EXPRESII LAMBDA SI PROCESARE DE STREAM-URI

TEHNICI DE PROGRAMARE

UNIVERSITATEA TEHNICA CLUJ-NAPOCA

FACULTATEA DE AUTOMATICA SI CALCULATOARE

SPECIALIZAREA CALCULATOARE SI TEHNOLOGIA INFORMATIEI

Student Urcan Denisa-Teodora

Grupa 30226 CTI-Romana seria B

Cuprins

1. Obiective

1.1. Obiectiv principal

1.2. Obiectiv secundar

2.Analiza problemei

2.1. Analiza problemei

2.2. Modelarea

3. Proiectare

3.1. Decizii de proiectare

3.3. Structuri de date

3.4. Proiectare clase

3.5. Algoritmi

4. Implementare

5. Rezultate

6. Concluzii

7. Bibliografie

1. Obiective
   1. Obiectivul principal

Subiectul temei consta in propunerea, proiectarea si implementarea unei aplicatii care analizeaza intregistrarile de pe parcursulor mai multor zile, pe baza lor obtinandu-se anumite date statistice. Datele generate se bazeaza pe folosirea exclusiva a stream-urilor si a expresiilor lambda. Cerintele aferente acestei teme sunt respectarea paradigmelor programarii orientate pe obiecte, structurate in pachete, clase cu numar maxim 300 linii de cod si metode cu numar maxim 30 de linii. Implementarea temei respecta conventiile de numire Java, si implicit stilul de scriere camel case.

Pentru a respecta punctajul si a obtine nota maxima, se defineste clasa MonitoredData care are ca si campuri de data(variabile instanta) trei string-uri, reprezentand start-time, sau timpul de inceput, end-time, sau timpul de terminare si activity-label, reprezentand activitatea inceputa si terminata in intervalul de timp specificat. Cerintele specificate sunt respectate in intregime, precum citirea din fisier a unor activitati cu ajutorul carora se realizeaza o lista de obiecte de tipul MonitoredData, urmand monitorizarea activitatilor care apar pe parcursul fiecarei zi.

Se cere pentru fiecare linie din fisier sa retinem timpul in care se intampla activitatea. Se cere pentru fiecare activitate sa obtinem timpul total pe perioada de monitorizare. De asemenea se cere filtrarea activitatilor care au in 90% din datile in care apar, un timp de executie sub 5 minute.

1.2 Obiective secundare

|  |  |  |
| --- | --- | --- |
| Obiectiv secundar | Descriere | Capitol |
| Folosirea structurilor de date convenabile | Structurile de date sunt alese astfel incat sa faciliteze operatiile de adaugare, extragere si sortare a datelor | 3 |
| Impartirea pe pachete | Fiecare pachet este format din mai multe clase care indeplinesc sarcini inrudite | 3 |
| Impartirea pe clase | Fiecare clasa se ocupa de manipularea unui singur obiect, respectiv a unei singure operatii. | 3 |
| Algoritmi utilizati | Algoritmii utilizati pentru a realiza diferitele operatii sunt cei matematici. | 3 |

1. Analiza problemei
   1. Analiza problemei

Analiza problemei presupune identificarea claselor proiectului, functionalitatile pe care acesta le implementeaza, legaturile dintre ele si modelul impus din punct de vedere al structurii. Programarea orientata pe obiecte ofera strategia de dezvoltare „Top-Down”, prin care utilizatorului ii sunt dezvaluite doar informatiile de suprafata. Aplicatia, care are ca si scop principal efectuarea operatiilor unor activitati structurata in clase, fiecare detinand o functionalitate, implementate ulterior.

Datorită strategiei oferite de programarea orientate pe obiecte, se pot identifica clasele, denumite prin substantive, și funcționalitățile, denumite cu ajutorul verbelor.

Asadar, folosind un numar minim de informatii, programarea orientata pe obiecte ne ofera perspectiva implementarii unei aplicatii folosindu-ne doar de infomatii de suprafata.

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.

O funcţie lambda (funcţie anonimă) este o funcţie definită şi apelată fără a fi legată de un identificator. Funcţiile lambda sunt o formă de funcţii ,,incuibate” (nested functions) în sensul că permit accesul la variabilele din domeniul funcţiei în care sunt conţinute.

Funcţiile anonime au fost introduse de către Alonzo Church în anul 1936, în teoria sa despre calculele lambda.

*Lambda expresiile* ne permit să creăm instanţe ale claselor cu o singură metodă într-un mod mult mai compact.

O *lambda expresie* constă:

* dintr-o listă de parametri formali, separaţi prin virgulă şi cuprinşi eventual între paranteze rotunde,
* săgeata direcţională ->,
* un body ce constă dintr-o expresie sau un bloc de instrucţiuni.

O interfaţă funcţională (*functional interface*) este orice interfaţă ce conţine doar o metodă abstractă. Din această cauză putem omite numele metodei atunci când implementăm interfaţa şi putem elimina folosirea claselor anonime. În locul lor vom avea *lambda expresii*. O interfaţă funcţională este anotată cu @FunctionalInterface. Pentru a înţelege modul în care se lucrează cu *lambda expresii* am construit un mic exemplu prin care am creat colecţii de obiecte sortate după diverse criterii. Implementarea interfeţei Comparator a fost făcută într-o clasă anonimă, folosind *lambda expresii*. Implementarea cu *lambda expresii*a fost posibilă pentru că în versiunea 8 Comparator este anotată cu @FunctionalInterface.

2.2 Modelarea

Pentru realizarea acestei aplicatii au fost necesare cateva clase, printre care se numara urmatoarele: MonitoredData in care modelam felul in care este reprezentata o activitate si Rezolvari in care sunt metodele care rezolva fiecare cerinta.

3.1. Structuri de date

Pentru stocarea datelor despre activitati, respectiv date s-a folosit o structura de tip ArrayList, datele fiind introduse ulterior in tabele. Aceste tipuri de date au lungime variabila, care se modifica odata cu operatia realizata. Totodata, datorita faptului ca adaugarea unui client se face la sfarsitul cozii, parcurgerea listei presupune un timp necesar O(n).

3.2. Proiectare clase

Proiectarea claselor imiplica folosirea unor interfete predefinite in cadrul libariilor Java, implementarea interfetei ActionListener de catre clasa Controller permite „ascultarea” si interpretarea comenzilor date de utilizator.

3.3. Algoritmi

Algoritmii care se remarca face referire la modul de implementare a streamu-rilor si a expresiilor lambda.

1. Implementare
   * 1. Clasa MonitoredData

Clasa MonitoredData defineste cateva atribute definitorii: timpul de inceput al activitatii, start-time, timpul de terminare a activitatii, end-time, si activitatea. Clasa implementeaza un constructor specific, care primeste ca argument atributele definitorii descrise mai sus. Variabilele instanta alea clasei sunt private din motive de protejare si ascundere a informatiilor legate de implementare fata de utilizator, motiv pentru care clasa contine metode accesoare si mutatoare. Aceasta clasa se ocupa si de afisarea „prietenoasa” a unui client, pentru a transpune imaginea acestuia cat mai aproape de realitate. Toate tipurile folosite sunt tipuri de data.

Cele trei atribute definitorii definite de aceasta clasa sunt reprezentate sub forma unui sir de caractere(String).

public class MonitoredData {

private String startTime;

private String endTime;

private String activity;

public MonitoredData() {

this.startTime = "";

this.endTime = "";

this.activity = "";

}

public MonitoredData(String startTime, String endTime, String activity) {

this.startTime = startTime;

this.endTime = endTime;

this.activity = activity;

}

*@Override*

public String toString() {

return this.startTime + " " + this.endTime + " " + this.activity + "\n";

}

public String getStartTime() {

return startTime;

}

public void setStartTime(String startTime) {

this.startTime = startTime;

}

public String getEndTime() {

return endTime;

}

public void setEndTime(String endTime) {

this.endTime = endTime;

}

public String getActivity() {

return activity;

}

public void setActivity(String activity) {

this.activity = activity;

}

}

* + 1. Clasa Main

Clasa Main nu defineste atribute specifice, ea fiind responsabila doar de implementarea metodelor cerute in cadrul cerintei. S-au folosit stream-uri atat pentru a parcurge structurile de date cat si pentru a implementa cerintele date. Prima metoda implementata se refera la modalitatea de citire a datelor din fisier, si retinerea lor intr-o structura de date de tip List, Lista definita sa contina obiecte de tip MonitoredData. Astfel, metoda defineste calea completa spre fisierul din care se va realiza citirea ulterioara a datelor, precum si un fisier. Metoda incearca preluarea fiecarei linii si citirea acestei prin intermediul stream-urilor, astfel ca se incearca, pentru fiecare linie in parte, preluarea sa, delimitarea string-ului preluat dupa regex-ul „ ”, si maparea(transformarea) fiecarui element splitat intr-un string. Astfel, datorita constructorului cu atribute din metoda MonitoredData care are ca si argumente 3 stringuri reprezentand timpul de inceput, start time, timpul de sfarsire a activitatii, end time, precum si activitatea. Aceasta metoda arunca exceptia IOException, in cazul in care fisierul nu s-a putut deschide.

public List<MonitoredData> readFromFile() throws IOException {

String fileName = "d://PT2019//pt2019\_30226\_urcan\_denisa\_assignment\_5//Activities.txt";

File file = new File(fileName);

List<MonitoredData> monitoredDataList = new ArrayList<MonitoredData>();

try (Stream<String> streamLine = Files.*lines*(file.toPath())) {

streamLine.forEach(line -> {

// System.out.println(line);

// List<String> splitLine = Arrays.asList(line.split(" "));

// System.out.println("After split " + splitLine);

List<String> splitLine = Stream.*of*(line.split(" ")).map(elem -> new String(elem))

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

// System.out.println("Start time" + splitLine.get(0).split(" ")[0]);

monitoredDataList.add(new MonitoredData(splitLine.get(0), splitLine.get(1), splitLine.get(2)));

});

}

return monitoredDataList;

Metoda de mai jos numara zilele din lista generata de metoda anterioara. Se folosesc stream-uri si expresii lambda pentru a mapa mai intai data de inceput, care este preluata din intregul timp de inceput. Metoda creaza astfel o lista de zile prin stream-ul distinct.count.

public long countDays(List<MonitoredData> monitoredDataList) {

long count = 0;

List<String> days = monitoredDataList.stream().map(x -> Stream.*of*(x.getStartTime().split(" "))

.map(elem -> new String(elem)).collect(Collectors.*toList*()).get(0)).collect(Collectors.*toList*());

// sau puteam face direct monitoredDataList.stream().map(x ->

// x.getStrartTime().split(" ")[0]).collect(Collectors.toList());

count = days.stream().distinct().count();

return count;

}Pentru crearea unei structuri de tip Map<String, Integer> de activitati pe baza listei de activitati create in cadrul primei cerinte, se grupeaza activitatile cu metoda Collectors.groupingBy numarandu-le in acelasi timp.

public Map<String, Integer> mapOfActivities(List<MonitoredData> monitoredDataList) {

Map<String, Long> mapActivities = null;

mapActivities = monitoredDataList.stream()

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

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

mapActivities.forEach((k, v) -> map.put(k.trim(), Integer.*parseInt*(v + "")));

return map;

Crearea unei structuri de acelasi tip pentru fiecare zi s-a folosit o metoda similara, cu o grupare in plus fata de cea de la metoda trecuta.

public Map<String, Map<String, Long>> mapOfActivitiesForEachDay(List<MonitoredData> monitoredDataList) {

Map<String, Map<String, Long>> map = null;

map = monitoredDataList.stream().collect(Collectors.*groupingBy*((x -> x.getStartTime().split(" ")[0].trim()),

Collectors.*groupingBy*((x -> x.getActivity().trim()), Collectors.*counting*())));

return map;

}

Pentru a calcula durata fiecarei activitati, reprezentata de timpul de sfarsit a activitatii si timpul de inceput a activitatii respective pe baza listei generate in cadrul primei cerinte, s-a convertit string-ul preluat cu ajutorul formatului SimpleDateFormat, care dupa parsare detine metoda .getTime(), care transforma data preluata in milisecunde. Se realizeaza diferenta dupa care se mapeaza intr-o lista de Long, urmand sa fie transformata in ora:minute:Secunde prin metoda matematica clasica prin impartirea la 1000(pentru a obtine secunde), apoi /3600 pentru ore, /60%60 pentru a obtine minutele, %60 pentru a obtine secundele.

*@SuppressWarnings*("deprecation")

public Map<String, Map<String, Long>> mapOfActivitiesForEachDay(List<MonitoredData> monitoredDataList) {

Map<String, Map<String, Long>> map = null;

map = monitoredDataList.stream().collect(Collectors.*groupingBy*((x -> x.getStartTime().split(" ")[0].trim()),

Collectors.*groupingBy*((x -> x.getActivity().trim()), Collectors.*counting*())));

return map;

}

}

Durata fiecare activitati se calculeaza pentru fiecare activitate distincta in parte se calculeaza cu ajutorul a doua liste, una in care se salveaza activitatile distincte si una care apeleaza o metoda care returneaza intr-o lista de String-uri durata activitatii. IntStream-ul colecteaza informatiile celor 2 liste si le intrepatrunde(zip).

*@SuppressWarnings*("deprecation")

public List<String> endTimeMinusStartTime(List<MonitoredData> monitoredDataList) {

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

List<Long> dateRecord = monitoredDataList.stream().map(x -> {

try {

return ((simpleDateFormat.parse(x.getEndTime())).getTime()

- (simpleDateFormat.parse(x.getStartTime())).getTime());

} catch (ParseException e) { // **TODO** Auto-generated catch block e.printStackTrace();

return null;

}

}).map(elem -> new ~~Long~~(elem)).collect(Collectors.*toList*());

return dateRecord.stream()

.map(e -> Stream.*of*(" " + e / 1000 / 3600 + ":" + e / 1000 / 60 % 60 + ":" + e / 1000 % 60)

.map(elem -> new String(elem)).collect(Collectors.*toList*()).get(0))

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

}

public void durationOverEachActivity(List<MonitoredData> monitoredDataList) {

List<String> activities = monitoredDataList.stream().map(elem -> elem.getActivity().toString())

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

List<String> act = endTimeMinusStartTime(monitoredDataList);

// System.out.println(activities);

// System.out.println(act);

IntStream.*range*(0, activities.size()).mapToObj(index -> act.get(index) + " " + activities.get(index))

.collect(Collectors.*toList*()).forEach(System.***out***::println);

// System.out.println(res);

}

Durata totala pentru fiecare activitate se calculeaza pentru fiecare activitate. Pentru fiecare activitate in parte, se mapeaza durata dupa care e transformata in formatul clasic hh:mm:ss.

public Map<String, String> totalDurationForEachActivity(List<MonitoredData> monitoredDataList) {

Map<String, String> totalDuration = null;

List<String> activities = monitoredDataList.stream().map(x -> x.getActivity()).distinct()

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

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

for (String activity : activities) {

Long durationPerActivity = monitoredDataList.stream()

.filter(x -> x.getActivity().toString().equals(activity)).mapToLong(x -> {

try {

return ((simpleDateFormat.parse(x.getEndTime())).getTime()

- (simpleDateFormat.parse(x.getStartTime())).getTime());

} catch (ParseException e) { // **TODO** Auto-generated catch block e.printStackTrace();

return (Long) null;

}

}).sum();

String duration = durationPerActivity / 1000 / 3600 + ":" + durationPerActivity / 1000 / 60 % 60 + ":"

+ durationPerActivity / 1000 % 60;

System.***out***.println(activity.trim() + ": " + duration);

// totalDuration.put(activity, durationPerActivity);

}

return totalDuration;

}

Ultima cerinta, reprezentand activitatile care au peste 90% durata mai mica de 5 minute, se realizeaza prin crearea unei liste distincte a activitatilor.

public void activities90Percent(List<MonitoredData>monitoredDataList) {

List<String> activities = monitoredDataList.stream().map(x -> x.getActivity()).distinct()

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

List<String> res = new ArrayList<String>();

Long result;

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

Long up, down;

for(String activity : activities) {

up = monitoredDataList.stream().filter(x -> x.getActivity().compareTo(activity) == 0).filter(y -> {

try {

return (simpleDateFormat.parse(y.getEndTime()).getTime() - simpleDateFormat.parse(y.getStartTime()).getTime()) < 300000;

} catch (ParseException e) {

e.printStackTrace();

return false;

}

}).count();

down = monitoredDataList.stream().filter(x -> x.getActivity().compareTo(activity) == 0).count();

result = up/ down \* 100;

if(result >= 90 ) {

res.add(activity);

}

}

for(String s: res) {

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

}

}

1. Bibliografie

https://stackify.com/streams-guide-java-8/

https://www.mkyong.com/java8/java-8-stream-read-a-file-line-by-line/

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

https://stackoverflow.com/questions/14830313/retrieving-a-list-from-a-java-util-stream-stream-in-java-8

https://stackoverflow.com/questions/24882927/using-java-8-to-convert-a-list-of-objects-into-a-string-obtained-from-the-tostri

https://stackoverflow.com/questions/35650974/create-list-of-object-from-another-using-java8-streams#

https://www.baeldung.com/java-stream-operations-on-strings

https://stackoverflow.com/questions/1780385/java-hashmapstring-int-not-working

https://www.baeldung.com/java-groupingby-collector

https://www.codingame.com/playgrounds/3312/java-8-streams-cookbook/groupby-join-and-grouping