Bank Account Management Console Application

INTRODUCTION:

This console-based Bank Account Management application simulates key banking operations. It provides features to create and manage accounts, handle transactions, and generate reports. The system supports two account types, **Current Account** and **Savings Account**, each with distinct properties. Users can perform essential functions like deposits, withdrawals, transfers, and viewing transaction history. The application’s modular design makes it scalable and straightforward.

EXPLANATION

This SRS document outlines the requirements for a Console-Based Banking Application developed in Java, utilizing SQL for database management. The application provides essential banking functionalities, such as account management, transactions, and report generation, designed to operate via a simple console interface. By focusing on user-friendly interactions, it enables efficient handling of common banking tasks like deposits, withdrawals, and balance checks.

CODE IMPLEMENTATION:

The program is implemented with a structured, object-oriented approach. Each core function—such as creating an account, viewing details, updating information, handling deposits, and generating reports—is encapsulated in dedicated methods for better organization and reuse.

WORKING:

The application begins by displaying a menu with nine options. Each option corresponds to a function that the user can execute by entering the option number. Here’s how each function works:

1. **Create Account**: Initializes a new account (Current or Savings) by capturing user details like name, contact info, and account type. Depending on the type, additional attributes like interest rate (for Savings) or overdraft limit (for Current) are assigned.
2. **View Account Details**: Displays all details of a specific account based on the account number or ID entered. Details include account type, balance, and any specific properties (e.g., interest rate or overdraft limit).
3. **Update Account Information**: Allows updating of contact information or other modifiable fields. This helps keep the account details accurate.
4. **Deposit**: Accepts an amount and credits it to the account balance, ensuring that the deposit is accurately logged in the transaction history.
5. **Withdraw**: Deducts an amount from the account if funds are sufficient, preventing overdraft for Savings accounts and enforcing limits for Current accounts. It also logs the withdrawal.
6. **Transfer**: Moves a specified amount from one account to another, ensuring both accounts’ balances are adjusted and the transaction is recorded.
7. **View Transaction History**: Displays past transactions, including deposits, withdrawals, and transfers for the account, with details like date and amount.
8. **Generate Reports**: Summarizes account activity over a specified period, useful for end-of-month or end-of-year reviews.
9. **Exit**: Exits the application, safely ending the session.

FUNCTIONAL REQUIRMENTS:

**5.1 Account Management**

* **Create Account**:
  + Users must be able to create a new account by providing required details (e.g., name, account type, etc.).
  + Each account will be assigned a unique account ID and stored in the SQL database.
* **View Account Details**:
  + Users can view their account details, including their balance, account type, and other personal information stored in the database.
* **Update Account Information**:
  + Users can update their personal details such as address or contact number.
  + Changes are saved into the SQL database.

**5.2 Transaction Management**

* **Deposit**:
  + Users can deposit a specified amount into their account.
  + The system should check if the amount is valid and then update the balance and transaction history.
* **Withdraw**:
  + Users can withdraw funds, but the system will check for sufficient funds in the account before allowing the transaction.
* **Transfer**:
  + Users can transfer funds between accounts by specifying the target account and the amount.
  + The system will validate the transfer and update the balances of both the source and destination accounts.

**5.3 Report Generation**

* **View Transaction History**:
  + Users can view a history of all transactions associated with their account, including deposits, withdrawals, and transfers.
* **Generate Reports**:
  + The system generates summary reports of the account’s transactions, balance, and account details.

**5.4 Exit**

* **Exit**:
  + The application should allow users to exit the program safely, ensuring that all transactions are saved and changes to accounts are committed to the database.

PROGRAM CODE:

FileName: DatabaseConnection.java

package BankingApplication;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class DatabaseConnection

{

private static final String URL = "jdbc:mysql://localhost:3306/banking\_app";

private static final String USER = "root";

private static final String PASSWORD = "root";

public static Connection getConnection() throws SQLException

{

return DriverManager.getConnection(URL, USER, PASSWORD);

}

public static void testConnection()

{

try (Connection conn = getConnection())

{

if (conn != null)

{

System.out.println("Database Connected Successfully");

}

else

{

System.out.println("Failed to connect to the database");

}

}

catch (SQLException e)

{

System.out.println("Database Connection Failed");

e.printStackTrace();

}

}

public static void main(String[] args)

{

testConnection(); // Call this method to check connection

}

}

FileName: BankingApp.java

package BankingApplication;

//BankApplication.java

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import java.util.HashMap;

import java.util.Scanner;

public class BankingApp {

private static HashMap<String,BankAccount> accounts=new HashMap<>();

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

public static void main(String[] args) {

while(true) {

System.out.println("\n--- Banking Application ---");

System.out.println("1. Create Account");

System.out.println("2. View Account Details");

System.out.println("3. Update Account Information");

System.out.println("4. Deposit");

System.out.println("5. Withdraw");

System.out.println("6. Transfer");

System.out.println("7. View Transaction History");

System.out.println("8. Generate Reports");

System.out.println("9.Exit");

System.out.print("Choose an option: ");

int choice =scanner.nextInt();

scanner.nextLine();

switch(choice) {

case 1:

createAccount();

break;

case 2:

viewAccountDetails();

break;

case 3:

updateAccountInfo();

break;

case 4:

deposit();

break;

case 5:

withdraw();

break;

case 6:

transfer();

break;

case 7:

viewTransactionHistory();

break;

case 8:

generateReports();

break;

case 9:

System.out.println("Thank you for using the banking application!");

return;

default:

System.out.println("Invalid choice. Please try again.");

}

}

}

//Create new Account

//Create a new account

private static void createAccount() {

String accountNumber;

while(true) {

System.out.print("Enter Account Number (8-16 digits):" );

accountNumber =scanner.nextLine().trim();

//check if account already exist

if(accounts.containsKey(accountNumber)) {

System.out.println("Error: Account with this number already exists.");

continue;

}

//Validate account number format

if(!isValidAccountNumber(accountNumber)) {

System.out.println("Error: Account number must be between 8 and 16 digits.");

}else {

break; //valid account number

}

}

String accountHolderName;

while(true) {

System.out.print("Enter Account Holder Name: ");

accountHolderName=scanner.nextLine().trim();

if(accountHolderName.isEmpty()) {

System.out.println("Error: Account holder name cannot be empty.");

}

else {

break;

}

}

String accountType;

while(true) {

System.out.print("Enter Account Type (S for Savings / C for Current): ");

String accountTypeInput=scanner.nextLine().trim().toLowerCase();

if(accountTypeInput.equals("s")) {

accountType="Savings";

break;

}

else if(accountTypeInput.equals("c")) {

accountType="Current";

break;

}

else {

System.out.println("Error: Invalid input. Please enter 'S' for Savings or 'C' for Current.");

}

}

String address;

while(true) {

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

address=scanner.nextLine().trim();

if(address.isEmpty()) {

System.out.println("Error: Address cannot be empty.");

}

else {

break;

}

}

String contactNumber;

while(true) {

System.out.print("Enter Contact Number (10 digits):");

contactNumber=scanner.nextLine().trim();

if(contactNumber.isEmpty()) {

System.out.println("Error: Contact number cannot be empty.");

}

else if(!isValidContactNumber(contactNumber)) {

System.out.println("Error: Invalid contact number. It must be a 10-digit number.");

}

else {

break;

}

}

try {

BankAccount newAccount = new BankAccount(accountNumber, accountHolderName,accountType,address,

contactNumber);

//Add to in-memory accounts map

accounts.put(accountNumber,newAccount);

//Save to database

try(Connection conn = DatabaseConnection.getConnection()){

String sql = "INSERT INTO account(accountNumber,accountHolderName,accountType,balance,address,"

+ "contactNumber) VALUES (?,?,?,?,?,?)";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setString(1, newAccount.getAccountNumber());

stmt.setString(2, newAccount.getAccountHolderName());

stmt.setString(3, newAccount.getAccountType());

stmt.setDouble(4, newAccount.getBalance());

stmt.setString(5, newAccount.getAddress());

stmt.setString(6, newAccount.getContactNumber());

int rowsInserted =stmt.executeUpdate();

if(rowsInserted > 0) {

System.out.println("Account created successfully and saved to database!");

}

}

catch(SQLException e) {

e.printStackTrace();

}

}

catch(IllegalArgumentException e) {

System.out.println(e.getMessage());

}

}

//validation methods

private static boolean isValidAccountNumber(String accountNumber) {

return accountNumber.matches("\\d{8,16}"); //account number must be 8 to 16 digits

}

private static boolean isValidContactNumber(String contactNumber) {

return contactNumber.matches("\\d{10}"); //contact number must be exactly 10 digits

}

//View account details

private static void viewAccountDetails() {

System.out.print("Enter Account Number: ");

String accountNumber=scanner.nextLine();

BankAccount account=accounts.get(accountNumber);

if(account!=null) {

account.displayAccountInfo();

}

else {

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

}

}

//Update account information

//Update account information

private static void updateAccountInfo() {

System.out.print("Enter Account Number: ");

String accountNumber=scanner.nextLine();

BankAccount account =accounts.get(accountNumber);

if(account!=null) {

System.out.print("Enter new Address: ");

String address=scanner.nextLine();

System.out.print("Enter new contact number:");

String contactNumber=scanner.nextLine();

//update in=memory account details

account.setAddress(address);

account.setContactNumber(contactNumber);

//Update in database

try(Connection conn = DatabaseConnection.getConnection()){

String sql="UPDATE accounts SET address = ?,contactNumber = ? WHERE accountNumber =?";

PreparedStatement stmt = conn.prepareStatement(sql);

stmt.setString(1, address);

stmt.setString(2, contactNumber);

stmt.setString(3, accountNumber);

int rowsUpdated=stmt.executeUpdate();

if(rowsUpdated>0) {

System.out.println("Account information updated successfully in database.");

}

else {

System.out.println("Failed to update account information in database.");

}

}

catch(SQLException e) {

e.printStackTrace();

}

}

else {

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

}

}

//deposit money

private static void deposit() {

System.out.print("Enter Account Number: ");

String accountNumber=scanner.nextLine();

BankAccount account =accounts.get(accountNumber);

if(account!=null) {

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

double amount = scanner.nextDouble();

account.deposit(amount);

try(Connection conn = DatabaseConnection.getConnection()){

String sql="UPDATE accounts SET balance = ? WHERE accountNumber = ?";

PreparedStatement stmt=conn.prepareStatement(sql);

stmt.setDouble(1, account.getBalance());

stmt.setString(2, account.getAccountNumber());

stmt.executeUpdate();

//insert transaction record

sql="INSERT INTO transactions (accountNumber, accountType, amount) VALUES (?,'Deposit',?)";

stmt=conn.prepareStatement(sql);

stmt.setString(1, accountNumber);

stmt.setDouble(2, amount);

stmt.executeUpdate();

System.out.println("Deposit successful and recorded in database.");

System.out.println("Current Balance: "+ account.getBalance()); //display current balance

}

catch(SQLException e) {

e.printStackTrace();

}

}

else {

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

}

}

//withdraw money

private static void withdraw() {

System.out.print("Enter Account Number: ");

String accountNumber = scanner.nextLine();

BankAccount account=accounts.get(accountNumber);

if(account!=null) {

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

double amount = scanner.nextDouble();

//check if withdrawal is successful in th BankAccount class

if(amount > 0 && amount<=account.getBalance()) {

account.withdraw(amount);

try(Connection conn = DatabaseConnection.getConnection()){

//update the balance in the accounts table

String sql = "UPDATE accounts SET balance = ? WHERE accountNumber = ?";

PreparedStatement stmt=conn.prepareStatement(sql);

stmt.setDouble(1, account.getBalance());

stmt.setString(2, account.getAccountNumber());

stmt.executeUpdate();

//insert a transaction record for withdrawal

sql="INSERT INTO transactions(accountNumber,accountType,amount) VALUES (?,'Withdrawal',?)";

stmt=conn.prepareStatement(sql);

stmt.setString(1, accountNumber);

stmt.setDouble(2, amount);

stmt.executeUpdate();

System.out.println("Withdarawal successful and recorded in database.");

System.out.println("Current Balance: "+account.getBalance() ); //display current balance

}

catch(SQLException e) {

e.printStackTrace();

}

}

else {

System.out.println("Insufficient balance or invalid withdrawal amount.");

}

}

else {

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

}

}

//transfer money

private static void transfer() {

System.out.print("Enter your Account Number: ");

String fromaccount\_number = scanner.nextLine();

System.out.print("Enter the recipient Account Number: ");

String toaccount\_number = scanner.nextLine();

System.out.print("Enter the amount to be transferred: ");

double amount = scanner.nextDouble();

BankAccount fromAccount = accounts.get(fromaccount\_number);

BankAccount toAccount = accounts.get(toaccount\_number);

if(fromAccount != null && toAccount != null) {

if(amount > 0 && amount <= fromAccount.getBalance()) {

fromAccount.withdraw(amount);

toAccount.deposit(amount);

try(Connection con = DatabaseConnection.getConnection()){

con.setAutoCommit(false);

String sql = "UPDATE accounts SET balance = ? WHERE accountNumber = ?";

PreparedStatement stmt = con.prepareStatement(sql);

stmt.setDouble(1, fromAccount.getBalance());

stmt.setString(2, fromAccount.getAccountNumber());

stmt.executeUpdate();

stmt.setDouble(1, toAccount.getBalance());

stmt.setString(2, toAccount.getAccountNumber());

stmt.executeUpdate();

sql = "INSERT INTO transactions(accountNumber, accountType, amount, description) VALUES (?, " + " 'Transfer Out', ?, ?)";

stmt = con.prepareStatement(sql);

stmt.setString(1, fromaccount\_number);

stmt.setDouble(2, amount);

stmt.setString(3, "Transferred to " + toaccount\_number);

stmt.executeUpdate();

sql = "INSERT INTO transactions(accountNumber, accountType, amount, description) VALUES (?, 'Transfer In', ?, ?)";

stmt = con.prepareStatement(sql);

stmt.setString(1, toaccount\_number);

stmt.setDouble(2, amount);

stmt.setString(3, "Recieved from " + fromaccount\_number);

stmt.executeUpdate();

con.commit();

System.out.println("Amount transferred successfully and recorded in the datbase");

}

catch(SQLException e) {

e.printStackTrace();

}

}

else {

System.out.println("Transfer failed. Insufficient balance or invalid amount");

}

}

else {

System.out.println("One or both the accounts can not be found");

}

}

//view transaction history

private static void viewTransactionHistory() {

System.out.print("Enter Account Number: ");

String accountNumber =scanner.nextLine();

BankAccount account = accounts.get(accountNumber);

if(account!=null)

{

account.displayTransactionHistory();

}

else{

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

}

}

//generate reports

//BankApplication.java

//inside the generateReports() method

private static void generateReports() {

System.out.println("\n--- Report Generation ---");

System.out.println("1. Customer Details");

System.out.println("2. Transaction History");

System.out.println("3. Total Balance");

System.out.println("4. Number of Accounts by Type");

System.out.println("Choose an option: ");

int choice =scanner.nextInt();

scanner.nextLine();

switch(choice) {

case 1:

generateCustomerDetailsReport();

break;

case 2:

System.out.print("Enter Account Number: ");

String accountNumber = scanner.nextLine();

BankAccount account=accounts.get(accountNumber);

if(account!=null) {

account.displayTransactionHistory();

}

else {

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

}

break;

case 3:

generateTotalBalanceReport();

break;

case 4:

generateAccountTypeReport();

break;

default:

System.out.println("Invalid choice. Please try again.");

}

}

//Report 1

private static void generateCustomerDetailsReport() {

System.out.println("\nCustomer Details Report:");

for(BankAccount account : accounts.values()) {

account.displayAccountInfo();

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

}

}

//report 2

private static void generateTotalBalanceReport() {

double totalBalance=accounts.values().stream().mapToDouble(BankAccount::getBalance).sum();

System.out.println("Total balance across all accounts: "+totalBalance);

}

//report 3

private static void generateAccountTypeReport() {

long savingsCount = accounts.values().stream().filter(acc -> acc.getAccountType().equalsIgnoreCase("Savings")).count();

long currentCount = accounts.values().stream().filter(acc -> acc.getAccountType().equalsIgnoreCase("Current")).count();

System.out.println("Number of Savings Account: "+ savingsCount);

System.out.println("Number of Current Accounts: "+currentCount);

}

}

FileName: BankAccount.java

package BankingApplication;

import java.util.ArrayList;

import java.lang.IllegalArgumentException;

import java.util.List;

public class BankAccount {

private String accountNumber;

private String accountHolderName;

private String accountType;

private double balance;

private String address;

private String contactNumber;

private List<String> transactionHistory;

public BankAccount(String accountNumber, String accountHolderName, String accountType, String address, String contactNumber) {

if(!isValidAccountNumber(accountNumber)) {

throw new IllegalArgumentException("Account number must be between 8 and 16 digits");

}

this.accountNumber = accountNumber;

this.accountHolderName = accountHolderName;

this.accountType = accountType;

this.address = address;

this.balance = 0.0;

this.contactNumber = contactNumber;

this.transactionHistory = new ArrayList<>();

}

private boolean isValidAccountNumber(String accountNumber) {

return accountNumber.matches("\\d{8,16}");

}

public String getAccountNumber(){

return accountNumber;

}

public String getAccountHolderName() {

return accountHolderName;

}

public String getAccountType() {

return accountType;

}

public double getBalance() {

return balance;

}

public void setAddress(String address) {

this.address = address;

}

public void setContactNumber(String contactNumber) {

this.contactNumber = contactNumber;

}

public void deposit(double amount) {

if(amount > 0) {

balance += amount;

transactionHistory.add("Deposited: " + amount);

}

else {

System.out.println("Invalid deposit amount");

}

}

public void withdraw(double amount) {

if(amount > 0 && amount <= balance) {

balance -= amount;

transactionHistory.add("Withdrew: " + amount);

}

else {

System.out.println("Insufficient balnce or invalid withdrawal amount");

}

}

public void transfer(BankAccount toAccount, double amount) {

if(amount > 0 && amount <= balance) {

this.withdraw(amount);

toAccount.deposit(amount);

transactionHistory.add("Transferred " + amount + " to " + toAccount.getAccountNumber());

toAccount.transactionHistory.add("Recieved " + amount + " from " + this.getAccountNumber());

}

else {

System.out.println("Transfer failed. Insufficient balnce or invalid amount");

}

}

public void displayAccountInfo() {

System.out.println("Account Number : " + accountNumber);

System.out.println("Account Holder : " + accountHolderName);

System.out.println("Account Type : " + accountType);

System.out.println("Balance : " + balance);

System.out.println("Address : " + address);

System.out.println("Contact Number : " + contactNumber);

}

public void displayTransactionHistory() {

System.out.println("Transaction History for Account: " + accountNumber);

for(String transaction : transactionHistory) {

System.out.println(transaction);

}

}

public String getContactNumber() {

return this.contactNumber;

}

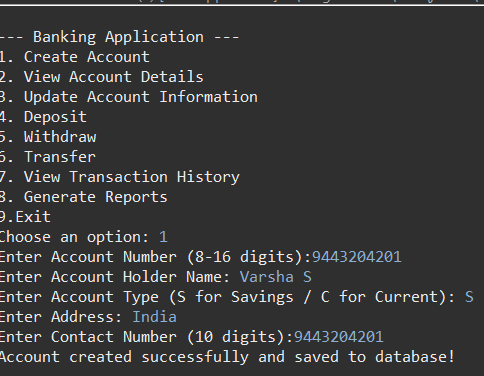
public String getAddress() {

return this.address;

}

}

SAMPLE OUTPUT



ER Diagram:

The **ER Diagram** represents the structure of the application’s database, detailing the relationships between entities.

**Entities and Attributes:**

1. **Account**
   * **Account\_ID** (Primary Key): Unique identifier.
   * **Account\_Type**: Type of account (Current or Savings).
   * **Customer\_Name**: Name of the account holder.
   * **Contact\_Info**: Contact details of the user.
   * **Balance**: Holds the current balance of the account.
2. **Current Account**
   * **Account\_ID** (Foreign Key): Linked to **Account**.
   * **Overdraft\_Limit**: Overdraft allowance for this account type.
3. **Savings Account**
   * **Account\_ID** (Foreign Key): Linked to **Account**.
   * **Interest\_Rate**: Interest rate applicable to the savings account.
   * **Min\_Balance**: Minimum balance requirement for a savings account.
4. **Transaction**
   * **Transaction\_ID** (Primary Key): Unique identifier for each transaction.
   * **Transaction\_Type**: Type of transaction (Deposit, Withdrawal, Transfer).
   * **Amount**: Transaction amount.
   * **Date**: Date of the transaction.
   * **Account\_ID** (Foreign Key): Links back to **Account**.
5. **Deposit**
   * **Transaction\_ID** (Foreign Key): Links to **Transaction**.
   * **Deposit\_Amount**: Amount deposited.
6. **Withdrawal**
   * **Transaction\_ID** (Foreign Key): Links to **Transaction**.
   * **Withdrawal\_Amount**: Amount withdrawn.

Relationships:

* **Account** to **Current Account** or **Savings Account**: One-to-One relationship, where an account can be either of the two types.
* **Account** to **Transaction**: One-to-Many relationship, as each account can have multiple transactions.
* **Transaction** to **Deposit** or **Withdrawal**: One-to-One relationship, detailing specific information about the transaction.

Non-Functional Requirements

* **Performance**: Each action should be processed in under a second for efficient console response.
* **Usability**: Designed for easy navigation with simple commands and clear prompts.
* **Reliability**: Ensures data accuracy with transaction validation and error handling.
* **Security**: Implements access controls for account data protection and ensures secure handling of sensitive information.