Java Console-Based Banking System

Status Completed

Owners Saurabh Sonawane

Overview

A simple console based banking application that allows users to **create accounts, deposit, withdraw, check balance, and transfer money** between accounts.

Project Design & Features

Use Case :

- A user can create **Savings** or **Current** accounts.
- Perform basic banking operations.
- View transaction history.

Features & Functionalities

Feature	Description
User Authentication	Users can log in using their account number.
Account Management	Create **Savings** or **Current** accounts.
Deposit Money	Add money to the account balance.
Withdraw Money	Withdraw money with overdraft protection.
Fund Transfer	Transfer money between accounts.
Transaction History	Maintain and display transaction logs.

A. Core Functionalities

Your **Banking Application** should have the following features:

1. **User Management**
    - Create a new account (Savings, Current)
    - Login and authentication
    - View account details
2. **Banking Operations**
    - Deposit money
    - Withdraw money (with overdraft check for Current Account)
    - Transfer funds between accounts
    - Check balance
3. **Transaction History**
    - Maintain transaction logs
    - Show mini-statement
4. **Multi-threading (Optional but impressive)**

    - Handle multiple transactions simultaneously

5. **Exception Handling**
    - Handle insufficient balance
    - Handle invalid inputs

6. **File Handling (Optional)**
    - Store transaction history in a file

7. **JDBC (Optional for Database Integration)**
    - Store user details and transaction logs in MySQL/PostgreSQL

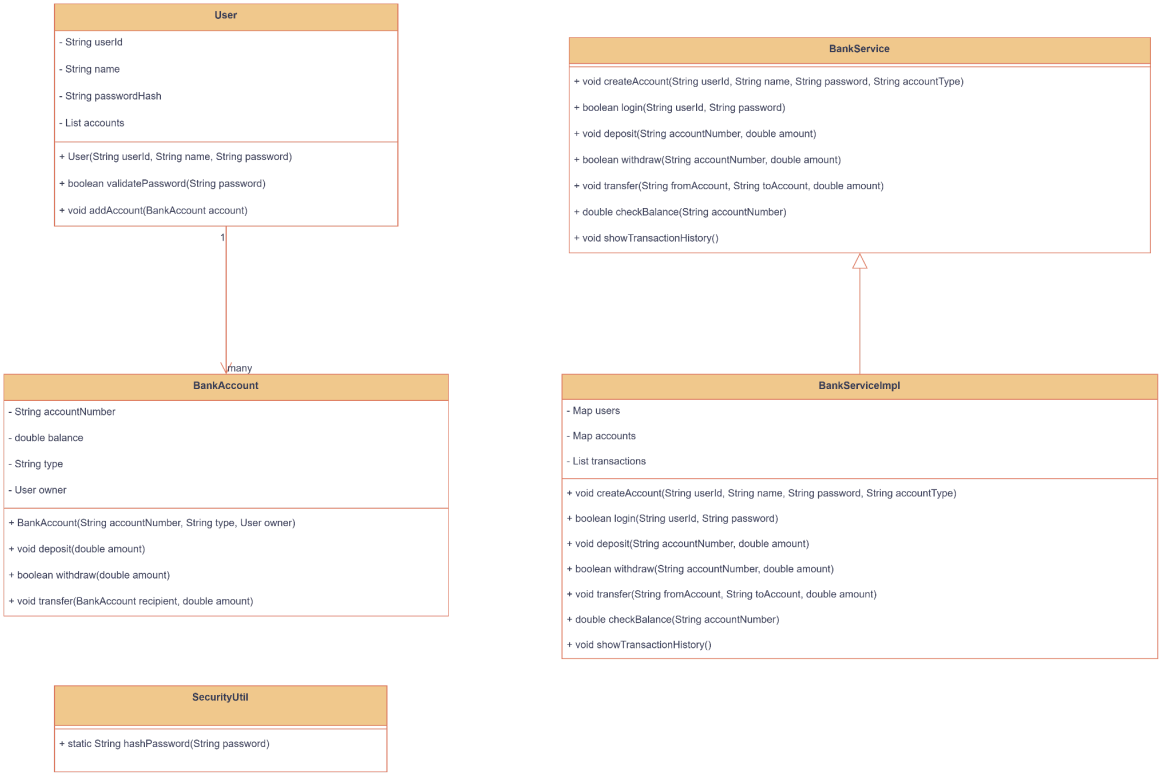
B.Technologies Used

- **Programming Language:** Java 17+
- **Core Java Concepts:** OOP (Encapsulation, Inheritance, Polymorphism, Abstraction)
- **Collections:** HashMap, ArrayList
- **Exception Handling:** Custom exceptions
- **Concurrency (Optional):** Multi-threading
- **Data Storage:** File Handling or JDBC (MySQL)

System Architecture

Code Design & Explanation

Class Diagram (UML)



Key Classes & Responsibilities

Class	Responsibility
`Account` (Abstract)	Base class with deposit, withdraw, and display methods.
`SavingsAccount`	Subclass of `Account` with interest calculation.
`CurrentAccount`	Subclass of `Account` with overdraft limit.
`Bank`	Manages all accounts (HashMap).
`BankingApplication`	Main class handling user input and operations.

Implementation Steps

1. **Design the Abstract `Account` Class**
    - Common fields: `accountNumber`, `balance`, `accountHolderName`.
    - Abstract methods: `deposit()`, `withdraw()`, `displayAccountDetails()`.

2. **Create `SavingsAccount` & `CurrentAccount` Classes**
    - Implement `withdraw()` differently (overdraft check in CurrentAccount).

3. **Build `Bank` Class (Account Manager)**
    - Use `HashMap<String, Account>` to store accounts.

4. **Implement `BankingApplication` (Main Class)**
    - Use `Scanner` for user input.
    - Provide a menu-driven interface.

Error Handling & Edge Cases

Scenario	Solution
Invalid account number	Show error message and prompt again.
Insufficient funds	Throw a custom exception (`InsufficientBalanceException`).
Invalid input (non-numeric)	Handle `InputMismatchException` gracefully.

Enhancements & Scalability

Enhancement	Description
Database Integration	Store user and transaction data in **MySQL** using JDBC.
REST API	Convert to a **Spring Boot REST API** for a web-based solution.
Multi-threading	Use `ExecutorService` for handling multiple transactions.
Authentication	Implement **username/password-based authentication**.

Testing Plan

Test Case	Expected Result
Create Account	Account successfully created.
Withdraw more than balance	"Insufficient funds" message.
Transfer between accounts	Balance deducted from sender and credited to receiver.

How to Run the Project

Prerequisites

- Java installed (`java -version`)
- IDE: IntelliJ IDEA / Eclipse

Compile using : `javac BankingApplication.java`

Run : `java BankingApplication`

Key OOP Concepts in Banking App

OOP Principle	How It's Used in the Banking App
1.Encapsulation -	Account data (balance, account number) is private, accessible only via methods.
2.Abstraction -	`Account` is abstract; users only interact with its child classes.
3.Inheritance -	`SavingsAccount` & `CurrentAccount` inherit from `Account`, reusing logic.
4.Polymorphism -	`withdraw()` & `deposit()` work differently in different accounts.

✅ Encapsulation

Encapsulation is the binding of data (variables) and methods into a single unit (class). It prevents direct access to sensitive data and ensures data security.

- *Example:* The Account class keeps balance as `private`, so it can only be modified through deposit() and withdraw() methods.

✅ Abstraction

Abstraction hides implementation details and only exposes necessary functionalities.

- *Example:* We define an abstract Account class that only specifies what every account should have (`deposit()` & `withdraw()`), but each account type implements it differently.

✅ Inheritance

Inheritance allows code reuse by creating a base class (Account) and extending it into different account types.

- *Example:* A `SavingsAccount` extends `Account`, inheriting its properties but adding its own withdrawal rules (e.g., minimum balance).

✅ Polymorphism

Polymorphism allows one method to have different behaviors in different classes.

- *Example:* `deposit()` and `withdraw()` behave differently in SavingsAccount and CurrentAccount, but they share the same method signature.

System Design and Class Structure

BankingApp
├── models
│ ├── **Account** (Abstract)
│ ├── **SavingsAccount** (Extends Account)
│ ├── **CurrentAccount** (Extends Account)
├── services
│ ├── **BankService** (Manages Accounts)
├── main
│ ├── **BankingAppMain** (User Interaction)