Universitatea Tehnică de Construcții București
Facultatea de Hidrotehnica

PCLP

Profesor Coordonator:

Sef lucrari dr. ing. Ramona Oana Flangea

Student:

Aruxandei Ștefan

Universitatea Tehnică de Construcții București

Facultatea de Hidrotehnica

Aplicatie Management Stoc

Cuprins

Introducere

Acest document oferă o descriere completă a unui mic sistem de gestionare a stocurilor implementat în Java. Scopul este de a oferi o documentație tehnică și funcțională pentru dezvoltatori, administratori și utilizatori care doresc să înțeleagă, să ruleze și să extindă aplicația.

Prezentare generală a proiectului

Proiectul este o aplicație de consolă Java care permite următoarele operații de bază asupra unui inventar de produse:

- Vizualizare inventar complet
- Adăugare produs nou
- Actualizare nivel stoc
- Căutare și filtrare după nume sau categorie
- Afișare statistici simple (valoare totală, alerte stoc)

Aplicația folosește o listă în memorie (ArrayList) pentru a stoca obiecte de tip **Produs**. Interfața este bazată pe linie de comandă (CLI), ușor de folosit pentru scenarii mici sau pentru testare rapidă.

Istoric succint al managementului stocurilor

Managementul stocurilor (inventory management) este o practică esențială în logistică și retail, având o istorie ce se întinde de la metode manuale de evidență (registre, cardex) până la sisteme informatizate moderne (ERP, WMS). Pe scurt:

- Epoca manuală: înregistrări pe hârtie; dificilă actualizarea în timp real.
- Automatizare timpurie: calculatoare mainframe și baze de date relaționale; rapoarte periodice.
- Sisteme moderne: integrări ERP, coduri de bare, RFID, urmărire în timp real și analitică avansată.

Acest proiect reprezintă o implementare simplificată, utilă pentru înțelegerea conceptelor fundamentale: SKU, cantitate, preț, categorie, alerte pentru stoc redus și calcul al valorii inventarului.

Arhitectura și componentele aplicației

Aplicația are o arhitectură monolitică și este compusă din elemente simple:

- Clasa principală (file) - conține metoda main și logica de interacțiune cu utilizatorul.
- Clasa internă statică Produs - modelul de date pentru un produs.
- Colecția inventory - un ArrayList care stochează instanțele Produs.
- Metode statice pentru operații: initializareDate, afisareAntet, afisareInventar, adaugareProdus, actualizareStoc, meniuCautare, afisareStatistici.

Structura codului și fragmente explicative

Mai jos sunt extrase și explicate părți cheie din codul sursă.

1. Declarații și inițializări

```
private static final List<Produs> inventory = new ArrayList<>();
private static final Scanner scanner = new Scanner(System.in);
```

Explicație: inventory stochează toate produsele; scanner citește inputul utilizatorului din linia de comandă.

2. Metoda main (flux principal)

```
public static void main(String[] args) {
  initializareDate();
  boolean running = true;
  while (running) {
    afisareAntet();
    // Meniu afișat și citire opțiune
    String input = scanner.nextLine();
    switch (input) {
      case "1" -> afisareInventar(inventory);
      case "2" -> adaugareProdus();
      case "3" -> actualizareStoc();
      case "4" -> meniuCautare();
      case "5" -> afisareStatistici();
      case "6" -> running = false;
      default -> System.out.println("Optiune invalida.");
    }
  }
}
```

Explicație: metoda main inițializează datele, apoi rulează un loop principal în care afișează un meniu și răspunde la comenzi.

3. Modelul Produs

```
static class Produs{
  int id;
  String name;
  int quantity;
  double price;
  String category;
  public Produs(int id, String name, int quantity, double price, String category) {
    this.id = id; this.name = name; this.quantity = quantity; this.price = price; this.category = category;
  }
}
```

Explicație: clasa simplă POJO (Plain Old Java Object) pentru a reprezenta un produs din inventar.

4. Adăugare produs (validare minimală)

```
private static void adaugareProdus() {
  try {
    System.out.print("Introduceti numele produsului: ");
    String name = scanner.nextLine();
    System.out.print("Introduceti categoria: ");
    String category = scanner.nextLine();
    int qty = Integer.parseInt(scanner.nextLine());
    double price = Double.parseDouble(scanner.nextLine());
    int newId = inventory.stream().mapToInt(p -> p.id).max().orElse(100) + 1;
    inventory.add(new Produs(newId, name, qty, price, category));
  } catch (Exception e) {
    System.out.println("Eroare: Introducere numerica invalida.");
  }
}
```

Explicație: metoda citește câmpurile produsului, convertește valorile numerice și creează un ID nou incremental. Tratarea erorilor este simplă, prin catch generic.

5. Actualizare stoc

```
private static void actualizareStoc(){
  System.out.print("Introduceti ID-ul produsului pentru actualizare: ");
  try{
    int id = Integer.parseInt(scanner.nextLine());
    Optional<Produs> prod = inventory.stream().filter(p -> p.id == id).findFirst();
    if (prod.isPresent()) {
      int change = Integer.parseInt(scanner.nextLine());
      int newQty = prod.get().quantity + change;
      if (newQty < 0) { System.out.println("Eroare: Nu puteti avea stoc negativ."); } else { prod.get().quantity =
newQty; }
    } else { System.out.println("ID-ul produsului nu a fost gasit."); }
  } catch(Exception e) { System.out.println("Input invalid."); }
}
```

Explicație: permite creșterea sau scăderea cantității; blochează stocul negativ.

6.Căutare și filtrare

```
private static void meniuCautare() {
  System.out.print("Cautare dupa (1) Nume sau (2) Categorie: ");
  String choice = scanner.nextLine();
  System.out.print("Introduceti cuvantul cheie: ");
  String keyword = scanner.nextLine().toLowerCase();
  List<Produs> results = inventory.stream().filter(p -> choice.equals("1") ?
p.name.toLowerCase().contains(keyword) :
p.category.toLowerCase().contains(keyword)).collect(Collectors.toList());
  afisareInventar(results);
}
```

Explicație: filtrare simplă folosind stream API în funcție de opțiunea utilizatorului.

7. Statistici simple

```
private static void afisareStatistici() {
  double totalValue = inventory.stream().mapToDouble(p -> p.price * p.quantity).sum();
  long lowStock = inventory.stream().filter(p -> p.quantity < 10).count();
  System.out.printf("Numar total de produse (SKU): %d\n", inventory.size());
  System.out.printf("Valoarea totala a inventarului: %.2f RON\n", totalValue);
  System.out.printf("Alerte stoc critic (<10): %d produse\n", lowStock);
}
```

Explicație: calculează valoarea totală a inventarului și numărul de produse cu stoc redus.

Instrucțiuni de instalare și rulare

Pentru a rula proiectul local:

1. Asigurați-vă că aveți instalat JDK 11+ (sau 8+ pentru compatibilitate similară).
2. Creați fișierul sursă file.java (sau redenumiți în InventoryApp.java pentru consistență cu nume de clasă publice).
3. Compilați:

   ```
   javac file.java
   ```

4. Rulați:

   ```
   java file
   ```

Observații: dacă redenumiți clasa publică, asigurați-vă că numele fișierului corespunde numelui clasei publice. De asemenea, aplicația folosește intrare standard pentru interacțiune, deci rulați dintr-un terminal.

Ghid de utilizare (funcționalități principale)

Meniul principal oferă următoarele opțiuni:

- [1] Vizualizare Inventar Complet - Afișează toate produsele cu format tabelar.
- [2] Adăugare Produs Nou - Solicita nume, categorie, cantitate și preț; generează un ID nou.
- [3] Actualizare Nivel Stoc - Permite creșterea/scăderea cantității pentru un ID existent.
- [4] Cautare si Filtrare - Caută după nume sau categorie (case-insensitive).
- [5] Panou Statistici (Analiza) - Arată valoarea totală a inventarului și alertele de stoc.
- [6] Iesire Sistem - Oprește aplicația.

Exemple de scenarii de utilizare

1) Adăugare produs nou:

Introduceti numele produsului: Mouse Wireless
Introduceti categoria: Electronice
Introduceti cantitatea initiala: 30
Introduceti pretul unitar: 49.99

2) Actualizare stoc:

Introduceti ID-ul produsului pentru actualizare: 104
Introduceti modificarea (ex: 10 pentru adaugare, -5 pentru eliminare): -3

Analiză, limitări și îmbunătățiri posibile

Limitări actuale:

- Stocul este gestionat doar în memorie; datele nu sunt persistate între rulări.
- Validarea inputului este minimă; formatul prețurilor și numere negative pot cauza comportamente nedorite dacă nu sunt tratate atent.
- Interfața este text-based — nu este prietenoasă pentru utilizatorii non-tehnici comparativ cu GUI/web.

Posibile îmbunătățiri:

- Persistență: adăugați salvare/încărcare din fișier (CSV, JSON) sau integrare cu o bază de date (SQLite, PostgreSQL).
- Validare avansată: verificare intrări, formate monetare, reguli pentru categorii, restricții de cantitate.

- Interfață: dezvoltarea unui GUI (Swing/JavaFX) sau API REST cu interfață web (Spring Boot + frontend).
- Autentificare și roluri: acces controlat pentru anumite acțiuni (numai admin poate șterge produse etc.).
- Raportare: generare de rapoarte PDF/CSV, export date, grafice pentru analiză.

Design propus pentru versiuni viitoare

O arhitectură recomandată pentru extindere:

- Strat prezentare: API REST sau UI web.
- Strat serviciu: logică business pentru operatii pe produse, validări, tranzacții.
- Strat persistenta: repository/DAO pentru acces la baza de date.

Această separare ar: facilita testarea unitară, permițând în același timp scalarea și integrarea altor componente.

Plan de testare

Câteva teste de bază care ar trebui implementate:

- Unit tests pentru adăugare produs: ID unic, cantitate corectă și preț.
- Test actualizare stoc: creștere, scădere, prevenire stoc negativ.
- Test căutare: match corect pe nume și categorie (inclusiv cazuri insensibile la majuscule).
- Test interfață: simulare input invalid (caractere non-numerice la cantitate/preț).

Aspecte de securitate și bune practici

Chiar dacă este o aplicație simplă, recomandări generale:

- Validare input: nu se acceptă input nevalid sau injecții în cazul în care se adaugă persistenta SQL.
- Controlul erorilor: evitați blocuri catch generice, logați erorile și oferiți feedback util utilizatorului.
- Separare responsabilități: evitați clase monolitice; folosiți clase dedicate pentru I/O, logică business și model.

Fragment extins din cod (fișier completat pentru referință)

```
import java.util.*;
import java.util.stream.Collectors;

public class file{
  private static final List<Produs> inventory = new ArrayList<>();
  private static final Scanner scanner = new Scanner(System.in);
  // ... restul codului conform exemplului inițial ...
}
```

Posibile structuri de date alternative

Pentru performanță sau acces rapid pe ID se poate folosi un Map<Integer, Produs> (cheie = ID). Avantaje:

- Acces O(1) la produs după ID.
- Eliminarea necesității de stream().filter pentru anumite operații.

Tabel de comparare (implementări)

Implementare	Avantaje	Dezavantaje
ArrayList<Produs>	Simplu, iterabil ușor	Căutare O(n) pentru ID
HashMap<Integer,Produs>	Acces direct pe ID	Mai multă memorie, complexitate la iterare ordonată
Bază de date	Persistență, scalabilitate	Suprastructură, necesită setup

Concluzii

Acest proiect oferă o bază funcțională pentru gestionarea stocurilor la nivel prototype. Este potrivit pentru învățare, demo-uri sau aplicații foarte mici. Pentru producție sunt necesare extensii privind persistenta, validarea, securitatea și interfața utilizator.

Referințe și resurse suplimentare

- Wikipedia - Inventory management: https://en.wikipedia.org/wiki/Inventory_management
- Wikipedia — Just-in-time manufacturing: https://en.wikipedia.org/wiki/Just-in-time_manufacturing
- Oracle — Java Collections Framework: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/
- Baeldung — Guide to Java Streams: https://www.baeldung.com/java-8-streams
- Baeldung — Guide to Java Streams: https://www.baeldung.com/java-8-streams
- SAP — What is an ERP system: https://www.sap.com/products/what-is-erp.html
- NIST — Procurement and inventory standards și bune practice: https://www.nist.gov/

- SQLite — Lightweight SQL database: https://www.sqlite.org/index.html
- Spring — Building REST APIs with Spring Boot: https://spring.io/guides/gs/rest-service