Documentație TEMA 5 EXPRESII LAMBDA ȘI PROCESE STREAM

Nume: Câmpean Adelina Ioana

Grupa: 30227

Profesor Laborator: Pop Cristina

Cuprins

1. Cerințe Funcționale
2. Informații despre Stream și Expresii Lambda
3. Obiectivul temei
4. Analiza Problemei modelare, scenarii, cazuri de utilizare
   1. Diagrama USE CASE
5. Proiectare
   1. Diagrama de clase UML
   2. Diagrama de pachete
   3. Implementare
6. Rezultate
7. Concluzii
8. Bibliografie
9. Cerințe Funcționale

Luați în considerare sarcina de a analiza comportamentul unei persoane înregistrate de un set de senzori.

Jurnalul istoric al activității persoanei este stocat ca tupluri (start\_time, end\_time, activity\_label), unde start\_time și end\_time reprezintă data și ora la care fiecare activitate a început și sa încheiat în timp ce eticheta de activitate reprezintă tipul de activitate desfășurat de persoană: ieșire, toaletare, duș,

Dormit, mic dejun, prânz, cină, snack, timp de rezervă / TV, îngrijire.

Datele sunt distribuite pe parcursul mai multor zile ca multe intrări în jurnalul Activities.txt, luate din [1,2] și descărcabilă din fișierul Activități.txt din acest dosar.

Scrieți un program Java 1.8 utilizând expresii lambda și procesarea fluxurilor pentru a face sarcinile definite de mai jos.

1. Informații despre Stream

Introdus în Java 8, API-ul Stream este folosit pentru a procesa colecții de obiecte. Un flux este o secvență de obiecte care acceptă diverse metode care pot fi configurate pentru a produce rezultatul dorit.

Caracteristicile fluxului Java sunt:

* Un flux nu este o structură de date, ci o intrare din colecții, arhitecturi sau canale I / O.
* Fluxurile nu modifică structura de date originală, ci furnizează rezultatul numai conform metodelor configurate.
* Fiecare operație intermediară este executată leneș și, prin urmare, returnează un flux, prin urmare pot fi realizate diferite operații intermediare. Operațiile terminale marchează sfârșitul fluxului și returnează rezultatul.
* .map Metoda .map este utilizată pentru a cartografia elementele din colecție către alte obiecte în conformitate cu predicatul trecut ca argument.
* .filter Metoda de filtrare este utilizată pentru a selecta elemente în conformitate cu predicatul trecut ca argument
* .sorted Metoda este utilizată pentru a sorta elementele din Stream

Stream-urile sunt înfășurate în jurul unei surse de date, permițându-ne să operăm cu acea sursă de date și să facem procesarea respectivă convenabilă și rapidă.

Un Stream nu stochează memorie și în acest sens nu este nici o structură de date. De asemenea, nu modifică niciodată sursa de date care stă la baza acesteia.

* 1. Expresiile Lambda

Sunt metode anonime (metode fără nume) folosite pentru a implementa o metodă definită de o interfață funcțională. Este important să știți ce este o interfață funcțională înainte de a vă pune mâinile murdare cu expresii lambda.

Expresiile Lambda introduc un nou operator în Java ” ->”, care împarte expresia în două părți.

Ex: (d-> d\*d)

Partea din stânga reprezintă parametri necesari pentru expresie, care ar putea fi de asemenea si nuli, dacă nu este nevoie de parametri pentru expresia respectivă.

Partea din dreapta, reprezintă corpul expresiei lambda, care specifică acțiunea pe care dorim să o efectuăm. Putem gândi ca și cum operatorul ”->” ar însemna ”devine”, ex. d devine d\*d.

Map<String,Long> dr =

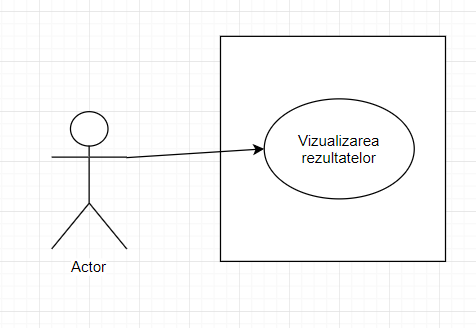
mo.stream()

.collect(Collectors.*groupingBy*(h-> "Din data de :" + h.getStart\_time() + " pana in data de: "+h.getEnd\_time() +" activitatea: "+ h.getActivity()+ " a durat: ", Collectors.*summingLong*(f->*data*(f.getEnd\_time(), f.getStart\_time()))));

1. Obiectivul temei

Această temă are ca scop familiarizarea cu expresiile lambda și cu folosirea de stream-uri, care au fost introduce doar din java 8, pentru o scriere mai ușoară a codului, mai scurtă, în loc de utilizarea mai multor bucle de tip for, se folosește un stream care poate să înlocuiască un for cu un forEach, un if cu un filter, o metodă de sortare cu sorted și de asemenea poate să pună într-o listă, mapă doar anumite zone prin intermediul instrucțiunii map.

1. Analiza Problemei, modelare, scenarii, cazuri de utilizare
   1. Diagrama USE – CASE



Un scenariu pentru diagrama de tip use case, este acela în care utilizatorul scrie în fișier un anumit număr de activități, cu un timp de începere, de terminare și cu numele activității respective. Codul va verifica câte zile au fost monitorizate în lista de activități, de câte ori se execută o anumită activitate în decursul unei zile, sau pe tot parcursul listei, de câte ori apare, și care este durata totală, sau durata pe o anumita activitate.

1. Proiectare
   1. Diagrama de clase UML

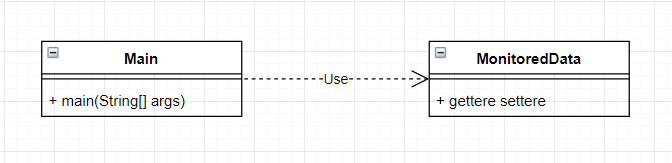
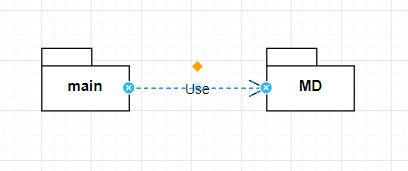


Diagrama contine 2 clase, respectiv 2 pachete, clasa Monitored Data și clasa main, făcând parte din pachetele MD si main.

* 1. Diagrama de pachete



Pachetele componente sunt MD și main.

Pachetul MD

Conține clasa Monitored Data, care este formată dintr-un start time, de tip string, un end time tot de tipul string si activity, care reprezintă numele fiecărei activități, tot de tipul string. Și anume, pentru fiecare activitate, se reține un timp de începere a activității și un timp de finalizare a acesteia. Există 2 constructori, unul cu parametri iar celălalt, fără. De asemenea, sunt și gettere si settere pentru a putea avea acces la variabilele de instanță declarate de tip private. Fiecare activitate, are de asemenea și o metodă to String, care afișează perioada în care a început activitatea, în care s-a finalizat și numele respectivei activități.

Pachetul main

Conține metoda statică main, în care se realizează cerințele.

Prima cerintă, se cerea citirea din fișier a listei de activități și salvarea acesteia într-un array List.

Într-o variabilă de tip String, fname, am salvat numele fișierului, apoi am instanțiat o listă de obiecte de tip MonitoredData, care are un start time, end time si un nume. În interiorul unei clauze try catch() pentru a prinde excepțiile, am folosit un Stream de tipul String pe care l-am inițializat cu Files.lines(Paths.get(fname)) care are rolul de a lua fiecare linie din fișier, cu numele corespunzător. Apoi am ales să folosesc forEach() care este o funcție predefinită a stream-urilor, care ia fiecare linie din stream și o procesează. Adică, în acest caz, pentru fiecare linie din stream, dorim sa îl parsăm în 3 categorii, start time, end time și numele activității. Pentru asta am creat mai întâi un obiect de tip Monitored Data, în interiorul căruia să adăugăm valorile. Apoi, am folosit o expresie lambda, în interiorul metodei de set Start time, sett end time si set activity, care preia substring-ul care ne interesează din linia respectivă. După asta, obiectul nou format, este adăugat în lista de obiecte de tip Monitored data.

Pentru cerința 2, trebuia să numărăm câte zile apar în total pe toata lungimea fișierului. Pentru a realiza acest lucru, am folosit o variabila de tip long în care sa salvez rezultatul, apoi am apelat lista de obiecte Monitored Data . stream, pentru a avea acces la lista și să ne folosim de stream-uri. Am verificat ca timpul de început să fie diferit fată de timpul de final, și anume, pentru fiecare zi, era o activitate, care începea în ziua respectivă, dar se termina în următoarea zi, astfel zilele din start si din end trebuiau sa fie diferite. Prin această metodă, era imposibil să numărăm și ultima zi, deoarece, au fost numărate oarecum, nopțile, astfel se va afișa cu ++nr, pentru a fi numărată și ultima zi.

Pentru cerința 3, trebuia să numărăm câte activități au apărut în total, în lista de monitorizare și să se salveze într-o mapă de tip String si int. Pentru a realiza acest lucru, am creat o mapa de tip String, Long, deoarece metoda care execută numărarea în stream-uri, returează un long. Am inițializat mapa cu mo. stream(), mo fiind lista de obiecte din fișier. Apoi am folosit metoda .collect specifică stream-urilor, care are rolul de a lua valorile din stream si a le salva unde dorim, în acest caz, într-o mapă. Și anume, folosim Collectors. groupingBy, pentru a grupa în mapă, atat un String cât și un Long. String-ul va fi reprezentat de numele activității, pentru asta folosim o expresie lambda, prin referință de metoda ” MonitoredData::getActivity ”, apoi, pentru a lua număra de câte ori apare fiecare activitate, o data ce le-am grupat, folosim Collectors.*counting*() , care returnează un long și acesta reprezintă numărul de câte ori apare fiecare activitate pe întreg fișierul. Astfel, în mapă se va salva, numele fiecărei activități și numărul de câte ori apare.

Pentru cerința 4, se cerea să aflăm de câte ori a apărut fiecare activitate, pentru fiecare zi.

Astfel, am am creat o mapa de tip String, Long, deoarece metoda care execută numărarea în stream-uri, returează un long. Am inițializat mapa cu mo. stream(), mo fiind lista de obiecte din fișier. Apoi am folosit metoda .collect specifică stream-urilor, care are rolul de a lua valorile din stream si a le salva unde dorim, în acest caz, într-o mapă. Și anume, folosim Collectors. groupingBy, pentru a grupa în mapă, atat un String cât și un Long. String-ul va fi reprezentat de data de început pentru fiecare activitate și numele acesteia, astfel încât gruparea cu grouping by nu se mai realizează doar pe baza numelui activității, ci și prin data de început, pentru a lua fiecare zi. Apoi se apelează din nou, Collectors.counting pentru a număra.

Pentru cerința 5, trebuie sa aflăm durata fiecărei activități, de pe fiecare linie. Care se calculează cu end time – start time. Pentru asta, am realizat separat o metodă care convertește un string intr-o variabilă de tip date, dată calendaristică. Acolo, se primesc ca și parametri, 2 string-uri, care reprezintă end time si start time, în interiorul metodei creem o variabilă de tip Date format, pe care o inițializăm cu Simple Date format, care reprezintă formatul datei din fișier. Apoi pentru a prinde excepția Parse, folosim un try catch, în interiorul căruia, în variabile de tip Date, apelăm formatul realizat anterior, cu metoda parse, care primește ca și argument un string, pe care îl convertește în dată, în funcție de formatul pe care l-am oferit noi. Apoi, pentru calculul diferenței, dintre cele 2 date, folosim o variabila long, în care salvăm rezultatul, și facem end – start, apoi convertim valoarea în minute. Am realizat o nouă mapă, în care apelăm mo.stream(), apoi am folosit metoda .collect, în care dorim să grupăm după data de început, data de finalizare și numele activitățiilor, apoi pentru a calcula durata, apelăm metoda realizată anterior, prin folosirea unei variabile lambda, pentru a putea lua parametri, din lista de obiecte, folosindu-ne de gettere.

**public** **static** **long** data (String a, String b ) {

DateFormat form = **new** SimpleDateFormat("yyyy-mm-dd HH:mm:ss");

Date start;

Date end;

**long** fin = 0;

**long** dif = 0;

**try** {

start = form.parse(b);

end = form.parse(a);

fin = end.getTime() - start.getTime();

dif = Math.*abs*(TimeUnit.***MINUTES***.convert(fin,TimeUnit.***MILLISECONDS***));

} **catch** (ParseException e) {

e.printStackTrace();

}

**return** dif;

}

Pentru cerința 6, trebuia sa grupăm toate activitățiile și să vedem cât durează fiecare, pe întrega perioadă de monitorizare. Astfel, singura diferență față de cerința anterioară este faptul că, aici se grupează datele, doar dupa numele activității, fără să mai conteze în ce dată s-a executat fiecare activitate. Iar summingLong adună toate duratele, pentru fiecare activitate.

Pentru cerința 7, trebuia să verificăm care activități au durata mai mică de 5 minute, în 90% din cazuri, pe perioada de monitorizare. Pentru a realiza acest lucru, am folosit mai multe mape. Inițial, am folosit o mapa string, long, în care am salvat fiecare activitate, grupată cu numărul de câte ori apare în listă. Apoi, o altă mapa a fost folosită, pentru a verifica activitățiile care au durata mai mică decât 5 minute, prin metoda .filter. apoi am salvat în mapă, activitățiile grupate care au durata mai mică decât 5 minute, cu numărul de câte ori apar. De ex, activitatea Sleeping apare de 2 ori cu durata mai mică decât 5 minute. Printr-un array list, am parcurs a doua mapă și am verificat, prin filter, dacă valoarea fiecărui element din mapă, adică numărul de câte ori apare, este mai mare sau egal decât 90% din numărul de apariții al fiecarei activități. Adică am realizat 0.9\* numărul de câte ori apare fiecare activitate, apoi am salvat cheia din mapa, și anume string-ul, care reprezintă numele activității care îndeplinește această condiție și am adăugat-o în listă, prin Collectors.toList().

Map <String, Long> actt = mo.stream()

.collect(Collectors.*groupingBy*(d->d.getActivity(),Collectors.*counting*()));

Map <String, Long> m2 = mo.stream()

.filter(g-> *data*(g.getEnd\_time(), g.getStart\_time())<=5)

.collect(Collectors.*groupingBy*(d->d.getActivity(),Collectors.*counting*()));

ArrayList<String> dat = (ArrayList<String>) m2.entrySet().stream()

.filter(d->d.getValue() >= 0.9\*actt.get(d.getKey()))

.map(d->d.getKey())

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

1. Rezultate

Rezultatele sunt vizibile în linia de comandă, fiecare cerință fiind afișată în consolă.

1. Concluzii

În concluzie, această aplicație folosește o nouă metodă de a scrie cod și anume prin intermediul expresiilor Lambda și a stream-urilor. Acest mod de a rezolva o problemă este mult mai elegant, mai scurt și oferă codului un aspect îngrijit, fiind scris scurt și ordonat.

Din această temă am învățat cum să folosesc stream-uri, cum să folosesc expresiile lambda, cum să leg expresiile de stream-uri și mai ales, cum să economisec timp, prin scrierea în acest mod, față de metodele clasice cu multe if-uri si bucle repetitive.

1. Bibliografie

* <https://www.geeksforgeeks.org/stream-in-java/>
* <https://stackoverflow.com/questions/52128545/how-can-i-use-java-8-streams-to-sort-an-arraylist-of-objects-by-a-primitive-int>
* <https://www.baeldung.com/java-date-difference\>
* <https://www.mkyong.com/java8/java-8-foreach-examples/>
* <https://stackify.com/streams-guide-java-8/>