

Banking Management System

Java | File I/O | Console-Based | Mini Project | Project Report

Project Report

Student Name: _____

Roll No.: _____

Course / Class: _____

Institution: _____

Academic Session: _____

Guide / Faculty: _____

Certificate

This is to certify that the project entitled “Banking Management System” has been developed as a mini project using Java and File I/O. The project demonstrates basic banking account operations through a console-based application.

Declaration

I hereby declare that this project report and the accompanying implementation are prepared for academic purposes. The work describes a console-based banking system using Java, object-oriented programming concepts, and file handling.

Acknowledgement

I express my sincere gratitude to my faculty/guide for guidance and support during the development of this project. I also acknowledge the Java documentation and learning resources used to understand file handling, classes, methods, exception handling, and console input.

Abstract

The Banking Management System is a beginner-level Java console application designed to simulate essential banking operations. The system allows users to create and close accounts, deposit money, withdraw money, and review transaction history. Account information and transaction records are persisted using files, so the application can retain data between executions. The project demonstrates practical use of Java classes, encapsulation, collections, file I/O, validation, exception handling, and menu-driven programming.

Introduction

Banking applications require structured handling of account information, balances, and transactions. This academic project models a simplified banking environment without connecting to a real bank or payment network. A console interface keeps the project easy to understand while allowing students to practice core Java programming concepts.

Problem Statement

A small academic banking application needs to maintain customer account records and perform basic transactions reliably. The system should provide a simple menu, validate input, prevent invalid withdrawals, preserve account information in files, and provide a transaction history for each account.

Objectives

1. Create and maintain bank accounts.
2. Support account closure.
3. Deposit and withdraw funds with validation.
4. Display account details and balances.
5. Maintain transaction history.

6. Persist application data using Java File I/O.
7. Demonstrate object-oriented programming and exception handling.

Scope

The system is intended for academic demonstration. It covers basic savings/current-style account records, deposits, withdrawals, account closure, and transaction history. It does not implement real banking infrastructure, online payments, KYC, interest calculation, authentication, encryption, multi-user concurrency, or integration with banking APIs.

Technology Stack

Programming Language: Java (JDK 17+ recommended)

Interface: Console / command line

Storage: Local text/serialized files using Java File I/O

Core concepts: OOP, classes/objects, methods, collections, exceptions, file handling, input validation.

System Architecture

The application follows a simple layered flow: Console Menu → Input Validation → Banking Service / Business Logic → Account & Transaction Models → File Storage. User commands are accepted from the console, validated, processed in memory, and persisted to local files.

Main Modules

1. Account Management: create, view, search, and close accounts.
2. Deposit Module: validates amount and increases account balance.
3. Withdrawal Module: validates amount and sufficient balance before reducing balance.
4. Transaction History: records deposits, withdrawals, and account events.
5. File Storage: saves account and transaction information locally.
6. Console Menu: provides navigation through numbered options.

Functional Requirements

FR1: Create a unique account.

FR2: Display account information.

FR3: Deposit a positive amount.

FR4: Withdraw only when sufficient balance exists.

FR5: Display transaction history.

FR6: Close an existing account.

FR7: Preserve data after program restart.

FR8: Handle invalid menu choices and malformed numeric input.

Non-Functional Requirements

Usability: clear numbered console menus and prompts.

Reliability: validate inputs and handle file errors.

Maintainability: separate model, service, and storage responsibilities.

Portability: run on systems with a compatible Java JDK.

Performance: suitable for a small academic dataset stored locally.

Data Design

Typical account fields: account number, holder name, account type, balance, status.

Typical transaction fields: transaction ID, account number, transaction type, amount, timestamp, resulting balance.

Files may be organized as `accounts.dat/accounts.txt` and `transactions.dat/transactions.txt` depending on the implementation.

Core Algorithms

Create account: validate customer details → generate/check unique account number → initialize balance → save account.

Deposit: find active account → validate amount > 0 → increase balance → create transaction → persist changes.

Withdraw: find active account → validate amount > 0 and amount ≤ balance → decrease balance → create transaction → persist changes.

Close: locate account → verify active status → mark closed → persist update.

Input Validation and Error Handling

The program should reject empty names, invalid account numbers, non-positive transaction amounts, withdrawals exceeding the available balance, duplicate account numbers, and invalid menu choices. Java exception handling can be used for `NumberFormatException`, `IOException`, and other expected runtime conditions.

Security Considerations

This is an academic simulation and should not be used for real banking. Sensitive information should not be stored in plain text in a production system. A real banking system would require authentication,

authorization, encryption, audit controls, secure databases, transaction integrity, fraud controls, and regulatory compliance.

Testing

Test cases should include: valid account creation; duplicate account attempt; zero/negative deposit; valid deposit; valid withdrawal; insufficient-balance withdrawal; account search; transaction history; account closure; operations on closed accounts; invalid menu input; corrupted/missing storage file; application restart and data recovery.

Sample Test Cases

TC01 Create account with valid details → account created.

TC02 Deposit 5,000 into active account → balance increases by 5,000.

TC03 Withdraw 2,000 with sufficient balance → balance decreases by 2,000.

TC04 Withdraw more than balance → transaction rejected.

TC05 Deposit 0 → transaction rejected.

TC06 Close active account → status changes to closed.

TC07 Deposit to closed account → transaction rejected.

TC08 Restart application → previously saved records are loaded.

Limitations

The system is console-based and intended for learning. Local file storage is not equivalent to a transactional database. There is no real authentication, encryption, network access, concurrent transaction control, or integration with external banking systems.

Future Enhancements

Possible extensions include a JavaFX/Swing GUI, MySQL/PostgreSQL database integration, user login and role-based access, password hashing, OTP/2FA, interest calculation, fund transfer, scheduled transactions, downloadable statements, audit logs, REST APIs, and automated unit/integration testing.

Conclusion

The Banking Management System provides a compact demonstration of how Java can be used to model accounts and transactions. It combines object-oriented programming, file persistence, validation, exception handling, and menu-driven interaction. Although it is not a production banking platform, it provides a practical foundation for learning how a stateful business application is structured.

References

1. Oracle Java Documentation — Java Language and API documentation.

2. Java Platform documentation for `java.io`, `java.nio.file`, `java.time`, collections, and exception handling.
3. Standard Java programming textbooks and classroom notes.
4. Institution-provided project guidelines, where applicable.