***DOCUMENTAȚIE***

TEMA 5 TEHNICI DE PROGRAMARE

***Expresii Lambda și procesarea stream-urilor***

*Pogăcean Rahela- Alexandra,*

Calculatoare și Tehnologia Informației

Anul II, seria B, grupa 30226

**Cuprins**

1. **Obiective**
   1. **Obiectiv principal**
   2. **Obiective secundare**
2. **Analiza problemei**
   1. **Descriere**
3. **Proiectare**
   1. **Decizii de proiectare**
   2. **Diagrame UML**
   3. **Structuri de date**
   4. **Proiectare clase și algoritmi**
4. **Implementare**
   1. **Metode**
5. **Rezultate și concluzii**
6. **Bibliografie**
7. **Obiective**
   1. **Obiectivul principal**

Obiectivul fundamental al acestei lucrări de laborator este analiza comporatamentului unei persoane înregistrat pe baza unui set de senzori. Acest lucru se revendică a fi realizat prin dezvoltarea și aplicarea mai multor concepte esențiale. În acest fel, proiectarea se face prin utilizarea expresiilor Lambda și a procesării stream-urilor, concepte nemaiîntâlnite până acum.

* 1. **Obiective secundare**

Un prim obiectiv secundar este constituit de respectarea paradigmelor programării orientate pe obiecte, structurarea codului in clase, ceea ce facilitează lizibilitatea și capacitatea de înțelegere a codului. Crearea diagramelor de clase și a use case-urilor reprezintă un alt obiectiv secundar al acestui proiect, precum și dezvoltarea ulterioară, propunându-se o serie de ameliorări prin care aplicația poate deveni mai complexă și mai facil de utilizat.

1. **Analiza problemei**

**2.1 Descriere**

Tema acestui proiect constă în analiza comporatamentului unei persoane înregistrat de un set de senzori. Afișarea istorică și cronologică a activitităților persoanei se realizează într-o manieră specifică, menționându-se timpul de început al activitității (start\_time), timpul de finalizare (end\_time), precum și denumirea activității (activity\_label). Start\_time și end\_time constituie data și ora exactă la care o activitate a început, respectiv s-a terminat, în timp ce activity\_label reprezintă tipul activității desfășurate de o anumită persoană (activități cotidiene, ca de exemplu Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare\_time/TV, Grooming).

Data se întinde pe mai multe zile, numărul echivalând cu numărul de intrări în fișierul Activity.txt. Task-ul acestui proiect constă în scrierea unui program Java 1.8 folosind expresii Lambda și procesare de stream-uri pentru a putea îndeplini cerințele amintite mai sus.

Expresiile Lambda reprezintă în mod fundamental instanțe ale interfețelor funcționale (o interfață cu o singură metodă abstractă se numește interfață funcțională). Un exemplu în acest sens îl constituie java.lang.Runnable. Expresiile Lambda implementează singura funcție abstractă, prin urmare implementează interfețe funcționale. Expresiile Lambda sunt adăugate din Java 8 și furnizează următoarele funcționalități. Permit să se trateze funcționalitățile ca argumente de metode, sau codul ca și date. O funcție poate fi creată fără să aparțină niciunei clase. O expresie Lambda poate fi utilizată drept un obiect sau executată ca o cerere.

O expresie Lambda este o funcție anonimă, o funcție ce nu are o denumire anume și nu aparține niciunei clase. Pentru a crea o expresie lambda, specificăm parametrii de intrare (dacă există) în partea stângă a operatorului lambda -> și plasăm expresia sau blocul de declarații în partea dreaptă a operatorului lambda. De exemplu, expresia (x, y) -> x + y specifică faptul că expresia lambda are două argumente x și y și returnează suma acestora. Spre deosebire de metodele din Java, care au o denumire, o listă de parametri, un corp și un tip returnat, expresiile lambda au doar corp și listă de parametri. Nu au nume, funcția fiind anonimă și nu au tip returnat, compilatorul Java 8 fiind capabil să detecteze tipul returnat verificând codul, de aceea nu e nevoie ca acesta să fie menționat explicit. Pentru a folosi expresii Lambda, trebuie fie să creezi o interfață funcțională, fie să folosești o intefață funcțională predefinită furnizată de Java. O interfață cu o singură metodă abstractă se numește interfață funcțională, de exemplu Runnable, callable, ActionListener.

Printre principalele caracteristici ale unei expresii lambda se numără următoarele. O expresie poate avea zero, unul sau mai mulți parametri. Tipul parametrilor poate fi declarat explicit sau poate fi dedus din context. Parametri multipli sunt incluși în mod obligatoriu între paranteze și separate prin virgule. Parantezele goale sunt folosite pentru a reprezenta o mulțime vidă de parametri. În situația în care există un singur parametru , daca tipul lui este deductibil, nu este obligatorie folosirea parantezelor. Corpul expresiilor lambda poate conține zero, una sau mai multe declarații.

Stream-ul este folosit pentru a procesa colecții de obiecte. Un stream este o secvență de obiecte care suportă diverse metode ce pot fi conectate cu scopul de a produce un anumit rezultat dorit. În cele ce urmează, voi prezenta trăsăturile fundamentale ale stream-urilor. Stream-urile nu modifică structura datelor originale, acestea doar furnizează rezultatul conform metodelor pipelined. O operație intermediară este executată și returnează un stream ca rezultat, prin urmare variate operații intermediare pot fi conectate. Operațiile terminale marchează sfârșitul stream-urilor și returnează rezultatul. Printre operațiile intermediare se numără map, care este folosită pentru a mapa elementele din colecții altor obiecte, conform argumentului de tip Predicate. Filter este folosit pentru a selecta elemente conform Predicate dat ca argument. Metoda sorted este folosită pentru a sorta stream-ul. Operațiile terminale, collect este utilizată pentru a returna rezultatul operațiilor intermediare dezvoltate de stream, forEach este folosită pentru a intera fiecare element din stream, reduce pentru a reduce elementele din stream la o singură valoare și are ca parametru un BinaryOperator.

Câteva observații importante ar fi acelea că un stream constă dintr-o sursă urmată de zero sau mai multe metode intermediare combinate și o metodă terminală pentru a procesa obiectele obținute din sursă conform metodelor descrise. Stream-ul e folosit pentru a calcula elemente potrivit metodelor pipelined fără a altera valoare originașă a obiectelor.

Există diverse tipuri de stream-uri. Acestea pot fi create din diverse surse de date, în mod special colecții. Lists sau Sets (liste sau mulțimi) suportă noi metode stream() si parallelStream() pentru acrea un stream fie secvențial, fie paralel. Stream-urile paralele sunt capabile să opereze cu mai multe thread-uri.

1. **Proiectare**
   1. **Decizii de proiectare**

În proiectarea și implementarea proiectului meu, am încercat să urmăresc scopul principal al acestei lucrări, și anume, folosirea expresiilor lambda și a stream-urilor. Astfel, am ales să structurez codul într-un singur pachet, divizat în două clase (MonitoredData și ToDOs). În cele ce urmează, voi prezenta amănunțit modul în care am abordat rezolvarea acestei probleme. În clasa MonitoredData am declarat 3 variabile esențiale (startTime, ce reprezintă timpul de începere al unei activități, endTime- timpul de finalizare al unei acțiuni și activity, care constituie denumirea activității în sine). Primele două sunt de tipul DateTime și am creat pentru ele o metodă de tipul DateTime, numită createDateTime(String s), care parsează șirul dat pentru a oferi posibilitatea preluării datelor necesare (stringul este divizat în funcție de caracterul ” ”, pentru a accesa data începerii, a terminării, respectiv denumirea activității, apoi, datele sunt împărțite, la rândul lor, după ” -” pentru a afla data calendaristică curentă, apoi după ” :”, pentru a accesa ora curentă.

În clasa ToDOs am implementat metodele necesare îndeplinirii cerințelor proiectului. Metoda getMDFromString(String result) creează un obiect de tipul MonitoredData prin parsarea fiecărei linii după ” \t\t” , detectând timpul de începere, de finalizare, respectiv activitatea corespondentă acestor timpi.

**public** **static** MonitoredData getMDFromString(String result) **throws** IOException {

MonitoredData md = **null**;

**if** (result.contains("\t\t")) {

String[] pieces = result.split("\t\t");

String start = pieces[0];

String end = pieces[1];

String act = pieces[2];

md = **new** MonitoredData(start, end, act);

} **else**

System.***out***.println("Cannot split!\n");

**return** md;

}

Cu ajutorul acestei metode se va putea realiza convertirea stream-ului, a conținutului fișierului într-o listă de obiecte de tipul MonitoredData. Folosind un stream, se preia conținutul fișierului ” Activities.txt ”, apoi, printr-o expresie lambda, se parcurge fiecare linie cu forEach și se adauga obiectul MonitoredData returnat de metoda precedentă în lista monitory. Astfel, întregul fișier devine o listă de date monitorizate.

**public** **static** **void** streamToList() **throws** IOException {

File file = **new** File("Activity.txt");

Stream<String> linesStream = Files.*lines*(file.toPath());

linesStream.forEach(a -> {

**try** {

*monitory*.add(*getMDFromString*(a));

} **catch** (IOException e) {

e.printStackTrace();

}

});

}

Pentru a putea calcula numărul de zile monitorizate pe întreaga perioadă, am creat o metodă ce returnează acest contor, de tipul long, obținut prin preluarea fiecărei zile a fiecărei date de începere (datele calendaristice de începere și finalizare coincid) și apelarea metodei distinct() care elimină duplicatele și a count(), care le contabilizează.

**public** **static** **long** noOfMonitoredDays() {

**long** count = *monitory*.stream().map(a -> a.getStartTime().getDayOfMonth()).distinct().count();

**return** count;

}

Pentru a afla de câte ori a apărut o activitate pe întreaga periodă de monitorizare, am ales să folosesc structura Map<String, Long>, pentru maparea activităților propriului contor. Astfel, metoda Map<String, Long> appearActivities() returnează un map de tipul menționat, un map creat prin parcurgerea listei monitory, a colectării elementelor dorite grupându-le după activitate (groupingBy) și contorizându-le (counting).

**public** **static** Map<String, Long> appearActivities() {

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

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

**return** actt;

}

În îndeplinirea cerinței următoare este necesară contabilizarea numărului de apariții al fiecărei activități pentru fiecare zi a perioadei de monitorizare. Astfel, metoda appearActivitiesPerDay(String myDay) primește ca parametru ziua dorită și se parcurge lista monitory, filtrând elementele în funcție de zi (se verifică daca ziua respectivă echivalează cu ziua data ca parametru), apoi se colectează elementele, grupându-le după activitate și numărându-le.

Pentru a calcula durata unei activități de pe fiecare linie, am implementat metoda recordDuration() ce returnează o listă List<Duration>. Am creat această listă prin parcurgerea fiecarui element al listei monitory, preluarea datelor necesare din acesta și crearea unui obiect de tipul DateTime pentru timpul de start și de finish. Astfel, durata va fi un obiect de tipul Duration(begin, finish), care calculează implicit durata dintre timpii dați ca parametri. Fiecare obiect de tipul Duration se adaugă în listă.

**public** **static** List<Duration> recordDuration() {

List<Duration> dd = **new** ArrayList<>();

*monitory*.forEach((MonitoredData mm) -> {

DateTime begin = **new** DateTime(mm.getStartTime().getYear(), mm.getStartTime().getMonthOfYear(),

mm.getStartTime().getDayOfMonth(), mm.getStartTime().getHourOfDay(),

mm.getStartTime().getMinuteOfHour(), mm.getStartTime().getSecondOfMinute());

DateTime finish = **new** DateTime(mm.getEndTime().getYear(), mm.getEndTime().getMonthOfYear(),

mm.getEndTime().getDayOfMonth(), mm.getEndTime().getHourOfDay(), mm.getEndTime().getMinuteOfHour(),

mm.getEndTime().getSecondOfMinute());

Duration d = **new** Duration(begin, finish);

dd.add(d);

});

**return** dd;

}

Următoarea cerință necesită calculul, pentru fiecare activitate, a întregii durate pe toată perioada de monitorizare. Astfel, am creat un Map<String, Long>, prin parcurgerea listei monitory, a colectării elementelor grupate după activitate și a însumării duratelor activităților. Astfel, am căutat fiecare apariție a fiecărei activități, am accesat durata asociată apariției respective și am însumat-o cu celelelate durate, folosind summingLong. Durata se calculează ca diferență între timpul de finalizare și cel de începere, în modul, returnată în minute.

**public** **static** Map<String, Long> activityDuration() {

Map<String, Long> dur = *monitory*.stream().collect(Collectors.*groupingBy*(m -> m.getActivity(), Collectors

.*summingLong*(a -> Math.*abs*(a.getEndTime().getMinuteOfDay() - a.getStartTime().getMinuteOfDay()))));

**return** dur;

}

În vederea unei mai bune vizibilități a rezultatelor task-urilor, am optat pentru afișarea acestora într-un fișier results2.txt, folosind noțiunile FileWriter (pentru a accesa fișierul) și PrintWriter (pentru a scrie rezultatele).

* 1. **Diagrame UML**

****

* 1. **Structuri de date**

Ca principale structuri de date, am folosit List (monitory, pentru a ține evidența obiectelor MonitoredData, dd- pentru duratele activităților), Map<String, Long> pentru a evidenția numărul de apariții al fiecărei activități, pe întreaga perioadă de monitorizare, dar și pentru fiecare zi, durata fiecărei activități).

* 1. **Proiectare clase și algoritmi**

Modul în care am ales să proiectez clasele a fost prezentat în detaliu în secțiunea 3.2 Decizii de proiectare.

1. **Implementare**
   1. **Metode**

Metodele implementate pentru realizarea cerințelor acestei aplicații sunt getMDFromString(), care creează un obiect de tipul MonitoredData prin parsarea unui string, streamToList() (preiau informațiile din fișier și fiecare obiect MonitoredData creat cu ajutorul metodei precedente este inserat într-o listă). Metoda noOfMonitoredDays() returnează numărul de zile monitorizate de-a lungul întregii perioade, appearActivities() returnează numărul de apariții al fiecărei activități pe întraga perioadă, appearActivitiesPerDay() returnează același număr, însă pentru fiecare zi a perioadei menționate. recordDuration() calculează durata activității de pe fiecare linie și returnează o listă de astfel de durate, iar activityDuration() calculează întreaga durată a fiecărei activități pe toată perioada. Modul în care aceste metode au fost implementate a fost prezentat în secțiunea 3.2 Decizii de proiectare.

1. **Rezultate și concluzii**

Rezultatele sunt afișate într-un fișier results2.txt. Pentru rezultate am folosit tot stream-uri, iar pentru a ajunge la acestea, am citit fiecare linie din fișierul Activities.txt cu ajutorul stream-urilor, am convertit fiecare linie, prin parsare, într-un obiect MonitoredData, cu split, determinând exact timpul de start, de finish și activitatea corespondentă acestor timpi de pe fiecare linie. Fiecare obiect MonitoredData a fost adăugat într-o listă de tipul MonitoredData, numită monitory.

1. **Bibliografie**

<https://stackoverflow.com/questions/53374186/how-to-read-a-file-line-by-line-with-java-stream>

<https://www.baeldung.com/java-write-to-file>

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

<https://www.mkyong.com/java8/java-8-streams-map-examples/>