

****

Proiectarea este relativ simpla, cel putin din punct de vedere al cantitatii de cod scrise. Pentru modelarea datelor transmise am creat o clasa MonitoredData care declara 3 variabile de instanta: startTime, endTime, activityLabel in care voi salva datele de pe ficare linie din fisierul de intrare(o linie corespunde cu o activitate). Pentru procesarea datelor am creat clasa Main care pe langa metodele de procesare contine si metoda main. Pentru procesarea datelor am folosit functionalitatile din Java 8 descrise mai sus.

Pentru metoda cu care citesc datele din fisier, readData(), folosesc urmatoarea secventa de cod :

List<MonitoredData> date = **new** ArrayList<MonitoredData>();  
List<String> buffer = **null**;  
  
**try** (Stream<String> stream = Files.*lines*(Paths.*get*(file))){  
 buffer = stream.collect(Collectors.*toList*());  
}**catch** (IOException e){  
 e.printStackTrace();  
}

Ca si structuri de date folosesc o serie de List<> si Map<>:

-**List<MonitoredData>** in care stochez datele citite din fisier

-**Map<String, Long>** in care voi stoca, pentru fiecare actiune definita cu ajutorul unui label, numarul de aparitii in fisier

-**Map<Integer, Map<String, Long>>** in care voi stoca pentru fiecare zi, numarul de activitati petrecute in acea zi

**4.Implementare**

Clasa MonitoredData in care am implementat cele 3 campuri startTime, endTime si activityLabel:

**public class** MonitoredData {  
 String **startTime**;  
 String **endTime**;  
 String **activityLabel**;  
  
 **public** MonitoredData(String s, String e, String a){  
 **this**.**startTime** = s;  
 **this**.**endTime** = e;  
 **this**.**activityLabel** = a.replaceAll(**"\\p{Z}"**, **""**);  
 }  
  
 **public** String getStart(){  
 **return startTime**;  
 }  
  
 **public** String getEnd(){  
 **return endTime**;  
 }  
  
 **public** String getLabel(){  
  
 **return activityLabel**;  
 }  
  
 **public int** getStartDay(){  
 **return** Integer.*parseInt*(**startTime**.substring(8, 10));  
 }  
  
 **public** Long getDuration(){  
 DateTimeFormatter f = DateTimeFormatter.*ofPattern*(**"yyyy-MM-dd HH:mm:ss"**);  
 LocalDateTime date1 = LocalDateTime.*from*(f.parse(**startTime**));  
 LocalDateTime date2 = LocalDateTime.*from*(f.parse(**endTime**));  
 Duration d = Duration.*between*(date1, date2);  
  
 **return** d.getSeconds()/60;  
  
  
 }  
  
 **public** Long getTimp(){  
 **return** getDuration();  
 }  
}

Implementarea este simpla, avand in vedere ca acesta este si scopul Java 8. Pentru a rezolva problema 2, nu a fost nevoie decat de o simpla linie de cod:

**public long** distinctDays(List<MonitoredData> d){  
 **long** dist = d.stream()  
 .filter(*distinctByKey*(p->p.getStart().substring(0,10)))  
 .count();  
  
 System.***out***.print(**"TASK\_2->Numarul de zile diferite:"**);  
 **return** dist;  
}

Din punctul acesta de vedere doar citirea a necesitat inca o structura repetitiva care pentru fiecare linie citita sa o imparta in datele specifice clasei MonitoredData:

**for** (String i : buffer){  
 String[] parts = i.split(**" "**);  
 String[] parts1 = parts[1].split(**" "**);  
 String[] parts2 = parts[2].split(**" "**);  
 date.add(**new** MonitoredData(parts[0]+**" "** + parts1[0],parts1[1]+ **" "** + parts2[0],parts2[1]));  
}

Implementarea pentru problema 3:

**public** Map<String,Long> diffOcc(List<MonitoredData> d){  
 **final** Map<String,Long> map = d.stream().collect(Collectors.*groupingBy*(MonitoredData::getLabel,Collectors.*counting*()));  
 **try**{  
 Files.*write*(Paths.*get*(**"TASK\_3.txt"**),()->map  
 .entrySet()  
 .stream().<CharSequence>map(rez->rez.getValue() + **" "** + rez.getKey()).iterator());  
 }**catch**(IOException e){  
 e.printStackTrace();  
 }  
 **return** map;  
  
}

Implementarea pentru problema 4:

**public** Map<Integer, Map<String, Long>> diffDay(List<MonitoredData> d){  
 Map<Integer, Map<String,Long>> rez = d.stream().collect(Collectors.*groupingBy*(MonitoredData::getStartDay,Collectors.*groupingBy*(MonitoredData::getLabel,Collectors.*counting*())));  
 **try**{  
 Files.*write*(Paths.*get*(**"TASK\_4.txt"**),()->rez.entrySet()  
 .stream().<CharSequence>map(g->g.getValue() + **" "** + **"\n\n\n\n\n"**).iterator());  
 }**catch**(IOException e){  
 e.printStackTrace();  
 }  
  
  
 **return** rez;  
  
}

Implementarea pentru problema 5:

**public** Map<String,Long> totalDuration(List<MonitoredData> d){  
 **final** Map<String, Long> rez = d.stream()  
 .collect(Collectors.*groupingBy*(MonitoredData::getLabel,Collectors.*summingLong*(MonitoredData::getDuration)));  
  
  
 **try**{  
 Files.*write*(Paths.*get*(**"TASK\_5.txt"**),()->rez.entrySet()  
 .stream()  
 .filter(x -> x.getValue() > 600)  
 .<CharSequence>map(g->g.getValue() + **" "** + **"\n\n\n\n\n"** + g.getKey()).iterator());  
 }**catch**(IOException e){  
 e.printStackTrace();  
 }  
  
  
 **return** rez;  
}

Implementarea pentru problema 6:

**public** List<String> filtruActiv(List<MonitoredData> d){  
 List<String> fin = **new** ArrayList<String>();  
 **final** Map<String,Long> rez1 = d.stream()  
 .collect(Collectors.*groupingBy*(MonitoredData::getLabel,Collectors.*counting*()));  
  
 **final** Map<String,Long> rez = d.stream()  
 .filter(x->x.getDuration() < 5)  
 .collect(Collectors.*groupingBy*(MonitoredData::getLabel,Collectors.*counting*()));  
  
  
 **for** (Map.Entry<String, Long> x : rez.entrySet()){  
 **double** tmp = (**double**)((**double**)rez1.get(x.getKey())/(**double**)rez.get(x.getKey()));  
  
 **if**(tmp == 1){  
 fin.add(x.getKey());  
 }  
  
 }  
  
 **try**{  
 Files.*write*(Paths.*get*(**"TASK\_6.txt"**),()->fin  
 .stream().<CharSequence>map(g->g.toString()).iterator());  
 }**catch**(IOException e){  
 e.printStackTrace();  
  
 }  
 **return** fin;  
  
  
}

Clasa Main:

**public static void** main(String args[]){  
 String fileName = **"Activities.txt"**;  
 Main m = **new** Main();  
 List<MonitoredData> data = m.readData(fileName);  
 m.diffOcc(data);  
 **long** rez = m.distinctDays(data);  
 System.***out***.println(rez);  
 m.diffDay(data);  
 m.totalDuration(data);  
 m.filtruActiv(data);  
}

**5. Rezultate**

Ca rezultat, am obtinut o aplicatie pentru procesarea datelor transmise de o serie de senzori speicifici unor activitati zilnice.

Avand aceasta interpretare a conceptului de” DAILY LIVING ACTIVITIES” putem observa imediat aplicabilitatea unui astfel de program care poate oferi date exacte despre activitatile zilnice pe o perioada indelungata de timp. Asfel ca beneficile folosiri acestuia ar putea avea impact asupra sanatatii si pana la felul cum ne gestionam timpul in fiecare zi.

**6. Concluzii, dezvoltari ulterioare**

In concluzie, tema aceasta m-a ajutat sa inteleg mai bine noile concepte introduse de Java 8 si in principal crearea de cod compact atunci cand avem de a face cu un flux de date mare. Cu

ajutorul stream-urilor si a expresiilor Lambda putem procesa si mapa datele in aproape orice si mod si pe aproape orice structura de date.

Se pot remarca un lung sir de dezvoltari ulterioare, avand in vedere autmatizarea tot mai avansata a fiecarei parti din viata noastra.

**7.Bibliografie**

<http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html>

<http://www.mkyong.com/java8/java-8-streams-filter-examples/>

<https://archive.ics.uci.edu/ml/datasets/Activities+of+Daily+Living+(ADLs)+Recognition+Using+Binary+Sensors>

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>