

- Descriere fiecare laborator+Proiect

Laboratorul 1:

Laborator introductiv cu tipuri de date și ce se întâmplă în cazul în care dorim să realizăm anumite operații cu acestea (în principal în cazul adunării).

Dacă se adună două tipuri de date numerice, se face conversia la cel mai mare.

Exemplu: `int + double => double`.

În cazul stringurilor, când acestea sunt însumate cu orice, se face concatenare.

Dacă se face suma unui char cu un tip numeric, va rezulta suma dintre valoarea numerică a respectivului char și elementul numeric cu care se face suma.

Laboratorul 2:

În acest laborator am făcut funcții care sunt similare cu cele din C++ (singurul lucru diferit este că, dacă facem o funcție, trebuie să specificăm dacă este public/private/protected plus dacă aceasta este statică sau nu).

Tot în acest laborator am văzut și cum se citesc datele de la tastatură:

```
Scanner sc = new Scanner(System.in);
```

```
sc.close();
```

Aceste două linii de cod sunt elementele importante pentru a putea citi datele corect.

Pentru fiecare tip de date există câte un "scanner" diferit. De exemplu:

- `sc.nextInt();` - pentru întregi (inturi);
- `sc.nextFloat();` - pentru numere reale (floaturi);
- `sc.nextLine();` - pentru stringuri (și consumă toată linia).

Laboratorul 3:

În acest laborator am învățat cum să facem și să folosim clase.

Elementele sunt similare cu C++, deși există anumite diferențe între acestea (dacă vrem să avem ceva virtual, folosim abstract, care are aceleași proprietăți ca și virtual).

Pe lângă clase, am învățat să facem și moșteniri între clase (folosim extends, care este cuvântul de definire al moștenirii). Și tot în acest laborator am învățat cum să utilizăm List, Map, etc.

Laboratorul 4:

În acest laborator am învățat cum să introducem o interfață pentru o clasă. Pentru a implementa o interfață pentru o clasă, folosim după definirea clasei cuvântul implements,

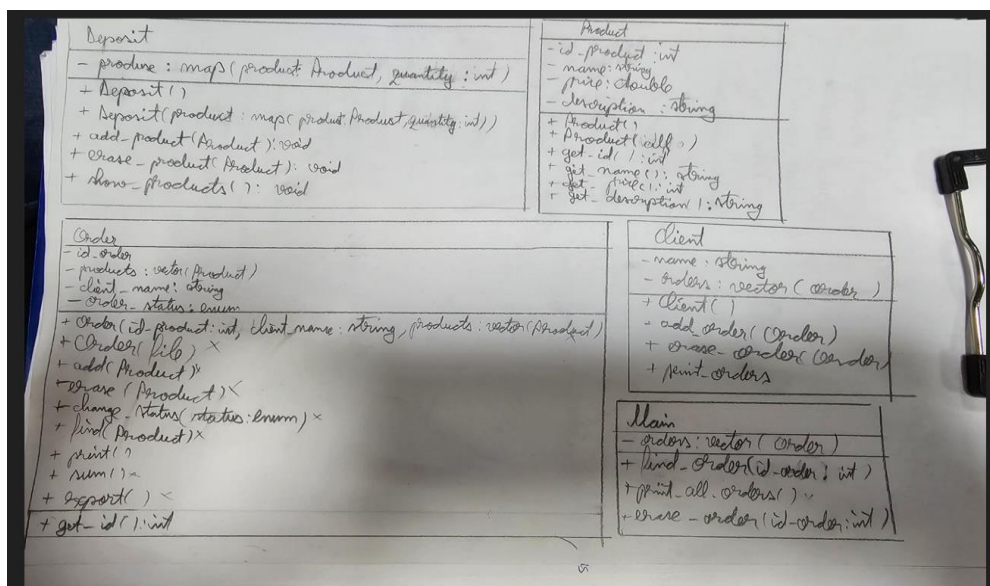
urmat de denumirea interfeței.

Am introdus interfețe pentru a:

1. Evita moștenirea multiplă (o clasă poate implementa mai multe interfețe, oferind flexibilitate în design);
2. Separarea implementării de definiție (promovează programarea orientată spre abstractizare și modularitate);
3. Polimorfism (permite folosirea unui tip comun pentru obiecte de clase diferite, facilitând procesarea uniformă).

Laboratorul 5:

În acest laborator am făcut aplicația **OrderNow**, în care am implementat multiple clase pentru a îndeplini taskurile cerute de aplicație. Tot în acest laborator ne-am exersat și abilitățile de brainstorming și teamwork, realizând acest mini-proiect în echipă de 4 studenți. De asemenea, am învățat cum să citesc date și să afișez date într-un fișier .txt cât și



Laboratorul 6:

În acest laborator am învățat cum să folosim fișiere JSON pentru a salva și a încărca date, deoarece acest tip de fișier este o alternativă a fișierelor .txt, având un aspect mult mai citet. Implementarea este una relativ simplă, bazată pe un anumit pattern.

Laboratorul 7:

În acest laborator am învățat cum să facem teste pe anumite părți din proiectele noastre.

Testele sunt utile atunci când vrem să verificăm dacă o anumită parte din codul nostru este validă și să vedem dacă, în urma testelor, aceasta poate sau nu să fie integrată și să ruleze corespunzător atunci când va fi integrată în codul final.

Laboratorul 8:

În acest laborator am învățat cum să folosim și să creăm clase template pentru a realiza un cod modular și ușor de utilizat pentru tipuri diferite de date.

Template-urile sunt utilizate pentru a evita declararea multiplă a aceleiași metode/clase.

Folosind template-uri, putem crea clase și metode care funcționează atât cu inturi, cât și cu tipuri de date mai complexe.

Structura proiectului și descrierea sa:

Ca proiect, am decis să creez o aplicație care ține evidența consultațiilor la un cabinet medical.

În acest proiect am introdus elemente din laboratoare pentru a avea o aplicație complexă și funcțională, care utilizează clase și interfețe ce rețin date. Aceste date sunt salvate în fișiere JSON. Datele salvate pot fi preluate din fișierul JSON pentru a fi afișate în interfața grafică a aplicației.

În acest proiect am următoarele clase de bază:

1. **Clasa abstractă Persoană:** reține numele și CNP-ul unei persoane.
2. **Clasa Pacient:** aici se rețin date despre istoricul medical al unui pacient.
3. **Clasa Medic:** reține specializarea medicului și zilele în care acesta lucrează.
4. **Clasa Consultație:** această clasă conține elementele din clasele Pacient și Medic, cât și data la care are loc consultația și starea acesteia (Programat, Completată, Anulată).

Pentru aceste clase a fost necesară implementarea a două interfețe care sunt utilizate de către clasele Pacient, Medic și Consultație. Aceste interfețe sunt:

- **Afișabil:** folosită pentru a afișa date despre cele trei clase.
- **Validabil:** verifică dacă datele introduse sunt valide sau nu (de exemplu, dacă CNP-ul este corect).

Pe lângă clase și interfețe, am creat și un enum care reține starea consultației.

Test

După ce am creat aceste clase, am realizat teste pentru a verifica corectitudinea metodelor implementate și am realizat ca toate acestea au trecut peste teste ceea ce confirmă implementarea bună a claselor cât și a metodelor membre. Testele sunt 5 la număr fiecare verificând corectitudinea elementelor din clase prezent în proiect.

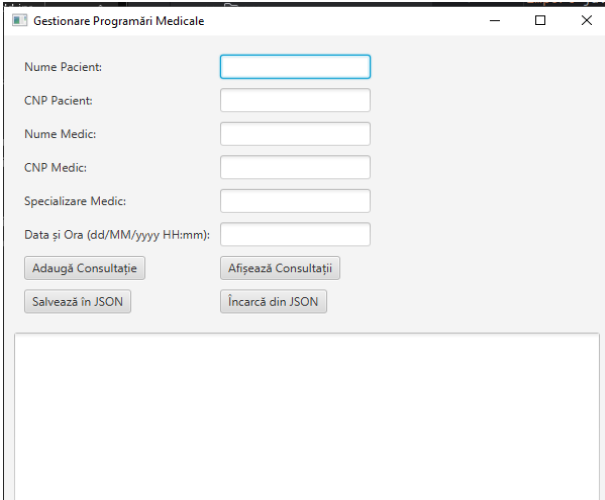
Clasa Principala Main

După acest test, am creat clasa **Main**, în care am introdus elemente atât pentru medici, cât și pentru pacienți și consultații, afișându-le ulterior în fișierul `programari.json`. După ce acestea sunt afișate în JSON, le încărcăm și le afișăm în consolă.

Interfata Grafică

La acest proiect am adăugat și o interfață simplă.

În interfață se adaugă datele în câmpuri predefinite.



Ce face fiecare buton:

1. Adaugă Consultație:

- Preia datele introduse în câmpuri (nume pacient, CNP pacient, nume medic, CNP medic, specializare medic, data și ora).
- Validează datele.
- Creează o consultație și o adaugă în lista programărilor dacă datele sunt valide.

2. Afișează Consultații:

- Afișează toate consultațiile existente în lista programărilor în zona de text (TextArea).

3. Salvează în JSON:

- Salvează toate consultațiile curente din lista programărilor într-un fișier JSON specificat (programari.json).

4. Încarcă din JSON:

- Încarcă consultațiile salvate anterior din fișierul JSON (programari.json) și le adaugă în lista programărilor curente.

Gestionările

În pachetul de gestionări avem două gestionări foarte importante:

1.Gestionarea Programărilor:

- Face adaugărea de programări în o listă.

```
package gestionari;

import baseClass.Consultatie;

import java.util.ArrayList;
import java.util.List;

public class GestionareProgramari { 4 usages
    private List<Consultatie> programari; 4 usages

    public GestionareProgramari() { 2 usages
        this.programari = new ArrayList<>();
    }

    public void adaugaProgramare(Consultatie programare) { programari.add(programare); }
    public void afisareProgramari() { programari.forEach(Consultatie::afisareDetalii); }
    public List<Consultatie> getProgramari() { return programari; }
}
```

2.Gestionare JSON:

Această gestionare permite exportul și importul datelor în/din format JSON.

Descrierea clasei:

1. Definirea unui obiect JSON:

```
private static final Gson gson = new GsonBuilder().setPrettyPrinting().create();
```

Funcția de export:

- Primește ca parametri:

- O cale către un fișier unde se va salva JSON-ul.
- O listă de programări care trebuie exportate.

Funcția convertește lista în format JSON folosind linia:

String json = gson.toJson(programari);- Aceasta preia datele din listă și le transformă în format JSON, pregătindu-le pentru salvare în fișier.

Funcția de import:

- Adaugă într-o listă de consultații elementele dintr-un fișier JSON.
- Datele din JSON sunt convertite automat în formatul specific al listei, asigurând compatibilitatea.

```
import java.util.List;

public class GestionareJSON {
    private static final Gson gson = new GsonBuilder()
        .setPrettyPrinting()
        .create();

    public static void exportaProgramari(String filePath, List<Consultatie> programari) {
        try (FileWriter writer = new FileWriter(filePath)) {
            String json = gson.toJson(programari);
            writer.write(json); // Scrie JSON-ul în fișier
            System.out.println("Programările au fost salvate în JSON!");
        } catch (IOException e) {
            System.out.println("Eroare la salvarea în JSON: " + e.getMessage());
        }
    }

    public static List<Consultatie> importaProgramari(String filePath) {
        try (FileReader reader = new FileReader(filePath)) {
            Consultatie[] programariArray = gson.fromJson(reader, Consultatie[].class);
            return new ArrayList<>(List.of(programariArray));
        } catch (IOException e) {
            System.out.println("Eroare la încărcarea din JSON: " + e.getMessage());
        }
        return new ArrayList<>();
    }
}
```

Ce am folosit din fiecare laborator

Din **laboratorul 1** am folosit tipurile de date precum string pentru majoritatea elementelor din proiect. Elemente precum CNP, nume, specializare.

Din **laboratorul 2** și **laboratorul 3** am folosit clase care sunt moștenite pentru a putea avea o implementare corectă. Din laboratorul 2 nu am folosit foarte multe elemente deoarece nu citesc date de la tastatură, ci le primesc prin UI și le transfer mai departe către JSON-uri. Totuși, la nivel de utilizare al funcțiilor, întreg codul este bazat pe funcții care permit implementarea corectă,

atât prin cele necesare claselor, cât și prin funcțiile auxiliare de verificare/validare a datelor. Când vine vorba de moșteniri, clasa de bază este **Persoană**, care este moștenită de către **Pacient** și **Medic**.

Din **laboratorul 4** am utilizat implementarea de interfețe, **Afisabil** și **Validabil**, care sunt moștenite în **Consultație**, **Medic** și **Pacient** pentru a verifica dacă datele introduse sunt valide și, dacă da, acestea sunt afișabile.

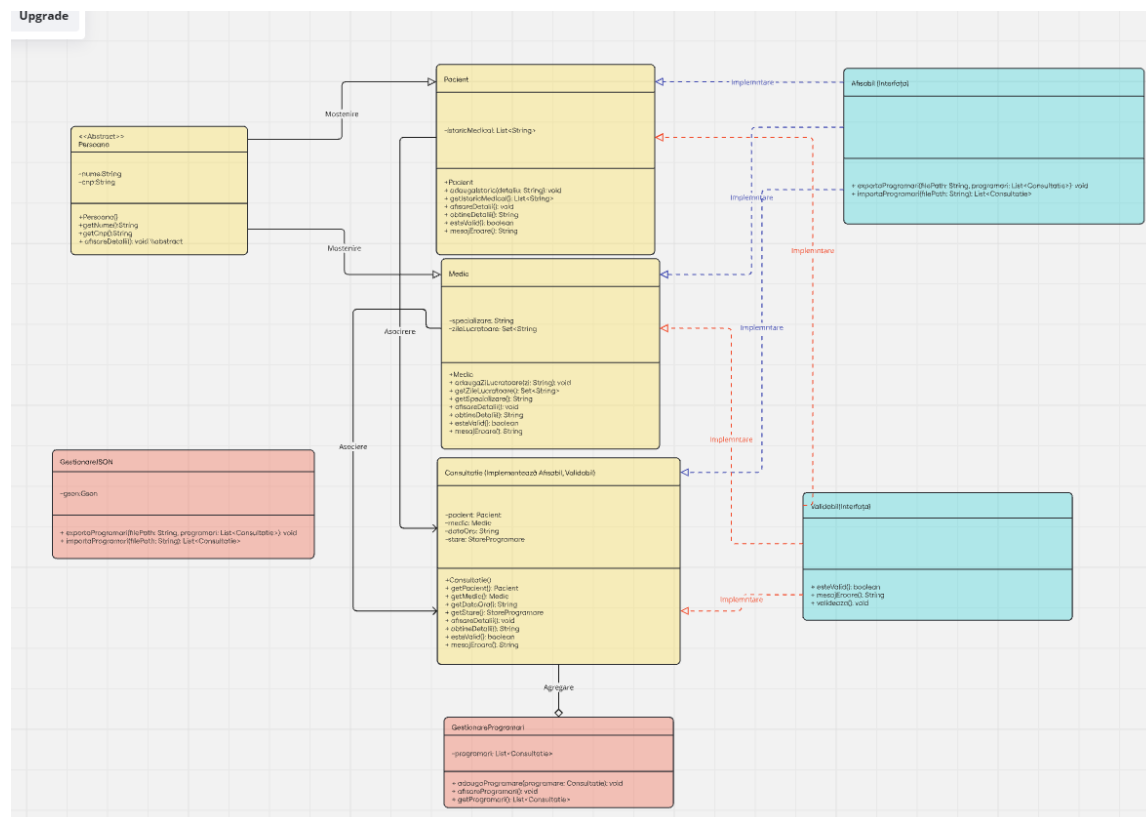
Din **laboratorul 6** am utilizat salvarea și importarea datelor dintr-un JSON în clasa **GestionareJson**, în care verific dacă datele au fost salvate în JSON și, dacă nu, afișez un mesaj corespunzător.

În acest proiect, testele sunt realizate pentru a confirma corectitudinea fiecărei metode(**laboratorul 7**).

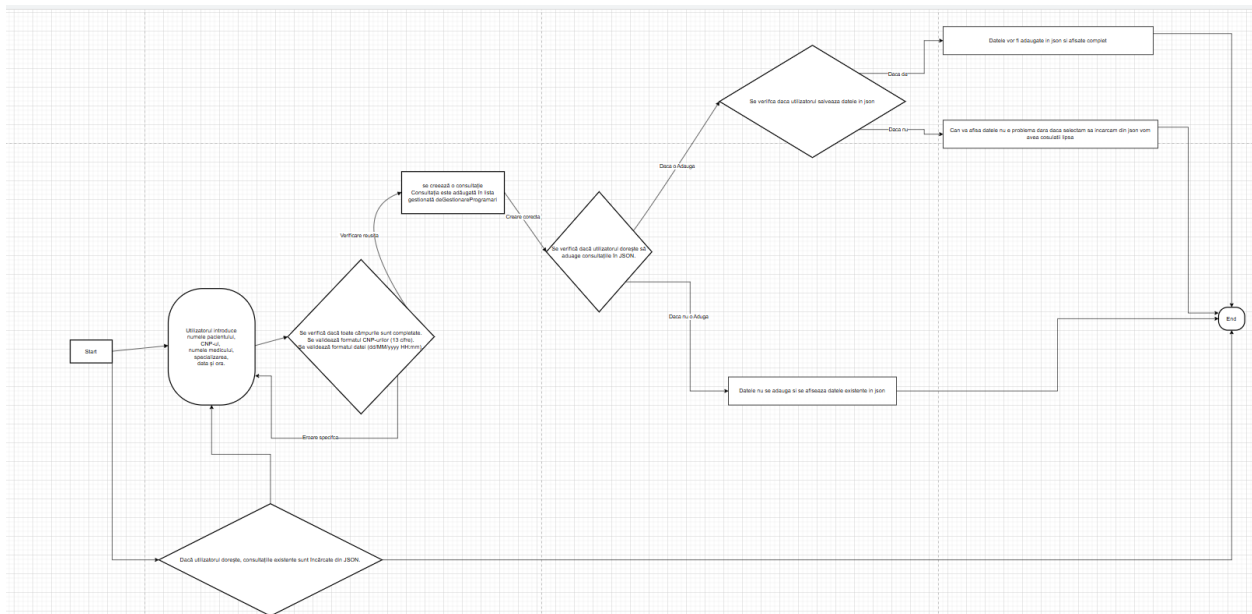
Din **laboratorul 8** nu am utilizat template-uri, însă aş putea să le folosesc în **Pacient** și **Medic** pentru gestionarea entităților (să creez o clasă care gestionează adăugarea, ștergerea și afișarea entităților, evitând astfel funcțiile multiple de afișare pentru clasele membre).

Cele 3 diagrame UML

Class :



Flow:



Use Case:

