Documentatie Tema 4
Processing Sensor Data Of Daily Living Activities

George Adam

Grupa 30225

Profesor Laborator: Dorin Moldovan

1. Obiectivul temei

Procesarea sub forma de stream este o paradigma a programarii calculatoarelor, echivalenta cu “dataflow programming”, “event stream processing” si “reactive programming” care permite aplicatilor sa foloseasca usor o forma limitata de procesare paralele.

Un stream reprezinta o secventa de elemte si suporta o multitudine de operatii pentru a efectua computatii asupra acestor elemente. Operatiile pe stream-uri sunt fie intermediare fie terminale. Operatiile intermediare returneaza un stream ca sa fie posibila inlantuirea mai multor operatii intermediare fara a fi nevoie de folosirea semnului “;”. Operatiile terminale sunt ori de tip void sau returneaza un rezultat care nu este stream. Spre exemplu, filter, map, si sorted sunt operatii intermediare, pe cand forrEach este o operatie terminala.

Majoritatea operatiilor pe stream-uri accepta un fel de expresie lambda , o interfata functionala care specifica exact cum se comporta operatia. Majoritatea acestor operatii trebuie sa nu intervina si imutabile. Caracteristicile stream-urilor sunt: acestea doar ofera un set de elemente secvential, la comanda si nu salveaza niciodata acest set de elemente; acestea accepta ca intrare Colectii, Array-uri si resurse de I/O ca si intrare. Majoritatea stream-urilor se returneaza pe ele insusi pentrua facilita procesul de pipelining si fiecare stream itereaza automat peste continutul elementelor sursa care ii sunt asignate.

Scopul principal al acestei teme este de a implementa o aplicatie de procesare a informatiilor transmise de niste senzori care transmit date despre activitatile de zi cu zi. Fiecare senzor isi salveaza datele intr-un fisier de tip .txt, pe care aplicatia il foloseste ca informatie de intrare. Aplicatia presupune citirea acestor informatii, realizeaza operatii pe acestea, de filtrare, iar rezultatele le scrie in fisiere de iesire cu nume sugestiv pentru fiecare operatie pe care o face. Operatiile sunt: numararea zilelor distincte care apar in fisier, cautarea activitatilor distincte si de cate ori se efectueaza, etc.

Obiectiv secundar	Descriere	Capitol
Ease of use	Datorita tematicii aplicatiei, aceasta trebuie doar rulata, avand fisierul de intrare, iar ea genereaza automat toate fisierele de iesire care contin informatiile de la operatiile care se efectueaza.	3
Generare	Se genereaza un fisier de iesire cu informatii despre fiecare operatie efectuata, denumit sugestiv.	3
Corectitudinea	Aplicatia permite vizualizarea datelor citite si a celor care rezulta din efectuarea operatiilor, astfel se poate demonstra corectitudinea.	3

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

- Cerinte functionale.

  - Definirea clasei MonitoredData care stocheaza informatiile citite din fisierul de intrare.
  - Citirea informatiilor din fisierul de intrare folosing stream-uri si expresii lambda.
  - Numararea zilelor distincte din fisierul de intrare folosind stream-uri si expresii lambda.
  - Numararea efectuarii fiecarei activitati pe parcursul intrevalului monitorizat folosing stream-uri si expresii lambda.
  - Numararea efectuarii fiecarei activitati in fiecare zi care apare in fisierul de intrare, folosing stream-uri si expresii lambda.

- Calcularea timpului total care apartine fiecarei activitati folosind stream-uri si expresii lambda.
- Filtrarea activitatilor care au durat mai putin de 5 minute in 90% din cazuri, folosind stream-uri si expresii lambda.
- Scrierea in fisier a rezultatelor rezultate de la efectuarea operatiilor folosind stream-uri si expresii lambda.

- Use-cases

Use-case	Solutie
Interfete functionale.	Se definesc interfete functionale pentru a implementa functii cu ajutorul expresiilor lambda. Aceste sunt utile la parsarea datelor de la senzori.
Definirea clasei MonitoredData.	Se creeaza o clasa MonitoredData in care se salveaza informatiile citite din fisierul de intrare.
Alegerea structurilor de date	Se folosesc structuri de date precumm liste si map-uri pentru a stoca datele senzorilor si cele necesare realizarii functiilor.
Afisarea in fisier	Toate rezultatele operatiilor efectuate asupra informatiilor sunt scrise in fisiere de iesire cu nume sugestiv, pentru a putea fi vizualizate si verificate.
Dezvoltarea algoritmilor	Se folosesc algoritmi de parsing.
Impartirea pe clase	Se creeaza clase pentru impartirea functionalitatii in pachete specifice. Nu se foloseste un anume Design Pattern, dar se urmareste lizibilitatea si respectarea paradigmelor OOP.
Implementarea solutiei	Se genereaza in faza initiala o diagrama UML pentru a avea structura proiectului, apoi se incepe implementarea propriu-zisa.
Testare	Se realizeaza prin informatiile afisate in fisierele de iesire specifice fiecarei operatii efectuate pe informatiile de la senzori.

3. Proiectare

- Decizii de proiectare

Proiectul este impartit in urmatoarele clase: FileOperation, App, DataOperations, MonitoredData. Totodata, sunt create interfetele functionale: Task3Util, Task4Util, Task5Util si Task6Util. Acestea sunt folosite pentru a implementa functii specifice implementarii unei anumite operatii cu ajutorul expresiilor lambda.

Clasa FileOperation faciliteaza citirea informatiilor din fisierul de intrare si scrierea unui text intr-un fisier a carui nume poate fi trimis ca parametru.

Clasa App contine functia main si este folosita pentru apelarea functiilor care executa operatiile cerute.

Clasa DataOperations contine o colectie de MonitoredData si implementeaza operatiile pe informatiile de la senzori cerute in cerinta.

Interfetele implementeaza cate o metoda de construire a unu String care va fi scris in fisierul de iesire specific fiecarei operatii.

Se foloseste clasa BufferedWriter pentru scrierea in fisier. Aceasta extinde clasa abstracta Writer si mentine un buffer intern de 8192 characters. Pe parcursul operatiei de scriere, caracterele sunt scrise in buffer-ul intern in loc de disc. Odata ce buffer-ul este plin sau writer-ul este inchis, atunci toate caracterele din buffer-ul intern sunt scrise pe disc. Asadar, numarul de accesari pe disc scade semnificativ, de unde rezulta o viteza crescuta in scrierea pe disk folosind BufferedWriter.

- Structuri de date

Structurile de date folosite sunt:

- List: Stocare informatii pentru parsat sau produse.
- ArrayList: Stocare de informatii despre produse.
- String: folosit la informatiile provenite din GUI si detalii despre produse.
- Integer/Double: pentru a salva preturile si stocurile produselor din baza de date.
- Map: pentru a salva comenzile cu lista de produse din ele.
- HashSet: pentru a salva elemente distincte.

- Proiectare clase

Clasele sunt:

- FileOperation: clasa pentru citire si scriere din fisier.
- App: clasa care contine functia mainunde sunt apelate functile pentru efectuarea operatiilor.
- DataOperations: clasa pentru a genera fisiere cu informatii despre comenzi.
- MonitoredData: clasa pentru salvarea informatiilor dla senzori.

- Algoritmi
  - Parsing:

    Parseaza informatiile pentru a efectua operatiile pe informatiile transmise de senzori.

  - Apeluri:

    Se apeleaza metode pentru prelucrarea informatiilor.

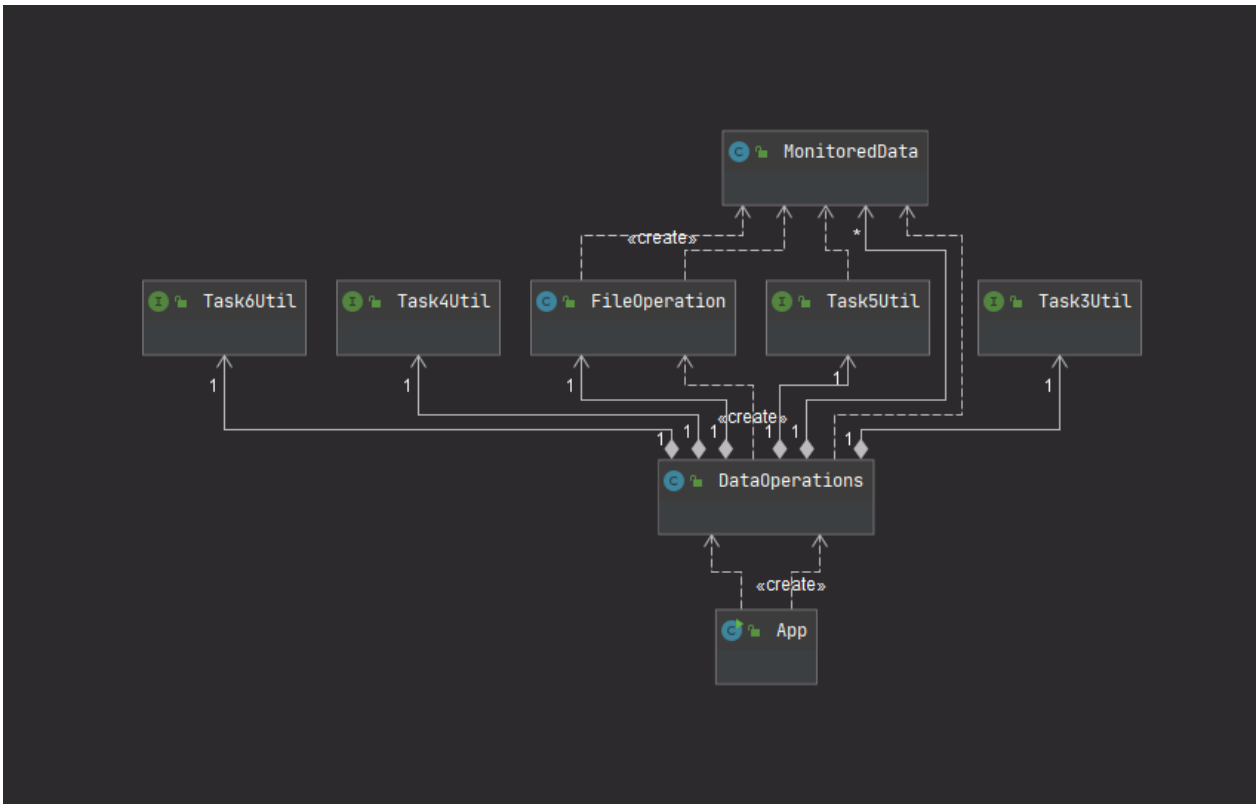
  - Verificarea operatiilor:

    Se creeaza fisiere de iesire pentru a verifica informatiile.

  - Scrierea in fisier:

    Se genereaza fisiere .txt care contin informatii despre operatii cu nume sugestiv.

- Diagrama UML



- Packages

  Pachetele prezente sunt FileIO, Lambdas, Logic.

  Pachetul Lambdas contine interfetele functionale create pentru a implementa functionalitati care se folosesc la realizarea operatiilor pe informatiile de intrare.

  Pachetul FileIO contine clasa FileOperation care implementeaza o functi de citit din fisier si o functie de scriere a orice String intr-un fisier cu numele transmis ca parametru.

4. Implementare

- Task3Util

```
void addOrUpdate(Map<String, Integer> map, String string);

default StringBuilder createString(Map<String, Integer> map) {
    StringBuilder sb = new StringBuilder();
    map.keySet().forEach(line -> sb.append(line).append(" ").append(map.get(line)).append("\n"));

    return sb;
}
```

Interfata functionala Task3Util defineste functia addOrUpdate care urmeaza sa fie implementata cu ajutorul expresiilor lambda. Functia default createString returneaza un String pentru a putea facilita scrierea rezultatului operatiei specifice in fisierul de iesire.

- Task4Util

```
void add(Integer day, Map<Integer, Map<String, Integer>> map);

default StringBuilder print(Map<Integer, Map<String, Integer>> map) {
    StringBuilder sb = new StringBuilder();
    map.forEach((key, value) -> {
        sb.append("Day ").append(key).append(": ").append("\n");
        value.forEach((string, nr) -> {
            sb.append("Activity ").append(string).append(" occurred ").append(nr).append(" times!").append("\n");
        });
    });

    return sb;
}
```

Interfata functionala Task4Util defineste functia add care urmeaza sa fie implementata cu ajutorul expresiilor lambda. Functia default print returneaza un String pentru a putea facilita scrierea rezultatului operatiei specifice in fisierul de iesire.

- Task5Util

```
void add(Map<String, Duration> map, MonitoredData data);

default StringBuilder printDurations(Map<String, Duration> map) {
    StringBuilder sb = new StringBuilder();

    map.forEach((key, value) -> {
        sb.append("Activity ").append(key).append(" has a total duration of ").
                append(parseDate(value.toString())).append("\n");
    });

    return sb;
}

default String parseDate(String string) {
    string = string.replace( target: "PT",  replacement: "");
    string = string.replace( target: "H",  replacement: "H:");
    string = string.replace( target: "M",  replacement: "M:");

    return string;
}
```

Interfata functionala Task5Util defineste functia add care urmeaza sa fie implementata cu ajutorul expresiilor lambda. Functiile defaultprintDurations si parseDate sunt folosite pentru a crea un String care reprezinta rezultatul operatiei specifice si care va fi scris in fisierul de iesire.

- Task6Util

```
AtomicInteger process(String elem);

default StringBuilder printActivities(List<String> list) {
    StringBuilder sb = new StringBuilder();

    list.forEach(string -> sb.append(string).append(" ").append("\n"));

    return sb;
}
```

Interfata functionala Task6Util defineste functia process care urmeaza sa fie implementata cu ajutorul expresiilor lambda. Functia default print returneaza un String pentru a putea facilita scrierea rezultatului operatiei specifice in fisierul de iesire.

- FileOperation

```
public void readFile(List<MonitoredData> monitoredData) throws IOException {
    File file = new File(this.path);
    Stream<String> fileText = Files.lines(Paths.get(file.getPath()));

    fileText.map(line -> line.split( regex: "\t\t"))
            .forEach(parsed -> monitoredData.add(new MonitoredData(parsed)));

}

public void writeToFile(String toWrite, String fileName) throws IOException {
    FileWriter file = new FileWriter(fileName);
    BufferedWriter wr = new BufferedWriter(file);

    wr.write(toWrite);
    wr.close();
}
```

Aceasta clasa este folosita pentru a facilita citirea rapida din fisier si a parsa liniile pentru a imparti informatia in colectia de clase MonitoredData.
Totodata, aceasta clasa implementeaza functia writeToFile care creeaza un fisier cu numele primit ca parametru si scrie String-ul primit ca parametru in el.

- MonitoredData

```
public MonitoredData(String[] parsed) {
    SimpleDateFormat formatter = new SimpleDateFormat( pattern: "yyy-MM-dd HH:mm:ss");

    try {
        startTime = formatter.parse(parsed[0]);
        endTime = formatter.parse(parsed[1]);
        activityLabel = parsed[2];
    } catch (ParseException e) {
        e.printStackTrace();
    }

}
```

Constructorul acestei clase imparte tabloul de String-uri in campurile clasei. Aceasta clasa a fost creata pentru a facilita salvarea informatiilor provenite de la senzori din fisierul de intrare.

- DataOperations

```
public DataOperations(String path) {
    monitoredData = new ArrayList<>();
    fileOperation = new FileOperation(path);
    addLambda = (Map<String, Integer> map, String string) -> {...};
    mapLambda = (Integer day, Map<Integer, Map<String, Integer>> map) -> {...};
    timeLambda = (Map<String, Duration> map, MonitoredData data) -> {...};
    percentageLambda = (String elem) -> {...};

    try {
        fileOperation.readFile(monitoredData);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public void printData() throws IOException {...}

public Integer countDistinctDays() {...}

public void printDistinctDays() throws IOException {...}

private Map<String, Integer> countDistinctActivities() {...}

public void printDistinctActivities() throws IOException {...}

private Map<Integer, Map<String, Integer>> countActivityPerDay() {...}

public void printActivitiesPerDay() throws IOException {...}

private Map<String, Duration> computeDuration() {...}

public void printDurations() throws IOException {
    fileOperation.writeToFile(timeLambda.printDurations(computeDuration()).toString(),  fileName: "Task_5.txt");
}

private List<String> filterActivities() {...}
```

Clasa DataOperation implementeaza fiecare task al acestei aplicatii.

addLambda, mapLambda, timeLambda, percentageLambda sunt implementarile functiilor definite in interfetele functionale mentionate mai sus.

In constructor se citeste si se salveaza continutul fisierului activities.txt. Functia countDistinctDays numara cate zile distincte apar in fisierul de intrare. Functia countDistinctActivities numara si salveaza intr-o colectie de tip Map<String, Integer> cate zile distincte au aparaut pe parcursul perioadei de monitorizare.

Functia countActivityPerDay numara si salveaza intr-o colectie de tip Map<Integer, Map<String, Integer>> cate activitati dintr-un tip s-au efectuat in fiecare zi.

Functia computeDuration calculeaza timpul total petrecut facand o activitate pe parcursul intregii perioade de monitorizare.

Functia filterActivities selecteaza doar activitatile care in 90% sau mai mult din cazurile cand au fost efectuate, au durat mai putin de 5 minute.

5. Rezultate

Aplicatia este salvata sub forma de fisier .jar. Aceasta se ruleaza cu comanda java -jar nume.jar. Fisierul activities.txt care contine informatiile de la senzori de pe intreaga perioada de monitorizare este presetat, deci nu poate fi dat altul ca si parametru in comanda de pornire.

Dupa rularea apicatiei, se vor crea fisierele .txt : Task_1, Task_2, Task_3, Task_4, Task_5, Task_6. Aceste fisiere contin rezultatele de la fiecare operatie pe care aplicatia trebuie sa o execute, asa cum este specificat in cerinta, si sunt denumite dupa care task este reprezentat de operatia respectiva.

6. Concluzii

Ideea cea mai importanta care a rezultat din realizarea acestei teme este, cu siguranta, importanta folosirii, daca nu a unui Design Pattern, a unei impachetari a claselor pentru a face codul lizibil, ordonat si reutilizabil. Totodata, folosirea programarii functionale cu expresii lambda si stream-uri a ajutat foarte mult la realizarea acestei teme intr-un mod foarte curat si ordnonat.

Este prezenta si nevoia de realizare a unui cod “curat”. Necesitatea de a realiza un cod atat de inteligibil incat nu are nevoie de a fi comentat este imperativa, deoarece reintoarcerea la el si/ sau modificarea acestuia devine mult mai usoara. Totodata, impartirea codului in clase si pachete specifice unei anumite probleme permite reutilizarea acestuia si evita scrierea unor bucati largi de cod care vor fi folosite doar o singura data.

Elementul care necesita dezvoltare ulterioara este implementarea unei interfete grafice pentru a usura utilizarea aplicatiei, sau pentru a selecta doar o parte din informatiile care trebuie efectuate. Totodata, aplicatia poate fi extinsa prin adaugarea informatiilor mai detaliate, care necesita adaugarea de noi senzori pentru a furniza aceste informatii. Odata cu introducerea a informatii mai detaliate, va trebui ca aplicatia sa fie extinsa din punct de vedere al functionalitatii, pentru a putea lucra cu acestea. Aceasta din urma nu este o problema, deoarece aplicatia este structurata si creata in asa fel incat sa permita dezvolatea usoara ulterior, astfel ca ce a fost implementat pana acum sa nu fie considerat o problema la implementari ulterioare.

Expresiile lambda sau functiile anonime sunt un concept in programarea calculatoarelor care semnifica o functie care nu este legata de un identificator. Expresiile lambda sunt de obicei argumente care sunt pasate la functii de ordin superior, sau folosite pentru constructia rezultatului unei functii de ordin superior care trebuie sa returneze o functie. Daca functia este folosita doar o data, sau de un numar limitat de ori, atunci o expresie lambda s-ar putea sa consume mai putine resurse decat o functie cu nume. O expresie lambda este construita dintr-o lista separata de virgula care contine parametru formali. Tipul de date al acestor parametru poate fi omis tot timpul, iar corpul lor poate fi o singura instructiune sau un bloc de instructiuni. Limitarile expresiilor lambda in limbajul Java sunt ca acestea pot sa arunce exceptii verificate, dar aceste expresii lambda nu vor functiona cu interfetele folosite de API-ul Collection. Totodata, variabilele declarate inafara functiei lambda nu vor [putea folosite in interiorul lor doar daca aceste sunt final, adica se garanteaza ca acestea sunt imutabile pentru a nu suferi modificari nedorite in expresie.

7. Bibliografie

Lamda Expressions: en.wikipedia.org/wiki/Anonymous_function

Functional Interfaces: https://www.baeldung.com/java-8-functional-interfaces

Functional Programming: http://tutorials.jenkov.com/java-functional-programming/index.html

File I/O: https://stackabuse.com/reading-and-writing-files-in-java/