**Project Report**

**Name :** Juturi Tejasri

**Email id :** juturitejasri@gmail.com

**Internship :** Java Development

**Project Title :** Snake Game in Java

**Task Number :** 01

**Sponsor :** Veritech

Language used : Java

Editor used : Java compiler editor

**Description :**

The ATM interface program will allow users to perform various banking operations such as checking balance, withdrawing money, and depositing money. Users will be prompted to enter their user ID and PIN upon startup for authentication. Upon successful authentication, users will gain access to the ATM functionalities. Error handling mechanisms will be implemented to handle invalid user input,insufficient funds, etc. The program will provide informative messages to guide users through the ATM interface.

**To create a ATM Interface in Java, you can follow these steps:**

• **Create a Java Project**: Begin by creating a new Java project in your preferred IDE or code editor.

• **Define ATM Functionalities:** Determine the functionalities to include in the ATM interface program, such as checking balance, withdrawing money, and depositing money.

**• Create User Class:** Define a user class to represent each user of the ATM. Include attributes like userID, userPIN, and accountBalance.

class User {

private String userID;

private String userPIN;

private double accountBalance;

public User(String userID, String userPIN, double accountBalance) {

this.userID = userID;

this.userPIN = userPIN

this.accountBalance = accountBalance;

}

// Getters and setters for attributes

public String getUserID() {

return userID;

}

public String getUserPIN() {

return userPIN;

}

public double getAccountBalance() {

return accountBalance;

}

public void setAccountBalance(double accountBalance) {

this.accountBalance = accountBalance;

}

}

• Create ATM Class: Define an ATM class that encapsulates ATM functionalities. Include

methods for performing operations like checking balance, withdrawing money, and

depositing money.

class ATM {

private User currentUser;

public boolean authenticateUser(String userID, String userPIN) {

// In a real-world scenario, you would validate against a database.

// For simplicity, a hardcoded user is used here.

User sampleUser = new User("12345", "6789", 1000.0);

if (userID.equals(sampleUser.getUserID()) && userPIN.equals(sampleUser.getUserPIN())) {

currentUser = sampleUser;

return true;

} else {

return false;

}

}

public double checkBalance() {

return currentUser.getAccountBalance();

}

public void withdraw(double amount) {

if (amount > currentUser.getAccountBalance()) {

System.out.println("Insufficient funds. Withdrawal canceled.");

} else {

currentUser.setAccountBalance(currentUser.getAccountBalance() - amount);

System.out.println("Withdrawal successful. Remaining balance: " + currentUser.getAccountBalance());

}

}

public void deposit(double amount) {

currentUser.setAccountBalance(currentUser.getAccountBalance() + amount);

System.out.println("Deposit successful. Updated balance: " + currentUser.getAccountBalance());

}

}

**• Implement User Authentication:** Prompt users to enter their user ID and PIN upon startup. Validate the entered credentials against stored user data.

• **Handle Input/Output:**

Use java.util.Scanner for user input and System.out.println for output

Use Java's input/output functionalities to interact with users. Display appropriate messages and prompts to guide users through the ATM interface.

class ATM {

private User current public boolean authenticateUser(String userID, String userPIN) {

// In a real-world scenario, you would validate against a database.

// For simplicity, a hardcoded user is used here.

User sampleUser = new User("12345", "6789", 1000.0);

if (userID.equals(sampleUser.getUserID()) && userPIN.equals(sampleUser.getUserPIN())) {

currentUser = sampleUser;

return true;

} else {

return false;

}

}

public double checkBalance() {

return currentUser.getAccountBalance();

}

public void withdraw(double amount) {

if (amount > currentUser.getAccountBalance()) {

System.out.println("Insufficient funds. Withdrawal canceled.");

} else {

currentUser.setAccountBalance(currentUser.getAccountBalance() - amount);

System.out.println("Withdrawal successful. Remaining balance: " + currentUser.getAccountBalance());

}

}

public void deposit(double amount) {

currentUser.setAccountBalance(currentUser.getAccountBalance() + amount);

System.out.println("Deposit successful. Updated balance: " + currentUser.getAccountBalance());

}

}

• **Error Handling:** Implement error handling mechanisms to deal with invalid user input, insufficient funds, etc. Provide informative error messages to assist users in correcting their mistakes.

**Coding**

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

class User {

private String userID;

private String userPIN;

private double accountBalance;

public User(String userID, String userPIN, double accountBalance) {

this.userID = userID;

this.userPIN = userPIN;

this.accountBalance = accountBalance;

}

public String getUserID() {

return userID;

}

public String getUserPIN() {

return userPIN;

}

public double getAccountBalance() {

return accountBalance;

}

public void deposit(double amount) {

accountBalance += amount;

}

public void withdraw(double amount) {

if (amount <= accountBalance) {

accountBalance -= amount;

} else {

System.out.println("Insufficient funds!");

}

}

}

class ATM {

private Map<String, User> userDatabase;

public ATM() {

userDatabase = new HashMap<>();

// Add some sample users for demonstration purposes

userDatabase.put("123456", new User("123456", "1234", 1000.0));

userDatabase.put("789012", new User("789012", "5678", 500.0));

}

public User authenticateUser(String userID, String userPIN) {

User user = userDatabase.get(userID);

if (user != null && user.getUserPIN().equals(userPIN)) {

return user;

} else {

return null; // Authentication failed

}

}

public void displayMenu() {

System.out.println("1. Check Balance");

System.out.println("2. Deposit Money");

System.out.println("3. Withdraw Money");

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

}

public static void main(String[] args) {

ATM atm = new ATM();

Scanner scanner = new Scanner(System.in);

System.out.println("Welcome to the ATM Interface!");

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

String userID = scanner.next();

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

String userPIN = scanner.next();

User authenticatedUser = atm.authenticateUser(userID, userPIN);

if (authenticatedUser != null) {

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

int choice;

do {

atm.displayMenu();

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

choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.println("Balance: $" + authenticatedUser.getAccountBalance());

break;

case 2:

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

double depositAmount = scanner.nextDouble();

authenticatedUser.deposit(depositAmount);

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

break;

case 3:

System.out.print("Enter withdrawal amount: $");

double withdrawalAmount = scanner.nextDouble();

authenticatedUser.withdraw(withdrawalAmount);

break;

case 4:

System.out.println("Exiting. Thank you!");

break;

default:

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

}

} while (choice != 4);

} else {

System.out.println("Authentication failed. Exiting...");

}

scanner.close();

}

**Challenges faced:**

**Security Concerns**:In a real-world scenario, you should avoid hardcoding user credentials in your code. Instead, use a database to store user information securely.

Consider implementing more secure authentication mechanisms, such as hashing user PINs.

**Error Handling**:The code currently handles insufficient funds gracefully. You might want to extend error handling for cases like invalid input during withdrawal and deposit operations (e.g., negative amounts).

**Data Persistence:** In this example, the user data is not persisted after the program execution. Consider saving user data to a file or a database so that it persists between program runs.

**Concurrency Issues**:The current implementation is not designed for concurrent access. If multiple users try to access the ATM simultaneously, you might encounter issues. Consider using synchronization mechanisms to handle concurrent access.

**Code Structure:** While the code is functional, consider breaking down the Main class into separate classes for better code organization. This can make the code more modular and easier to maintain.

**Decimal Precision:**When dealing with financial transactions, it's important to consider using BigDecimal instead of double to avoid precision issues with floating-point numbers.

**Internationalization:**The code assumes that the user inputs and system outputs are in English. If you plan to make the program accessible to users of different languages, consider internationalization (i18n) techniques.

**Unit Testing**: Consider implementing unit tests to ensure the correctness of individual methods and classes. This is especially important for a project handling financial transactions.

By addressing these considerations, you can enhance the robustness, security, and maintainability of your ATM simulation program.

**Learnings**

**Object-Oriented Design:** You've used object-oriented principles by creating separate classes for User and ATM. This promotes encapsulation and separation of concerns.

**Encapsulation:**The attributes of the User class are private, and access is provided through getters and setters. This encapsulation helps control access to the internal state of the class.

**Authentication Mechanism:**You've implemented a basic authentication mechanism in the ATM class. In a real-world scenario, this would involve more sophisticated methods and possibly database interactions.

**User Interaction:**The program effectively interacts with the user through the console. The use of the Scanner class allows input from the user, and the menu-driven interface guides users through the available options.

**Error Handling**:The code includes some basic error handling, such as handling insufficient funds during a withdrawal. It's important to consider various scenarios and edge cases to ensure the robustness of the program.

**Modularity**:The code is divided into separate methods, promoting modularity. The showMenu method, in particular, makes the main logic more readable and maintainable.

**Logical Flow:** The logical flow of the program is clear, starting with user authentication and then presenting a menu for different operations. The use of a loop ensures that the program continues running until the user decides to exit.

**Code Readability:**The code is readable and well-structured. Clear and meaningful variable/method names make it easy to understand the purpose of each component.

**User-Friendly Messages**:The program provides user-friendly messages, guiding the user through the authentication process and menu options.

As you continue working on projects, you might encounter more advanced concepts, such as database integration, security enhancements, and additional features. Keep up the good work, and feel free to explore further improvements and extensions to this project!

**Conclusion:**

The ATM class provides functionalities for checking the account balance, withdrawing funds, and depositing money. The logic includes basic error handling for insufficient funds during withdrawals.

**User Interaction**:The program interacts with users through the console, utilizing the Scanner class for input. The menu-driven interface guides users through different banking operations.

**Error Handling:**Basic error handling is implemented, addressing scenarios like insufficient funds during withdrawals. However, further enhancements could be made to handle various edge cases and invalid inputs more comprehensively.

**Readability and Structure:**The code is well-structured, with clear method names and logical flow. The showMenu method provides a structured way to present options to the user.

**Potential Improvements**:Considerations for future improvements include increased security measures, data persistence (using databases), more sophisticated error handling, and support for internationalization.

**Learning Outcomes**:The code demonstrates a good understanding of fundamental Java concepts, including classes, methods, user input/output, and basic error handling. It also showcases the ability to create a simple interactive program.