JAVA CASE STUDY (22) - ATM MACHINE

A case study Report submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted by

2023000598 – Navneeth

2023000600 – Nikhil Varma

2023000627 - Bhavana

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM SCHOOL OF TECHNOLOGY

Visakhapatnam

2025

Explanation:

ATM MACHINE SIMULATION

PROBLEM STATEMENT

Simulate the behavior of an ATM machine where users can check their balance, deposit money, withdraw money, and transfer funds Implement different types of accounts (e.g, checking, savings).

CONCEPTS USED

- **Inheritance:** Create a base Account class and subclasses like CheckingAccount and SavingsAccount.
- **Polymorphism** Implement method overriding for transaction types (e.g. withdrawing from savings vs. checking).
- **Constructors** Use constructors to initialize account details such as balance, account type, and user details.
- **Interfaces**, Handle cases such as uinsufficient funds, invalid PIN, or exceeding withdrawal limits.
- **Arrays:** Use arrays or lists to track transactions or account

KEY FEATURES

- Multi-user login and session management.
- Transaction history feature.
- Interface for easy command-line interaction.

CONCLUSION

This system effectively applies OOP principles to simulate realistic ATM machine operations.

With multiple accounts, sessions, and transaction history, it offers a robust and interactive user experience.

Code:

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Scanner;

interface Transactable {
    void deposit(double amount);
    void withdraw(double amount) throws InsufficientFundsException;
    void transfer(Account toAccount, double amount) throws
InsufficientFundsException;
}

abstract class Account implements Transactable {
    protected String accountNumber;
    protected String accountHolder;
    protected double balance;
    protected ArrayList<String> transactionHistory;

    public Account(String accountNumber, String accountHolder, double balance)
{
        this.accountNumber = accountNumber;
        this.accountHolder = accountHolder;
        this.balance = balance;
        this.transactionHistory = new ArrayList<>();
    }
```

```
    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        balance += amount;
        transactionHistory.add("Deposited: $" + amount);
        System.out.println("Deposit successful! New balance: $" + balance);
    }

    public void transfer(Account toAccount, double amount) throws
InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException("Transfer failed: Insufficient
funds.");
        }
        this.withdraw(amount);
        toAccount.deposit(amount);
        transactionHistory.add("Transferred: $" + amount + " to " +
toAccount.accountNumber);
        System.out.println("Transfer successful!");
    }

    public void printTransactionHistory() {
        System.out.println("\nTransaction History for " + accountHolder + " (" +
accountNumber + "):");
        for (String transaction : transactionHistory) {
```

```
            System.out.println(transaction);
        }
    }

    public abstract void withdraw(double amount) throws
InsufficientFundsException;
}

class CheckingAccount extends Account {
    private static final double OVERDRAFT_LIMIT = 100.00;

    public CheckingAccount(String accountNumber, String accountHolder, double
balance) {
        super(accountNumber, accountHolder, balance);
    }

    @Override
    public void withdraw(double amount) throws InsufficientFundsException {
        if (balance + OVERDRAFT_LIMIT < amount) {
            throw new InsufficientFundsException("Withdrawal failed: Overdraft
limit exceeded.");
        }
        balance -= amount;
        transactionHistory.add("Withdrawn: $" + amount);
        System.out.println("Withdrawal successful! New balance: $" + balance);
    }
}
```

```
class SavingsAccount extends Account {
    private static final double WITHDRAWAL_LIMIT = 500.00;

    public SavingsAccount(String accountNumber, String accountHolder, double balance) {
        super(accountNumber, accountHolder, balance);
    }

    @Override
    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount > WITHDRAWAL_LIMIT) {
            throw new InsufficientFundsException("Withdrawal failed: Exceeds savings withdrawal limit.");
        }
        if (balance < amount) {
            throw new InsufficientFundsException("Withdrawal failed: Insufficient funds.");
        }
        balance -= amount;
        transactionHistory.add("Withdrawn: $" + amount);
        System.out.println("Withdrawal successful! New balance: $" + balance);
    }
}

class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
```

```
        super(message);
    }
}

class User {
    private String name;
    private String pin;
    private HashMap<String, Account> accounts;

    public User(String name, String pin) {
        this.name = name;
        this.pin = pin;
        this.accounts = new HashMap<>();
    }

    public String getName() {
        return name;
    }

    public void addAccount(Account account) {
        accounts.put(account.accountNumber, account);
    }

    public Account getAccount(String accountNumber) {
        return accounts.get(accountNumber);
    }
```

```
    public boolean authenticate(String enteredPin) {
        return this.pin.equals(enteredPin);
    }

    public void printAccounts() {
        System.out.println("Accounts for " + name + ":");
        for (String accNum : accounts.keySet()) {
            System.out.println(" - " + accNum);
        }
    }
}

class ATM {
    private HashMap<String, User> users;
    private Scanner scanner;

    public ATM(Scanner scanner) {
        this.users = new HashMap<>();
        this.scanner = scanner;
    }

    public void registerUser(User user) {
        users.put(user.getName(), user);
    }
```

```
public void start() {
    System.out.print("Enter username: ");
    String username = scanner.nextLine();

    User user = users.get(username);
    if (user == null) {
        System.out.println("User not found!");
        return;
    }

    System.out.print("Enter PIN: ");
    String enteredPin = scanner.nextLine();

    if (!user.authenticate(enteredPin)) {
        System.out.println("Invalid PIN!");
        return;
    }

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

    System.out.print("Enter account number: ");
    String accountNumber = scanner.nextLine();

    Account account = user.getAccount(accountNumber);
    if (account == null) {
```

```
            System.out.println("Invalid account number!");
            return;
        }

        while (true) {
            System.out.println("\n1. Check Balance\n2. Deposit\n3. Withdraw\n4. Transfer\n5. Transaction History\n6. Exit");
            System.out.print("Choose an option: ");

            int choice;
            try {
                choice = scanner.nextInt();
            } catch (Exception e) {
                System.out.println("Invalid input! Please enter a number.");
                scanner.nextLine(); // clear invalid input
                continue;
            }

            try {
                switch (choice) {
                    case 1:
                        System.out.println("Balance: $" + account.getBalance());
                        break;
                    case 2:
                        System.out.print("Enter deposit amount: ");
                        account.deposit(scanner.nextDouble());
                        break;
```

```
            case 3:
                System.out.print("Enter withdrawal amount: ");
                account.withdraw(scanner.nextDouble());
                break;
            case 4:
                scanner.nextLine(); // consume leftover newline
                System.out.print("Enter target account number: ");
                String targetAccountNumber = scanner.nextLine();
                Account targetAccount = user.getAccount(targetAccountNumber);
                if (targetAccount == null) {
                    System.out.println("Invalid target account!");
                    break;
                }
                System.out.print("Enter transfer amount: ");
                account.transfer(targetAccount, scanner.nextDouble());
                break;
            case 5:
                account.printTransactionHistory();
                break;
            case 6:
                System.out.println("Goodbye!");
                return;
            default:
                System.out.println("Invalid choice!");
        }
    } catch (InsufficientFundsException e) {
```

```
                System.out.println(e.getMessage());
            } catch (Exception e) {
                System.out.println("Something went wrong. Please try again.");
                scanner.nextLine(); // to clear the buffer
            }
        }
    }
}

public class ATMApp {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        ATM atm = new ATM(scanner);

        User user1 = new User("Nikhil", "1234");
        user1.addAccount(new CheckingAccount("CHK123", "Nikhil", 1000));
        user1.addAccount(new SavingsAccount("SAV123", "Nikhil", 5000));

        User user2 = new User("Navneeth", "4321");
        user2.addAccount(new CheckingAccount("CHK123", "Navneeth", 9000));
        user2.addAccount(new SavingsAccount("SAV123", "Navneeth", 5000));

        User user3 = new User("Bhavana", "1012");
        user3.addAccount(new CheckingAccount("CHK123", "Bhavana", 8000));
        user3.addAccount(new SavingsAccount("SAV123", "Bhavana", 4000));
```

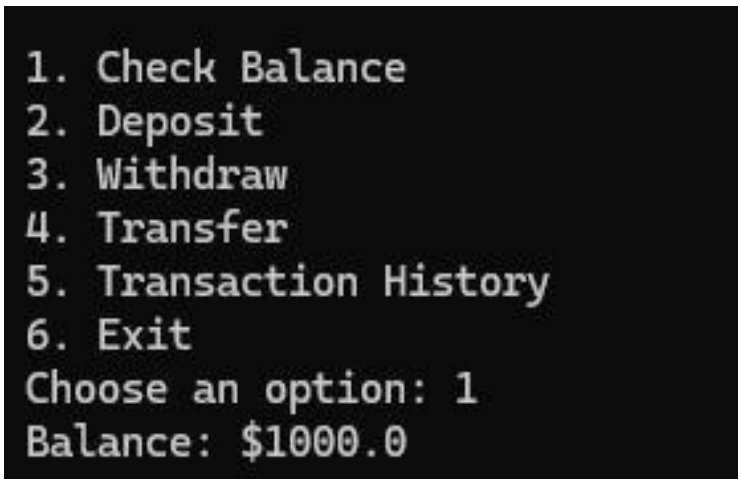
```
        atm.registerUser(user1);
        atm.registerUser(user2);
        atm.registerUser(user3);

        atm.start();

        scanner.close(); // important to close the scanner at the end
    }}
```

Output : The code will have multiple outputs due to its multiple scenarios possible due to the user's input data which can be him accessing his checkings or deposit accounts or any other accounts in general.

1. For Checking Balance



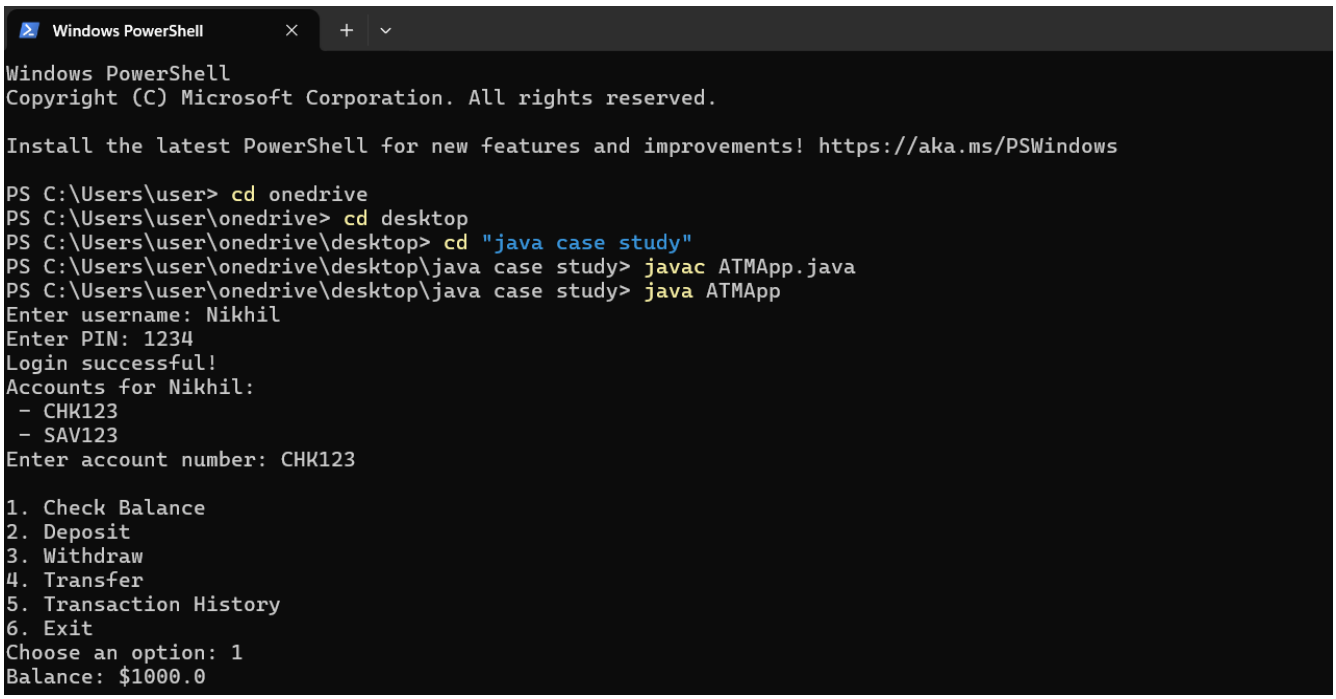
2.For Withdrawal of Account

```
1. Check Balance
2. Deposit
3. Withdraw
4. Transfer
5. Transaction History
6. Exit
Choose an option: 3
Enter withdrawal amount: 100
Withdrawal successful! New balance: $1100.0
```

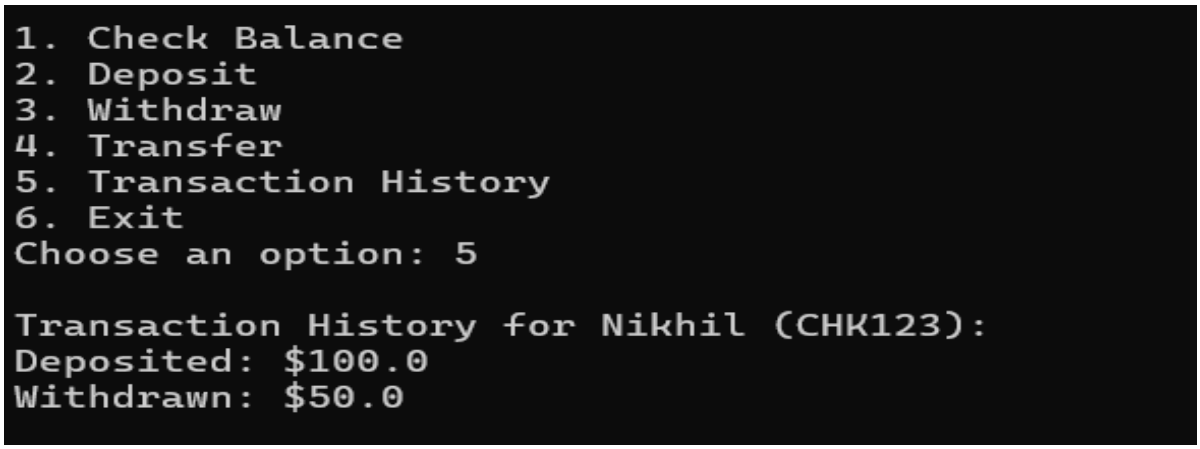
3.For Depositing in the Account

```
1. Check Balance
2. Deposit
3. Withdraw
4. Transfer
5. Transaction History
6. Exit
Choose an option: 2
Enter deposit amount: 200
Deposit successful! New balance: $1200.0
```

4. User(Nikhil) enters details to see any updates on his data(bank balance)



5.Transaction History



6.Transfer of money

```
1. Check Balance
2. Deposit
3. Withdraw
4. Transfer
5. Transaction History
6. Exit
Choose an option: 4
Enter target account number: SAV123
Enter transfer amount: 77
Withdrawal successful! New balance: $973.0
Deposit successful! New balance: $5077.0
Transfer successful!
```

Key Components of the Code:

1. Transactable Interface

- Declares methods that any account should implement:

void deposit(double amount);

void withdraw(double amount) throws InsufficientFundsException;

void transfer(Account toAccount, double amount) throws InsufficientFundsException;

---

2. Account Abstract Class

- Base class for different types of accounts.
- Contains:
    - Account number, holder name, balance
    - A list to track transaction history

- Implements the Transactable interface with logic for deposit() and transfer(), but **withdraw() is abstract**, so each account type can define its own rules.
- Has a method to print the transaction history.

---

3. CheckingAccount Class

- Subclass of Account.
- Allows overdraft up to $100:

if (balance + OVERDRAFT_LIMIT < amount)

- If withdrawal amount is within overdraft limit, the balance is reduced and transaction logged.

---

4. SavingsAccount Class

- Also a subclass of Account.
- Has a withdrawal limit of $500 per transaction.
- If withdrawal exceeds limit or balance, it throws InsufficientFundsException.

---

5. InsufficientFundsException

- Custom exception class for handling low balance or limits exceeded during transactions.

---

6. User Class

- Represents a user with:
    - Name
    - PIN for authentication
    - A collection of multiple accounts (using HashMap)
- Can add and retrieve accounts.

- Validates login via PIN.

---

7. ATM Class

- The main controller of the ATM simulation.
- Handles:
    - User login
    - PIN authentication
    - Account selection
    - Menu of operations:
        - Check balance
        - Deposit
        - Withdraw
        - Transfer between user's own accounts
        - View transaction history

---

8. ATMApp (Main Class)

- Sets up scanner and ATM instance.
- Creates **three users** (Nikhil, Navneeth, Bhavana), each with:
    - 1 checking account (CHK123)
    - 1 savings account (SAV123)
- Registers users with ATM.
- Starts the ATM interface.

Example Flow:

1. User runs the program.
2. Inputs their username and PIN.

3. Chooses an account to interact with.
4. ATM shows options like:
    - Deposit money
    - Withdraw money
    - Transfer money between their own accounts
    - View transaction history
    - Exit

Summary:

This Java program is a **basic simulation of an ATM system** using Object-Oriented Programming. It demonstrates:

- **Interfaces** for standard operations
- **Inheritance and Abstraction** for account types
- **Exception handling**
- **User authentication**
- **Polymorphism** via abstract methods (withdraw)
- **Use of collections (HashMap, ArrayList)**

These are all the outputs which can be implemented in the ATM Machine

THANK YOU

Corrected copy:

3. Chooses an account to interact with.
4. ATM shows options like:
    - Deposit money
    - Withdraw money
    - Transfer money between their own accounts
    - View transaction history
    - Exit

Summary:

This Java program is a **basic simulation of an ATM system** using Object-Oriented Programming. It demonstrates:

- **Interfaces** for standard operations
- **Inheritance and Abstraction** for account types
- **Exception handling**
- **User authentication**
- **Polymorphism** via abstract methods (withdraw)
- **Use of collections (HashMap, ArrayList)**

These are all the outputs which can be implemented in the ATM Machine

THANK YOU