**University of Science and Technology Chittagong**

**Department of Computer Science and Engineering**

**Batch – 43(B)**

**Lab Project : ATM Simulation System**

**Course Name: Object Oriented Programming**

**Course Code: CSE 124**

**Submitted by**

**Name: Nahian Reja Rafa**

**Roll No: 24070169**

**Reg No: 1220**

**Submitted to**

**Debabrata Mallik**

**Lecturer**

**Department of Computer Science & Engineering**

**Date of Submission: 20 / 05 / 2025**

**Table Of Contents**

[1. List of Figures 3](#_Toc198806537)

[2. Proposal 4](#_Toc198806538)

[2.1 Motivation 4](#_Toc198806539)

[2.2 Purpose 4](#_Toc198806540)

[2.3 Features 5](#_Toc198806541)

[2.4 Methodology 5](#_Toc198806542)

[3. Implementation 8](#_Toc198806543)

[**3.1** **Class Diagram** 8](#_Toc198806544)

[**3.2** **Code** 10](#_Toc198806545)

[3.2.1. **Explanation:** 12](#_Toc198806546)

[**3.2.2. Explanation:** 17](#_Toc198806547)

[**3.2.3. Explanation:** 23](#_Toc198806548)

[3.3. Screenshort: 24](#_Toc198806549)

[**3.4. Output:** 30](#_Toc198806550)

[**4.** **Remaining Work** 32](#_Toc198806551)

[**5.** **Future Implementation** 33](#_Toc198806552)

[**6.** **Conclusion** 34](#_Toc198806553)

[**7.Appendix** 35](#_Toc198806554)

[**7.1 References** 37](#_Toc198806555)

# List of Figures

|  |  |
| --- | --- |
| Fig-1 | Class Diagram |
| Fig-2.1 | Parent Class |
| Fig-2.2 | Parent Class |
| Fig-2.2.1 | Parent Class |
| Fig-2.3.1 | Main class |
| Fig-2.3.2 | Main Class |
| Fig-2.3.3 | Main Class |
| Fig-2.4 | Run |
| Fig-2.4.1 | Run |
| Fig-2.4.2 | Run |
| Fig-3 | Code |

**Project Title: ATM Machine Simulation**

# 2. Proposal

## 2.1 Motivation

In the modern era of digital banking, ATM (Automated Teller Machine) systems play a crucial role in providing

secure, fast, and user-friendly access to financial services. Learning how these systems work not only enhances

understanding of real-world banking applications but also strengthens programming skills—especially in object

oriented programming (OOP). This project is motivated by the desire to simulate a simple yet functional ATM system in Java that mimics the basic operations of a real-world ATM machine, while practicing core Java concepts.

## 2.2 Purpose

The main purpose of this project is to design and implement ATM simulation system using Java. The system allows a user to securely log in using a PIN, and then perform standard banking operations like checking

balance, depositing funds, and withdrawing money. It also includes security mechanisms such as account lockout

after multiple failed login attempts. Through this project, the goal is to:

❖ Understand and apply object-oriented programming principles (inheritance, abstraction, encapsulation).

❖ Gain hands-on experience with Java Swing for building graphical user interfaces.

❖ Simulate real-life ATM operations to enhance practical programming skills.

❖ Implement simple security measures in a banking simulation environment.

## 2.3 Features

This ATM simulation system includes the following key features:

❖ **User Authentication with PIN**: Secure login system that validates the entered PIN.

❖ **Account Lockout Mechanism**: After 3 failed login attempts, the account is temporarily locked for 30 seconds.

❖ **Check Balance**: Users can view their current account balance.

❖ **Deposit Money**: Allows users to deposit a valid amount to their account.

❖ **Withdraw Money**: Allows users to withdraw money if they have sufficient balance.

❖ **Input Validation**: Handles invalid inputs (e.g., negative amounts, non-numeric entries).

❖ **Modular Design**: Uses abstract and concrete classes to separate account logic from GUI, making the system

maintainable and extendable.

## 2.4 Methodology

The methodology for building the ATM Simulation System consists of several phases: **requirements analysis**, **system design**, **implementation**, **testing & validation**, and **documentation & deployment**. Each phase follows an iterative and incremental approach, ensuring that feedback at every step informs subsequent work.

**Requirements Analysis**

1. **Stakeholder Interviews:**
   * Met with course instructor and peers to clarify functional needs: registration, authentication, balance inquiry, withdrawal, transfer, and transaction logging.
   * Identified non-functional constraints: console-only interface, in-memory data, simple error handling.
2. **Use Case Definition:**
   * Wrote clear use cases (e.g., “User registers account,” “User transfers funds”) to enumerate all user–system interactions.
   * Captured edge cases such as insufficient funds, invalid credentials, and transfer to non-existent users.
3. **Requirements Specification Document:**
   * Compiled a short spec listing inputs, outputs, and success/failure criteria for each use case.
   * Prioritized core features for the first iteration; earmarked enhancements (e.g., persistence, GUI) as future work.

**System Design**

1. **Conceptual Modeling:**
   * Identified three primary classes—User, Account, and ATMSystem—and their relationships.
   * Chose composition: each User “has-a” single Account.
2. **UML Diagrams:**
   * Created a **class diagram** to capture attributes and methods.
   * Drafted a **sequence diagram** for key flows (login, withdrawal, transfer) to verify interaction logic.
3. **Design Patterns & Principles:**
   * Applied **Single Responsibility Principle**, assigning each class a clear, focused role.
   * Used **fail-fast validation** in the User constructor to prevent invalid object states (e.g., weak passwords).
   * Opted for **encapsulation**: sensitive fields (password, balance) are private, exposed only via controlled methods.

**Implementation**

1. **Development Environment:**
   * **JDK 11** (or higher) for language features like String.format().
   * **IntelliJ IDEA** (Community Edition) for code editing, refactoring support, and built-in console.
   * **Git** for version control: created a feature/atm-simulation branch, committed iteratively after each milestone.
2. **Coding Standards:**
   * Followed Oracle’s **Java Code Conventions**: meaningful identifiers, Javadoc stubs for public methods, consistent indentation.
   * Maintained **logging of actions** within Account via a List<String> transactions—no external libraries needed.

# Implementation

## **Class Diagram**

The following class diagram outlines the structure of the system using OOP:

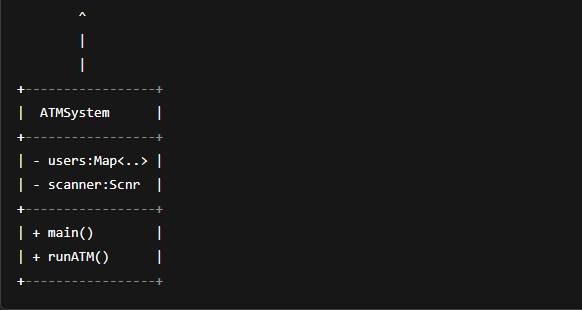
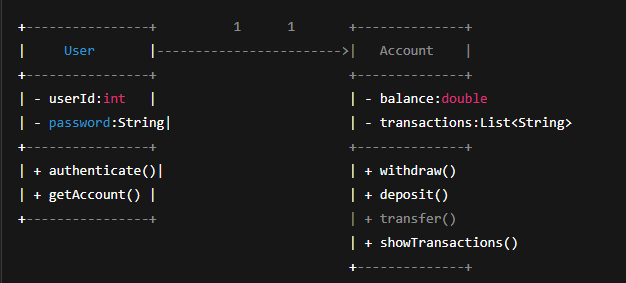


Fig-1. Class Diagram.

Explanation:

* The user starts the interaction by attempting to log in to the ATM system via the console.
* ATMSystem calls a method to fetch the User object using the entered userId from the internal Map<Integer, User>.
* The system calls authenticate(inputPassword) on the User object to validate the entered password.
* If the password is correct, it returns true.
* On successful login, ATMSystem invokes runATM(user) which allows the user to perform ATM operations like balance check, withdrawal, transfer, etc.
* If the user chooses to withdraw, the system calls withdraw(amount) on the Account object.
* This method checks balance, updates it, and logs the transaction.

## **Code**

Here’s the full implementation of the system:

User.java(Parent class):

package com.mycompany.atm;

public class User {

private int userId;

private String password;

private Account account;

public User(int userId, String password, double balance) {

if (!isValidPassword(password)) {

throw new IllegalArgumentException("Password must contain at least one special character (@, #, $, %)");

}

this.userId = userId;

this.password = password;

this.account = new Account(balance);

}

private boolean isValidPassword(String password) {

return password.matches(".\*[@#$%].\*");

}

public boolean authenticate(String inputPassword) {

return this.password.equals(inputPassword);

}

public Account getAccount() {

return account;

}

public int getUserId() {

return userId;

}

}

### 3.2.1. **Explanation:**

* The User class represents an individual user in the ATM system.
* It stores the user's unique ID, password, and their associated Account object.
* The constructor initializes the user with a given ID, password, and starting balance.
* It includes a **password validation check** to ensure it contains at least one special character (@, #, $, %).
* If the password is invalid, it throws an IllegalArgumentException.
* The method authenticate() compares the input password with the stored one to verify login.
* The getAccount() method returns the user's Account object for transactions.
* The getUserId() method returns the user's ID.
* This class ensures both **security (via password check)** and **account ownership** within the ATM system.

Account.java(Concrete child class):

package com.mycompany.atm;

import java.util.ArrayList;

import java.util.List;

public class Account {

private double balance;

private List<String> transactions;

public Account(double balance) {

this.balance = balance;

transactions = new ArrayList<>();

transactions.add("Account opened with ₹" + balance);

}

public double getBalance() {

return balance;

}

public boolean withdraw(double amount) {

if (amount <= balance) {

balance -= amount;

transactions.add("Withdrawn ₹" + amount + " | Balance: ₹" + balance);

return true;

}

transactions.add("Failed withdrawal of ₹" + amount + " due to insufficient balance.");

return false;

}

public void deposit(double amount) {

balance += amount;

transactions.add("Deposited ₹" + amount + " | Balance: ₹" + balance);

}

public void transfer(Account receiver, double amount) {

if (withdraw(amount)) {

receiver.deposit(amount);

transactions.add("Transferred ₹" + amount + " to another account");

receiver.addTransaction("Received ₹" + amount + " from another account");

} else {

transactions.add("Failed transfer of ₹" + amount + " due to insufficient balance.");

}

}

// Helper for receiver transaction log

private void addTransaction(String note) {

transactions.add(note + " | Balance: ₹" + balance);

}

public void showTransactions() {

System.out.println("Transaction History:");

if (transactions.isEmpty()) {

System.out.println("No transactions yet.");

} else {

for (String t : transactions) {

System.out.println("- " + t);

}

}

}

}

### **3.2.2. Explanation:**

* This Account class manages a bank account’s balance and transaction history.
* It has a balance variable to store the current amount of money.
* The transactions list keeps a record of all account activities.
* In the constructor, the balance is initialized and the transactions list is created as an empty ArrayList. Right after, a message noting the account opening with the initial balance is added to the transactions.
* The getBalance() method returns the current balance.
* The withdraw() method reduces the balance if there are sufficient funds and logs the success or failure in the transactions.
* The deposit() method adds money to the balance and logs the deposit in the transactions list.

ATMSystem.java(Main class):

package com.mycompany.atm;

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class ATMSystem {

private static Map<Integer, User> users = new HashMap<>();

private static Scanner scanner = new Scanner(System.in);

public static void main(String[] args) {

System.out.println("Welcome to ATM System");

// Register user

System.out.print("Create your User ID: ");

int userId = scanner.nextInt();

scanner.nextLine(); // consume newline

System.out.print("Create your Password (must include @, #, $, %): ");

String password = scanner.nextLine();

while (!password.matches(".\*[@#$%].\*")) {

System.out.println("Invalid password. Must contain at least one special character (@, #, $, %).");

System.out.print("Enter password again: ");

password = scanner.nextLine();

}

System.out.print("Enter initial balance: ₹");

double balance = scanner.nextDouble();

users.put(userId, new User(userId, password, balance));

System.out.println("User registered successfully!\n");

// Login section

System.out.print("Enter User ID to login: ");

int loginId = scanner.nextInt();

scanner.nextLine(); // consume newline

System.out.print("Enter Password: ");

String loginPassword = scanner.nextLine();

User user = users.get(loginId);

if (user != null && user.authenticate(loginPassword)) {

System.out.println("Login successful!");

runATM(user);

} else {

System.out.println("Invalid credentials.");

}

}

public static void runATM(User user) {

int choice;

do {

System.out.println("\n1. Balance Inquiry\n2. Withdraw\n3. Transfer\n4. View Transactions\n5. Exit");

System.out.print("Enter choice: ");

choice = scanner.nextInt();

Account account = user.getAccount();

switch (choice) {

case 1:

System.out.println("Your current balance is: ₹" + account.getBalance());

break;

case 2:

System.out.print("Enter amount to withdraw: ₹");

double amount = scanner.nextDouble();

if (account.withdraw(amount)) {

System.out.println("Withdrawal successful.");

} else {

System.out.println("Insufficient balance.");

}

break;

case 3:

System.out.print("Enter receiver User ID: ");

int receiverId = scanner.nextInt();

System.out.print("Enter amount to transfer: ₹");

double transferAmt = scanner.nextDouble();

User receiver = users.get(receiverId);

if (receiver != null) {

account.transfer(receiver.getAccount(), transferAmt);

System.out.println("Transfer attempted.");

} else {

System.out.println("Receiver not found.");

}

break;

case 4:

account.showTransactions();

break;

case 5:

System.out.println("Thank you for using the ATM System.");

break;

default:

System.out.println("Invalid option.");

}

} while (choice != 5);

}

}

### **3.2.3. Explanation:**

* This ATMSystem class simulates a simple ATM interface.
* It stores users in a HashMap with their User ID as the key.
* The program first registers a user by taking their User ID, password (which must include special characters), and initial balance.
* After registration, the user can log in with their credentials.
* If authentication succeeds, the runATM() method starts an interactive menu.
* This menu lets users check balance, withdraw money, transfer funds to another user, view transaction history, or exit.
* Withdrawal and transfer operations update the user’s account and log transactions.
* The system handles invalid inputs and provides feedback.
* The program loops until the user chooses to exit.

## 3.3. Screenshort:



Fig-2.1: parent class

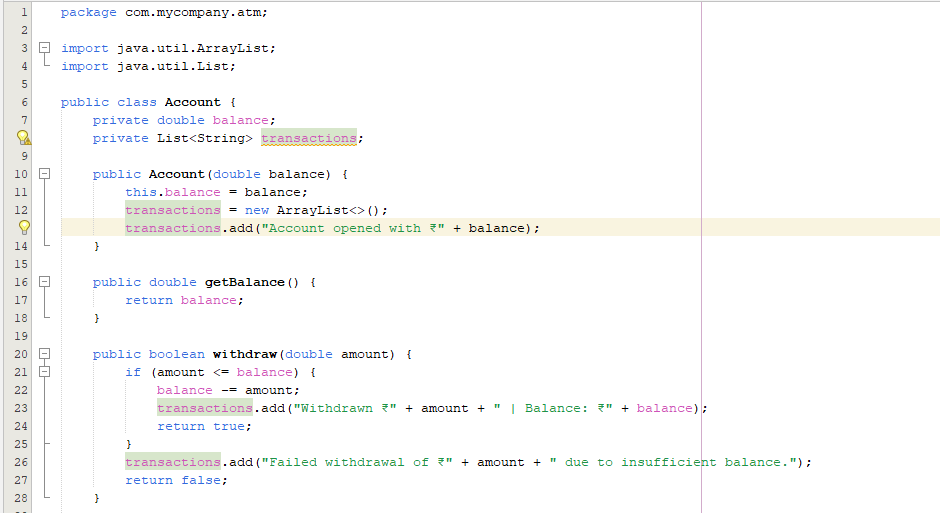


Fig-2.2: Parent class



Fig-2.2.1: Parent class

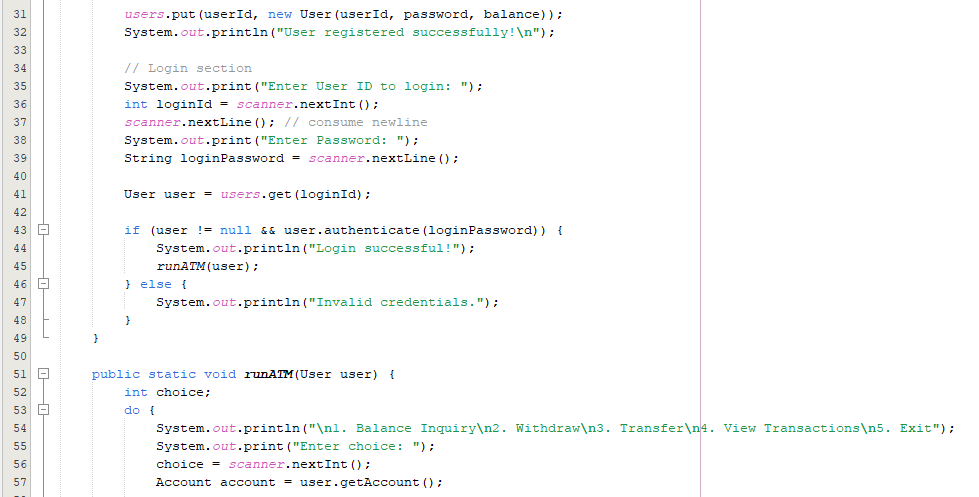


Fig-2.3.1: Main class

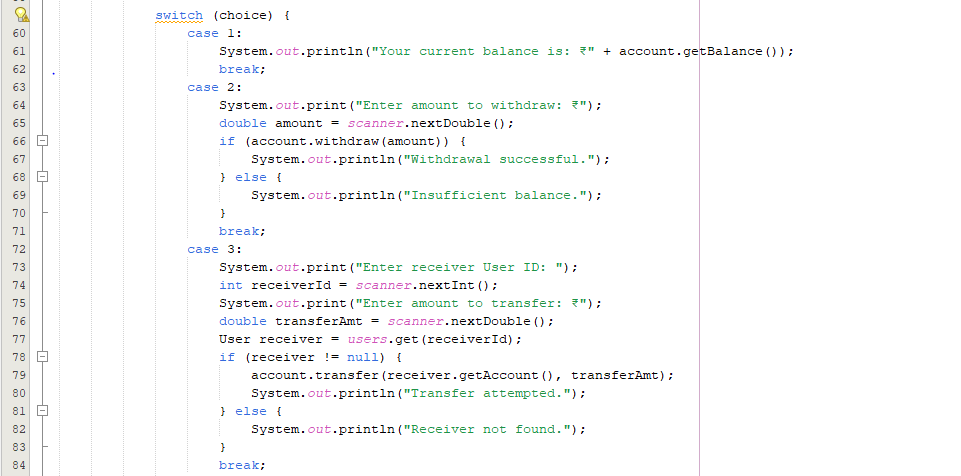


Fig-2.3.2: Main class

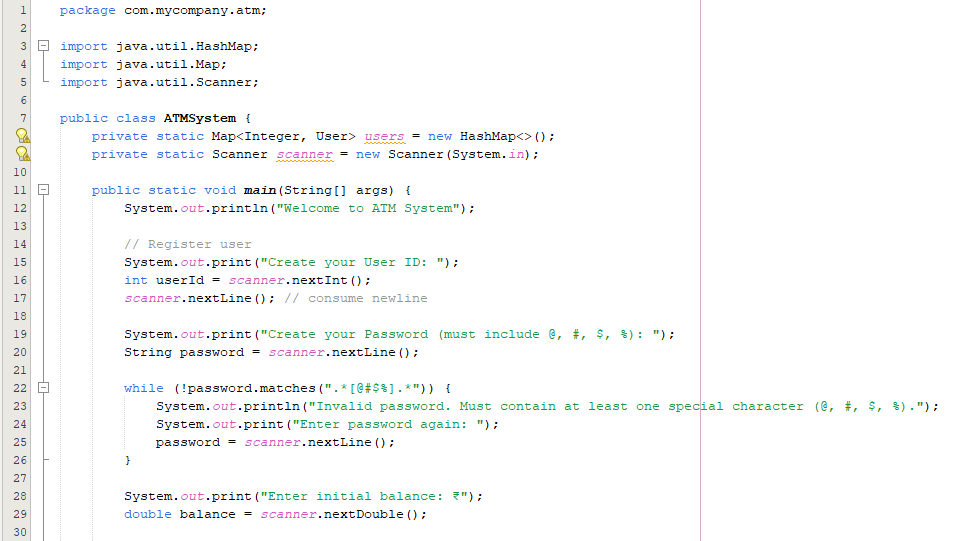


Fig-2.3.3: Main class

## **3.4. Output:**

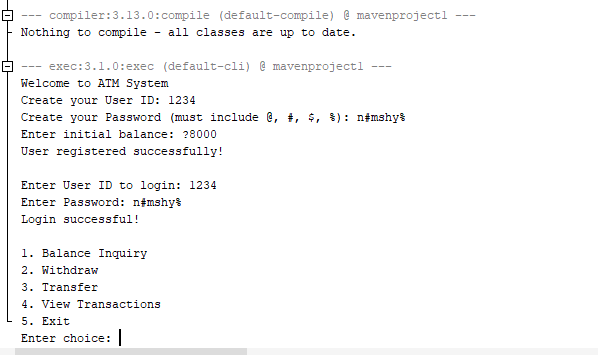


Fig-2.4: Run

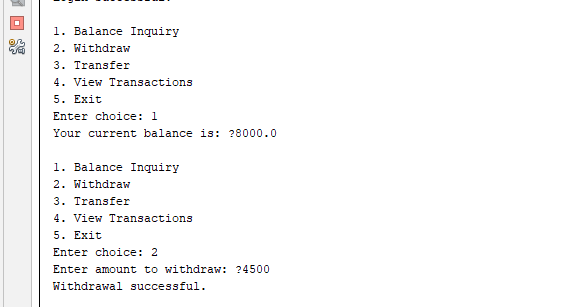


Fig-2.4.1:Run

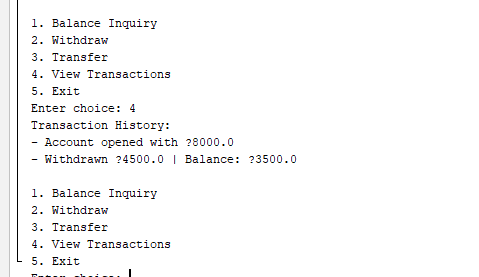


Fig-2.4.2:Run

# **Remaining Work**

The current Java ATM system implements basic functionalities including user registration with password validation, login authentication, balance inquiry, withdrawal, fund transfer between users, and viewing transaction history. However, several enhancements remain to be completed for a fully functional and secure ATM application. These include implementing data persistence to save user and transaction data between sessions, enhancing security by encrypting passwords and adding multi-factor authentication, and improving input validation to handle edge cases and prevent invalid operations. Additionally, features such as transaction limits, real-time transaction alerts, and support for multiple accounts per user could be added. A more user-friendly interface, either graphical or web-based, would improve usability. Finally, comprehensive error handling and unit testing are necessary to ensure system reliability and robustness before deployment.

# **Future Implementation**

To make this project more realistic and production-ready, the following features can be implemented in future

versions:

* Add persistent data storage (e.g., database) to save users and transactions across sessions.
* Implement password encryption for enhanced security.
* Introduce account lockout or cooldown after multiple failed login attempts.
* Add multi-factor authentication (MFA) for stronger user verification.
* Enable real-time transaction alerts via email or SMS notifications.
* Support multiple accounts per user for greater flexibility.
* Provide detailed reports and analytics of account activity.
* Integrate with external payment gateways or services.
* Improve user experience by developing a graphical or web-based interface.
* Implement comprehensive error handling and validation for robustness.
* Develop automated unit and integration testing to ensure reliability.

# **Conclusion**

The implemented ATM system successfully demonstrates the essential functionalities required for a basic banking application. Through user registration with password validation, secure login authentication, and a suite of transactional operations such as balance inquiry, withdrawal, fund transfers, and transaction history management, the system encapsulates core principles of object-oriented programming and interactive console-based user interfaces. This project highlights the importance of validating user inputs, managing account states, and maintaining detailed transaction logs, which are critical in any financial software.

Although the current system is functional and user-friendly, it represents a simplified model primarily designed for educational purposes. There remains substantial potential for future enhancements, including the introduction of persistent data storage for durability, advanced security mechanisms like password encryption and multi-factor authentication to protect sensitive user data, and a more intuitive user experience through graphical interfaces. By addressing these areas, the ATM system can evolve into a robust, secure, and scalable platform suitable for real-world banking needs.

Overall, this project not only reinforces fundamental programming concepts but also lays a strong foundation for further development in financial software engineering.

# **7.Appendix**

* **User Management:**  
  The User class handles the creation and validation of users. It ensures that each user has a unique ID and a secure password containing at least one special character (@, #, $, or %). It also provides authentication functionality by comparing the entered password with the stored one.
* **Account Operations:**  
  The Account class manages all banking transactions such as deposits, withdrawals, and fund transfers. It maintains a real-time balance and a list of all transaction logs for transparency. Each operation updates the transaction history, whether successful or failed.
* **ATM Interface Logic:**  
  The ATMSystem class is the main driver of the program. It provides a console-based interface that allows users to register, log in, and perform actions such as balance inquiry, withdrawal, transfer, and viewing transaction history. It uses a menu loop to handle user choices continuously until exit.

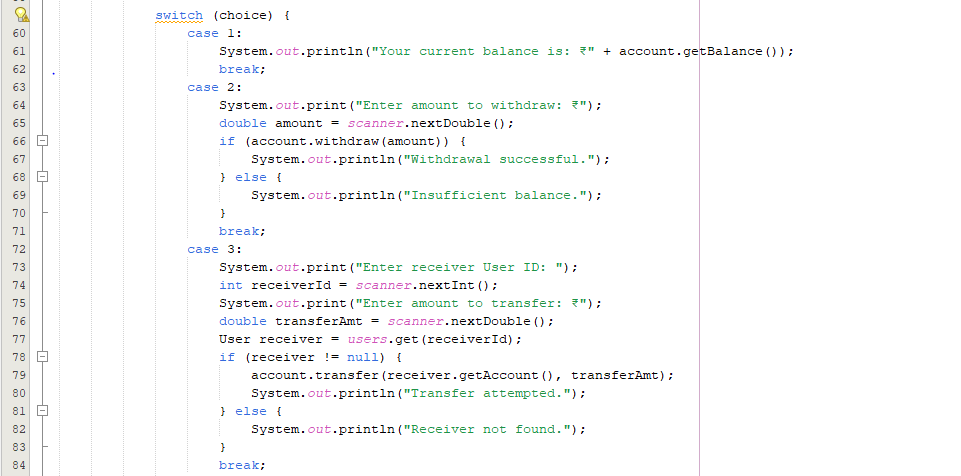


Fig-3: code

* **Data Handling and Validation:**  
  User data is stored in a HashMap for quick access and management. Input from users is collected using a Scanner, and validation checks (such as for password strength or sufficient balance) are performed at each step to ensure proper system behavior.
* **Transaction History Tracking:**  
  Every financial action, whether it's a deposit, withdrawal, or transfer, is recorded with details such as amount and resulting balance. This feature helps in reviewing past activities and adds accountability to the system.

## **7.1 References**

* Java Official Documentation: https://docs.oracle.com/javase/
* Java Swing Tutorials: https://docs.oracle.com/javase/tutorial/uiswing/
* W3 School: https://www.w3schools.com/
* w3 Resource: https://www.w3resource.com/c-programming-exercises/#google\_vignette
* Chatgpt by open AI