PERSONAL TRANSACTIONS MANAGER

BY: Aditya Raj Sharma
(24BCE10497)

NOVEMBER 24, 2025

Project Report: Personal Finance Tracker

Course: VITYARTHI Java Course

Student Name: Aditya Raj Sharma (24BCE10497)

Date: 24/11/ 2025

1. Introduction

The **Simple Personal Finance Tracker** is a console-based software application developed using Java. In an era where financial literacy and budget management are crucial, this tool provides a lightweight, offline solution for individuals to track their daily income and expenses. Unlike complex banking apps that require internet access and account linking, this application focuses on privacy, simplicity, and immediate feedback, allowing users to maintain a digital ledger of their finances on their local machine.

2. Problem Statement

Managing personal finances manually using physical notebooks or spreadsheets is often tedious, error-prone, and lacks immediate insight into spending habits. Individuals frequently struggle to keep an accurate running total of their net balance or lose track of specific expenditures over time.

There is a need for a simple, automated tool that eliminates the manual arithmetic of budgeting and provides a reliable, persistent record of financial activity without requiring complex accounting knowledge, subscription fees, or internet connectivity.

3. Functional Requirements

The system is designed to fulfill the following core functional requirements:

1. **Transaction Logging:**
    - The system shall allow users to input **Income** entries with amount, category, and description.
    - The system shall allow users to input **Expense** entries with amount, category, and description.
    - The system shall automatically capture the date of the transaction.
2. **Data Viewing:**
    - The system shall display a sequential list of all recorded transactions with their IDs, dates, types, and amounts.

3. **Financial Analysis:**
    - The system shall calculate and display the **Total Income**.
    - The system shall calculate and display the **Total Expenses**.
    - The system shall compute the **Net Balance** (Income - Expenses) in real-time.
4. **Data Management:**
    - The system shall allow users to delete a specific transaction using its unique ID.
    - The system shall ensure that deleting a transaction immediately updates the net balance.
5. **Persistence:**
    - The system shall save all data to a local file (transactions.ser) automatically upon any change.
    - The system shall load existing data automatically when the application starts.

4. Non-functional Requirements

- **Performance:** The application must handle list operations (add/delete/view) instantly (O(1) or O(n) complexity) without noticeable lag for a history of up to 10,000 transactions.
- **Security:** Data must be stored locally on the user's machine using binary serialization, ensuring that financial records are not exposed in plain text format to casual observers.
- **Usability:** The User Interface (UI) must be menu-driven and resilient to invalid inputs (e.g., entering text when a number is expected) without crashing.
- **Reliability:** The system must ensure data integrity is maintained across application restarts. Unique IDs for transactions must not overlap even after reloading data.

5. System Architecture

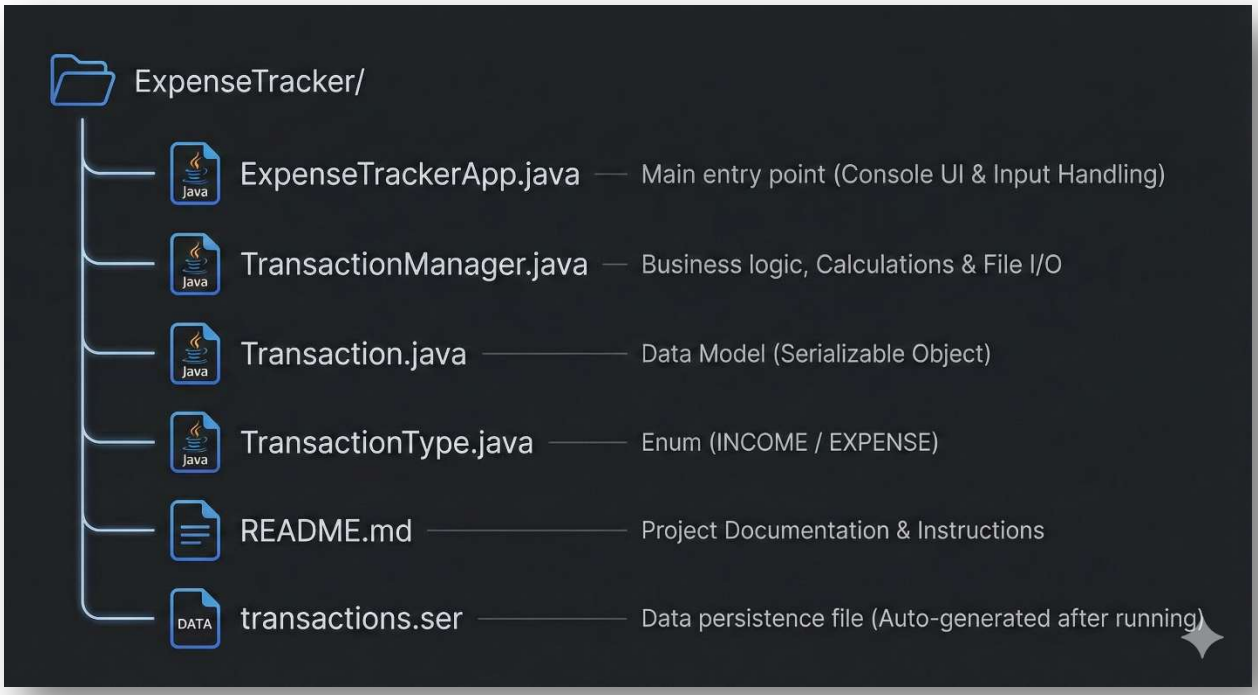
The application utilizes a modular architecture that separates the user interface from the business logic and data storage.

Layers:

1. **Presentation Layer:** Handles user input and displays data to the console.
2. **Business Logic Layer:** Manages the list of transactions and performs calculations.

3. **Data Layer:** Handles the serialization of objects to the file system.

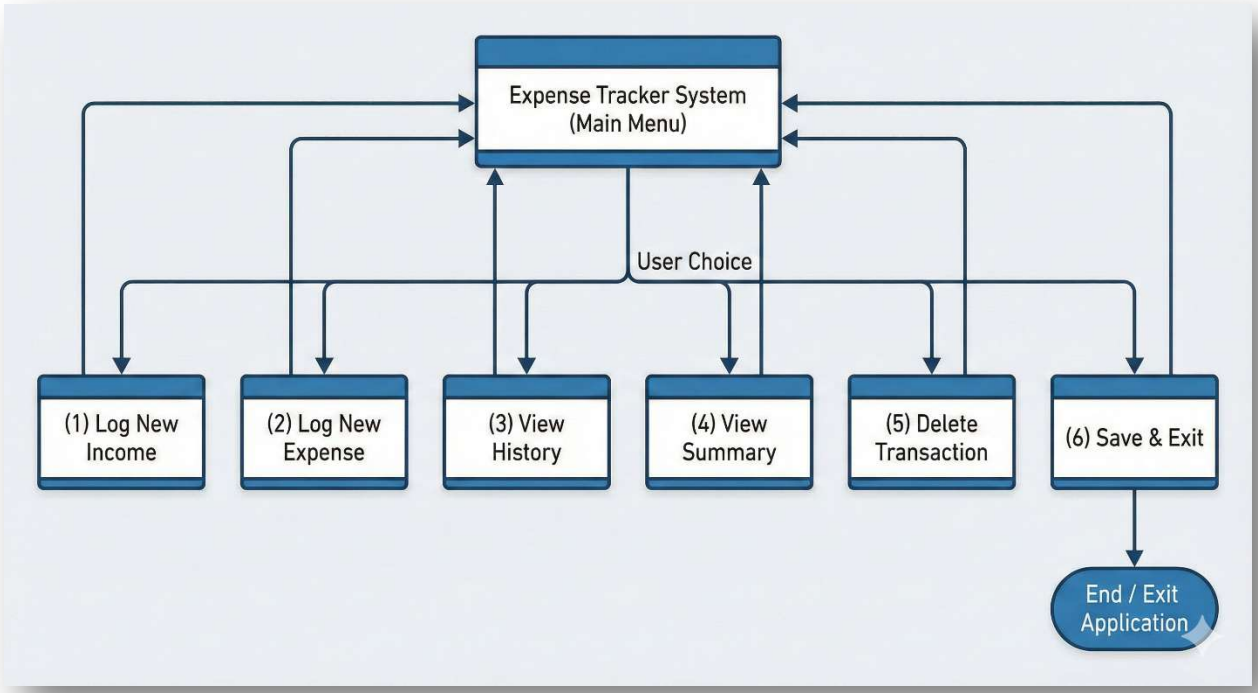
6. System Architecture Diagram



7. Design Diagrams

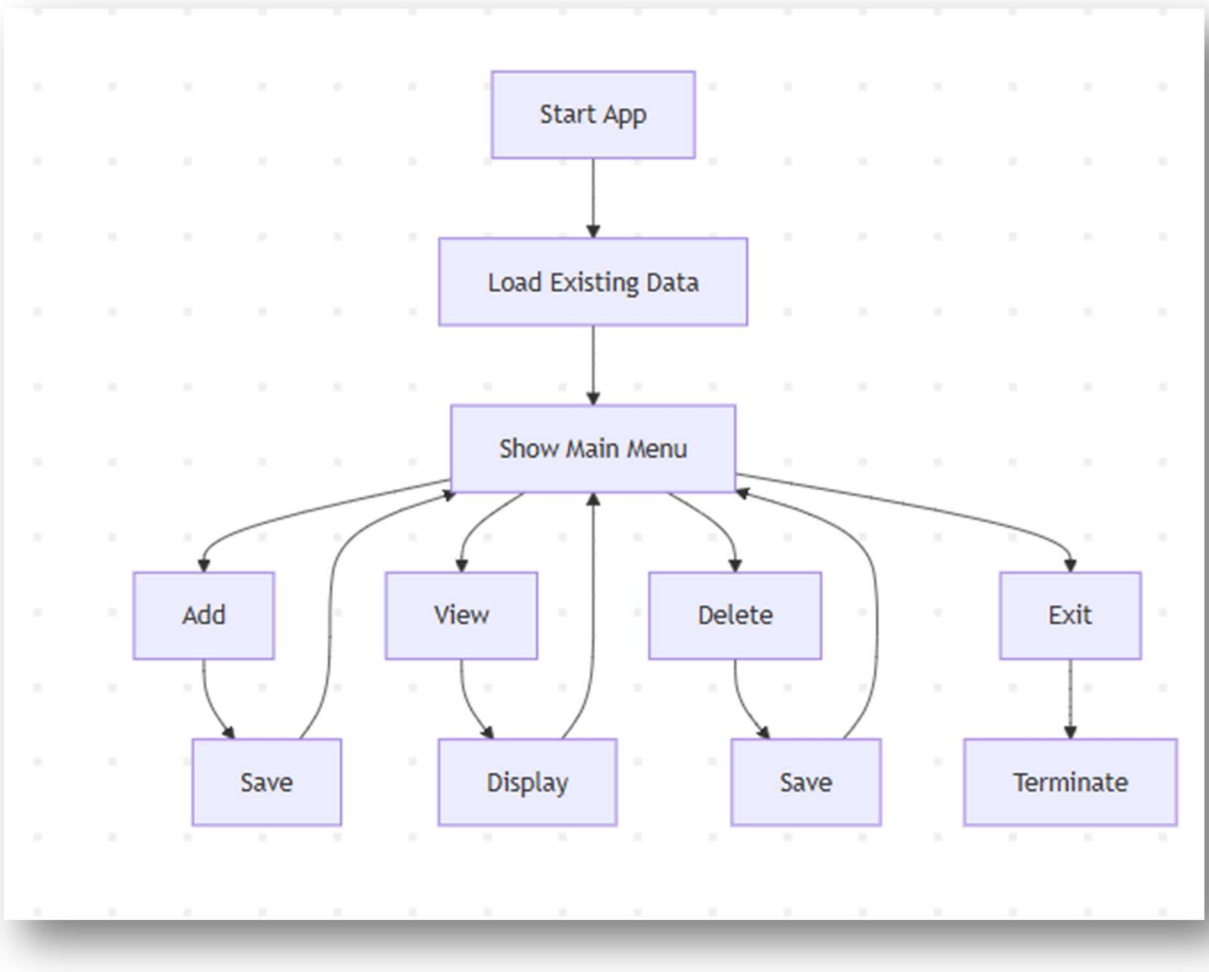
7.1 Use Case Diagram

This diagram illustrates the primary interactions a user has with the system.



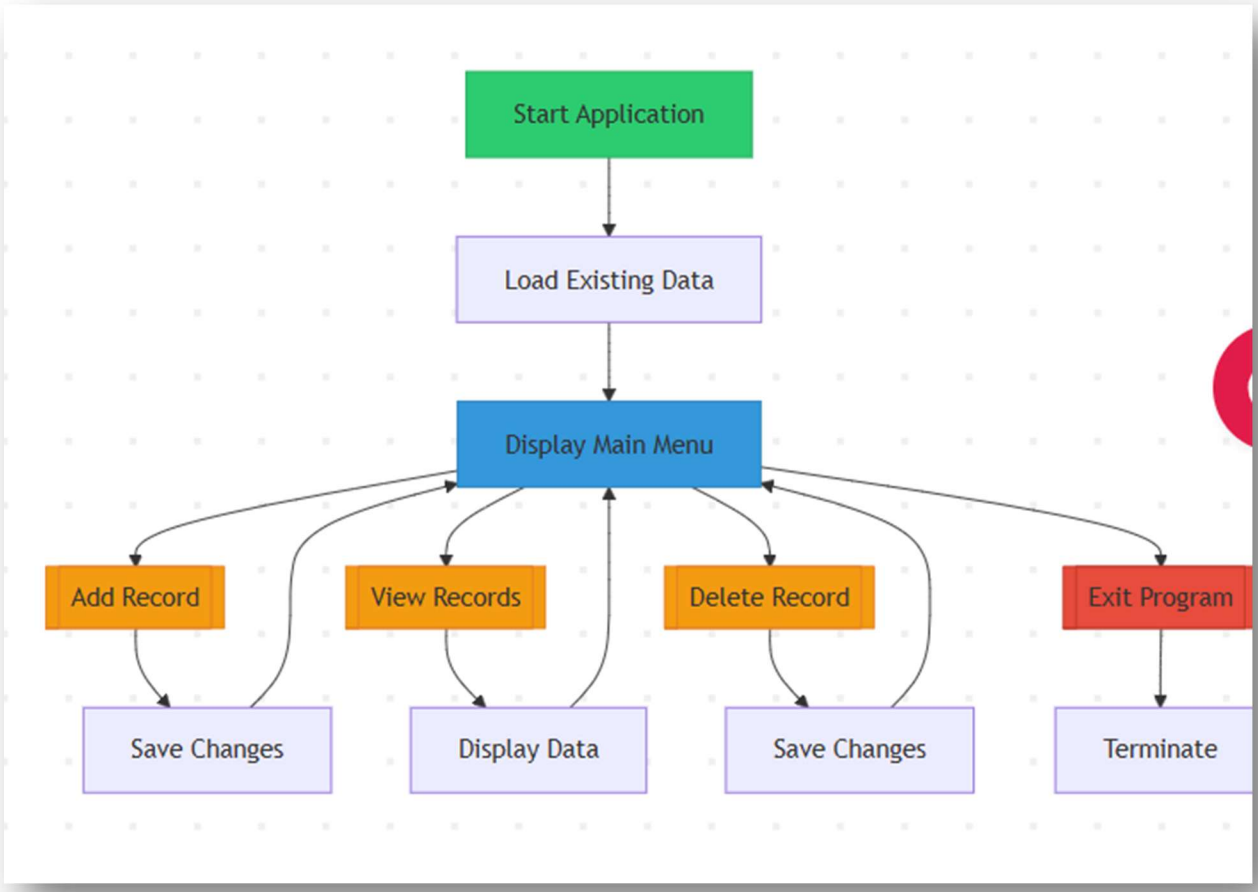
7.3 Sequence Diagram (Add Transaction)

Sequence of events when a user logs a new expense.



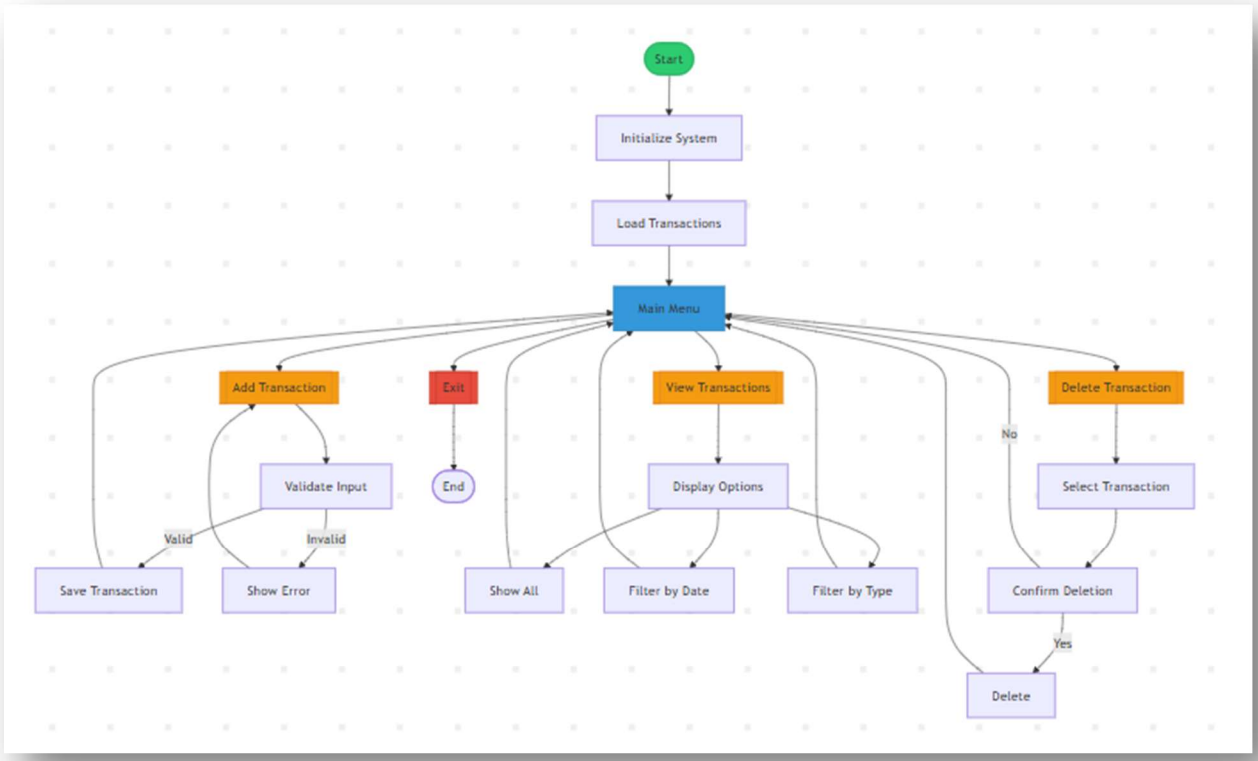
7.4 Class Diagram

The structural design of the Java classes.



7.5 Data Schema (ER Diagram Alternative)

Since the system uses Object Serialization rather than a relational database, this schema represents the stored object structure.



- **Entity:** Transaction
    - id (Integer, PK): Unique Identifier
    - type (Enum): INCOME or EXPENSE
    - amount (Double): The monetary value
    - category (String): E.g., "Food", "Rent"
    - description (String): User notes
    - date (LocalDate): Date of creation

8. Design Decisions & Rationale

1. **Java Serialization vs. Database:**
    - *Decision:* Used java.io.Serializable to store data in a .ser file.
    - *Rationale:* For a single-user "Personal" tracker, setting up a SQL database (MySQL/PostgreSQL) adds unnecessary complexity and configuration overhead. Serialization provides a zero-configuration, portable persistence mechanism perfect for this project scope.

2. **Console Interface (CLI):**
    - *Decision:* Built a text-based menu system.
    - *Rationale:* Focuses the development effort on solid logic and OOP principles (Backend/Logic) rather than UI frameworks (JavaFX/Swing), ensuring the core functionality is robust and bug-free.
3. **Modular Logic (TransactionManager):**
    - *Decision:* Separated business logic into a Manager class instead of keeping it in the Main class.
    - *Rationale:* adhere to the **Single Responsibility Principle**. The Main class handles User Input, while the Manager handles Data Processing. This makes testing easier.

9. Implementation Details

The project was implemented using **Java (JDK 17)**. Key implementation highlights include:

- **Persistence Mechanism:** The TransactionManager uses ObjectOutputStream to write the entire ArrayList<Transaction> to disk in one go. This simplifies the code significantly compared to writing line-by-line text files.
- **ID Generation:** A static idCounter in the Transaction class ensures unique IDs. Crucially, after loading old data, the TransactionManager scans the list to find the highest existing ID and updates the counter to prevent ID conflicts.
- **Dates:** Usage of java.time.LocalDate ensures modern, accurate date handling compared to the legacy java.util.Date.

Snippet: Loading Data

```
private Optional<List<Transaction>> loadTransactions() {
   try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(DATA_FILE))) {
      return Optional.of((List<Transaction>) ois.readObject());
   } catch (Exception e) {
      return Optional.empty();
   }
}
```

10. Screenshots / Results

(Note: Please insert actual screenshots of your running console here)

Figure 1: Main Menu

[Insert screenshot of the main menu showing options 1-6]

Figure 2: Adding an Expense

[Insert screenshot of user entering an expense: Amount 150, Category Food]

Figure 3: Financial Summary

[Insert screenshot showing Total Income, Total Expense, and Net Balance]

11. Testing Approach

The application underwent manual functional testing:

Test Case ID	Description	Input	Expected Output	Status
TC-01	Log Valid Income	5000, Salary	Success msg, ID generated	✅ Pass
TC-02	Log Valid Expense	100, Food	Success msg, ID generated	✅ Pass
TC-03	Invalid Amount	"abc"	Error: "Invalid input"	✅ Pass
TC-04	Negative Amount	-500	Error: "Must be positive"	✅ Pass
TC-05	Persistence	Restart App	Previous data visible	✅ Pass
TC-06	Delete Item	ID 1	Item removed from list	✅ Pass

12. Challenges Faced

1. **ID Synchronization:** Initially, restarting the app reset the ID counter to 1, causing new transactions to have duplicate IDs as old ones.
    - *Solution:* Implemented a updateIdCounterAfterLoad() method to scan loaded data and set the counter correctly on startup.
2. **Input Validation:** The Scanner class sometimes skipped lines (nextLine() issue) after reading numbers.
    - *Solution:* Added explicit Double.parseDouble(scanner.nextLine()) instead of mixing nextDouble() and nextLine().

13. Learnings & Key Takeaways

- **OOP Pillars:** Deepened understanding of **Encapsulation** (private fields in Transaction) and **Abstraction** (separating UI from Logic).
- **Serialization:** Learned how Java objects can be flattened to bytes and restored, which is fundamental to understanding how data travels across networks or saves to disks.
- **Modular Design:** realized that breaking code into Manager, Model, and App files makes debugging significantly faster.

14. Future Enhancements

- **Search & Filter:** Add functionality to filter transactions by Date Range or Category.
- **Budget Limits:** Allow users to set a monthly limit (e.g., ₹10,000) and warn them if they exceed it.
- **Export to CSV:** Add a feature to export the data to a .csv file so it can be opened in Microsoft Excel.

15. References

1. Oracle Java Documentation (Java IO, Java Time): *https://docs.oracle.com/en/java/*
2. Course Material: "Object Oriented Programming with Java" - Module 3 (File Handling).
3. Mermaid JS Documentation for Diagrams: *https://mermaid.js.org/*