

**VaultX**

*VaultX: The Secure Digital Banking System*

**Zain Al Hassan Khawaja (FA24-BCT-047)**

**Tehreem Zahra (FA24-BCT-043)**

**Furqan Ali (FA24-BCT-017)**

****

****

****

Contents

[**Project Title: VaultX – Secure Banking System** 2](#_Toc196176526)

[**Introduction:** 2](#_Toc196176527)

[**Core Functionalities and Methods:** 2](#_Toc196176528)

[**System Design – VaultX** 3](#_Toc196176529)

[**Overview** 3](#_Toc196176530)

[**Class Structure** 3](#_Toc196176531)

[Key Components: 3](#_Toc196176532)

[**Control Structures** 3](#_Toc196176533)

[**Loops** 3](#_Toc196176534)

[**CRUD Operations in VaultX** 4](#_Toc196176535)

[Create – signUp() 4](#_Toc196176536)

[Read – login() and checkBalance() 5](#_Toc196176537)

[5](#_Toc196176538)

[Update – deposit() and withdraw() 5](#_Toc196176539)

[Delete – deleteAccount() 6](#_Toc196176540)

[Encryption & Decryption Mechanism 7](#_Toc196176541)

[🔸 encrypt(String data) 7](#_Toc196176542)

[🔸 decrypt(String data) 7](#_Toc196176543)

[**Purpose of Encryption:** 8](#_Toc196176544)

[**Coherent Design Flow** 8](#_Toc196176545)

[**Challenges and Solutions in the VaultX Project** 8](#_Toc196176546)

[**Conclusion: What I Learned from the VaultX Project** 9](#_Toc196176547)

**Project Title: VaultX – Secure Banking System**

# **Introduction:**

VaultX is a secure console-based banking system designed to simulate core banking functionalities while focusing on user data protection using basic encryption techniques. The primary goal is to allow users to safely manage their accounts through a range of banking operations such as sign-up, login, balance inquiry, deposit, and withdrawal—all protected using a Caesar cipher encryption.

# **Core Functionalities and Methods:**

* **signUp()** – Allows new users to create an account with a username and password (encrypted before saving).
* **login()** – Authenticates existing users by matching input credentials with encrypted data.
* **recoverPassword()** – Enables users to retrieve their forgotten password by verifying their username.
* **encrypt(String data)** – Encrypts strings using a Caesar cipher to secure sensitive information.
* **decrypt(String data)** – Decrypts previously encrypted strings for internal verification.
* **checkBalance(String username)** – Displays the current account balance of the logged-in user.
* **deposit(String username, double amount)** – Adds funds to the user's account and updates the balance file.
* **withdraw(String username, double amount)** – Deducts funds from the user’s account if sufficient balance is available.
* **saveUserData(String username, String encryptedPassword)** – Saves the user's login details securely in a text file.
* **saveBalance(String username, double balance)** – Stores or updates the user's current balance in a separate file.
* **readBalance(String username)** – Reads the balance from the file for the specified user.

# **System Design – VaultX**

## **Overview**

The **VaultX Secure Banking System** is structured with clarity and simplicity to support secure and smooth banking operations through the command line. It employs a single Java class, organized into logically grouped **methods** to perform key **CRUD operations (Create, Read, Update, Delete)**. The program utilizes **control structures**, **loops**, and **basic file handling** to interact with user data securely.

## **Class Structure**

* **Class Name**: VaultX
* **Purpose**: Acts as the main controller of the application, handling user authentication, encryption, and banking operations.

### ****Key Components:****

* **Static methods**: Used for ease of access from main() without needing object creation.
* **Caesar Cipher encryption**: Ensures data security using simple string manipulation.
* **Text files**: Used for persistent storage of usernames, passwords, and balances.

## **Control Structures**

VaultX uses control structures for decision-making and validation:

* **if / else if / else**: To handle conditions like invalid login, insufficient balance, existing users, etc.
* **switch-case**: Used in the main menu to execute the selected option.
* **try-catch (optional if needed)**: Not used here, as per the course preference of no exception handling.

## **Loops**

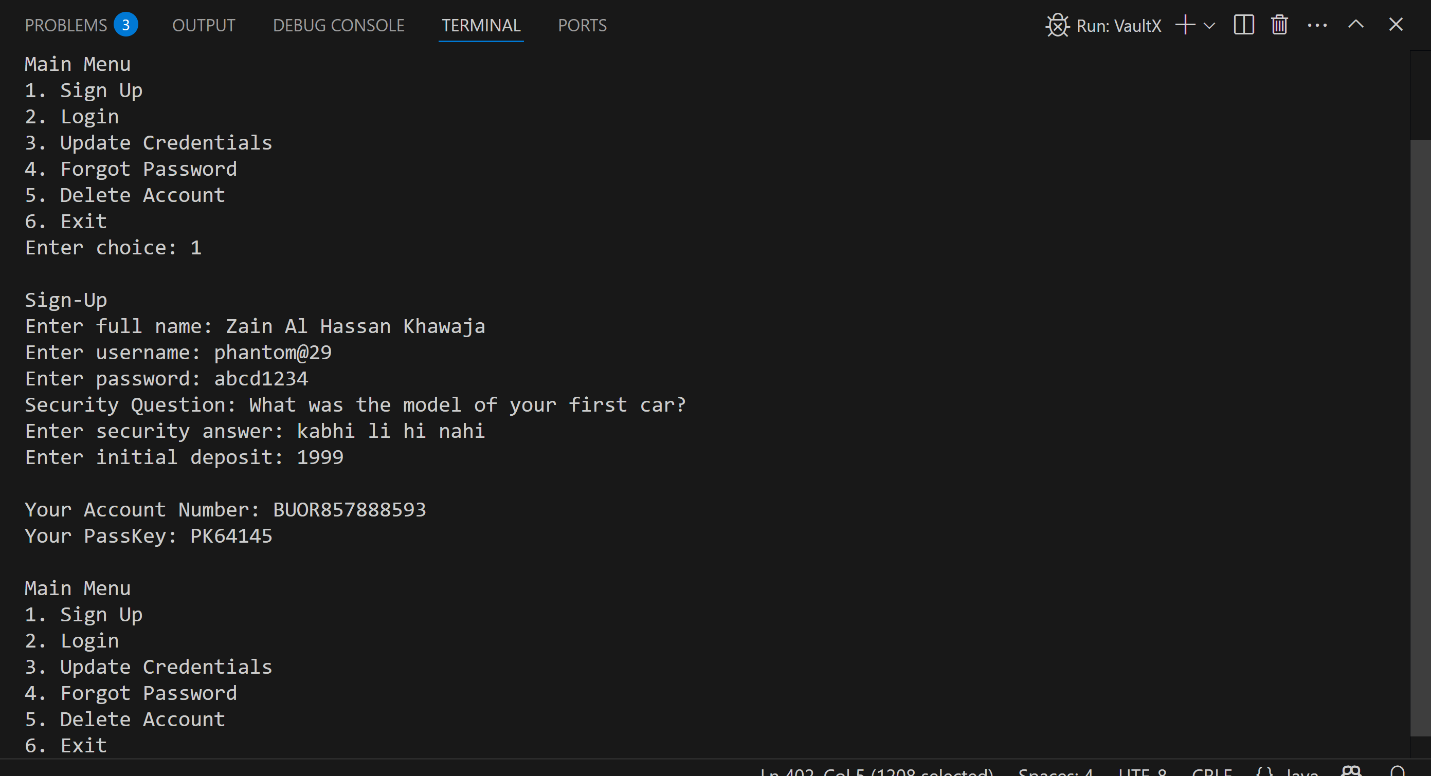
* **while loop**: Keeps the main menu running until the user chooses to exit.
* **for loop** (or while with condition): Used to read through stored file lines while verifying login credentials or checking username existence.

# **CRUD Operations in VaultX**

VaultX demonstrates the core **CRUD** (Create, Read, Update, Delete) principles using secure, memory-based Java methods and encrypted data storage. Each operation is implemented through dedicated functions ensuring user authentication and secure handling of data.

### Create – signUp()

* **Purpose:** Registers a new user by collecting and encrypting personal and banking details.
* **Process:**
  + Prompts for name, username, password, security question, answer, account number, passkey, and initial balance.
  + Encrypts and stores each detail in memory using the Caesar cipher.
* **Security:** Validates non-empty input; encrypts data immediately.



### Read – login() and checkBalance()

* **Purpose:**
  + The login() method authenticates the user and allows access to their account.
  + The checkBalance() method displays the current balance of the logged-in user.
* **Process:**
  + **login()**:
    - Prompts the user for their username and password.
    - Compares the entered credentials with the encrypted stored values.
    - If the credentials match, the user is logged in.
    - If unsuccessful, it prevents further access to the account.
  + **checkBalance()**:
    - Ensures the user is logged in by checking if the encryptedUsername is set.
    - If logged in, prints the balance stored in the encryptedBalance variable.
    - If not logged in, prevents access to the balance and asks for login.
* **Security:**
  + Both operations require user authentication. Only authenticated users can access account details, ensuring data security.

### 

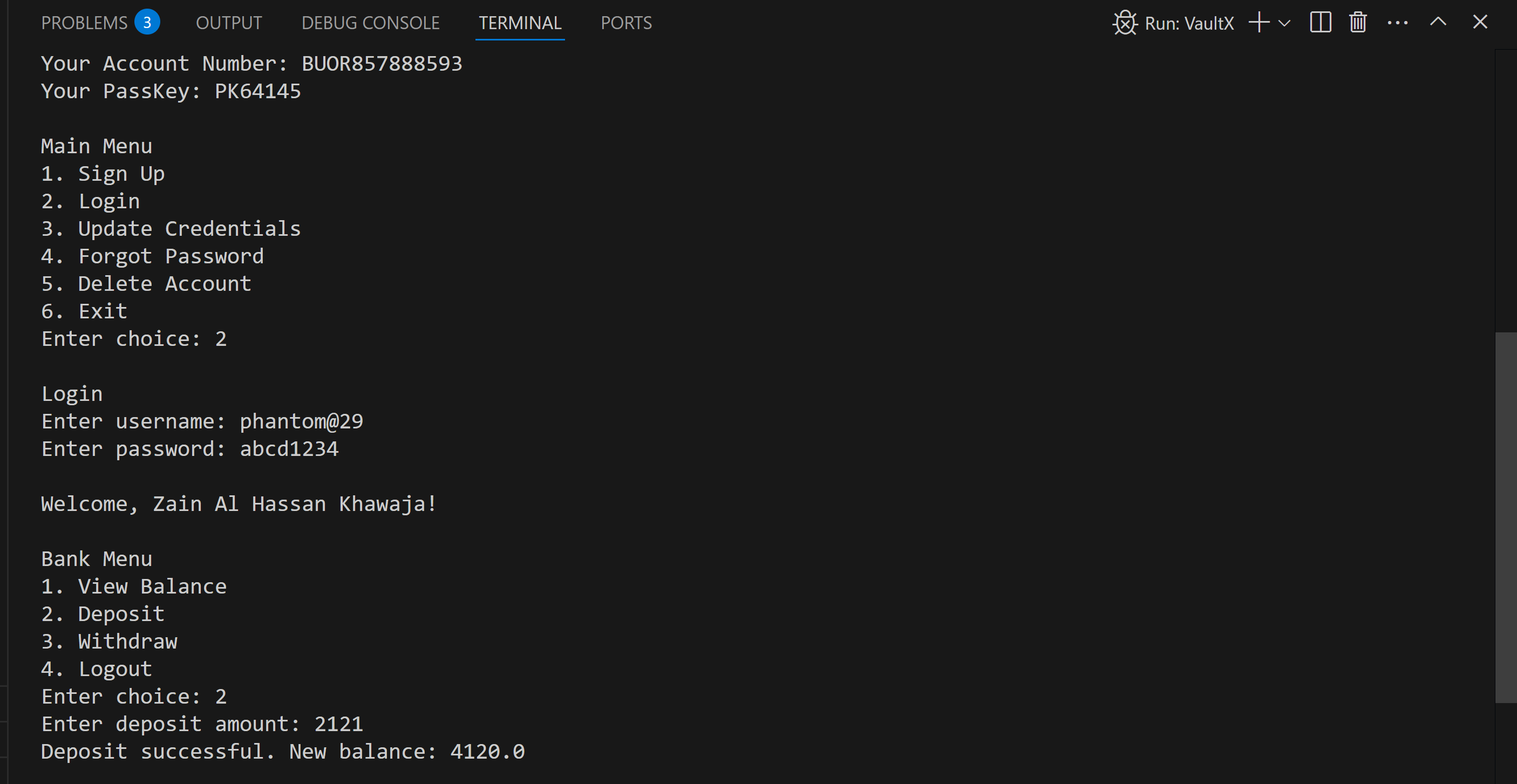
### Update – deposit() and withdraw()

#### **deposit()**

* **Purpose:** Allows user to add funds to their account.
* **Process:**
  + Prompts for deposit amount.
  + Validates that it's positive.
  + Updates the balance variable accordingly.

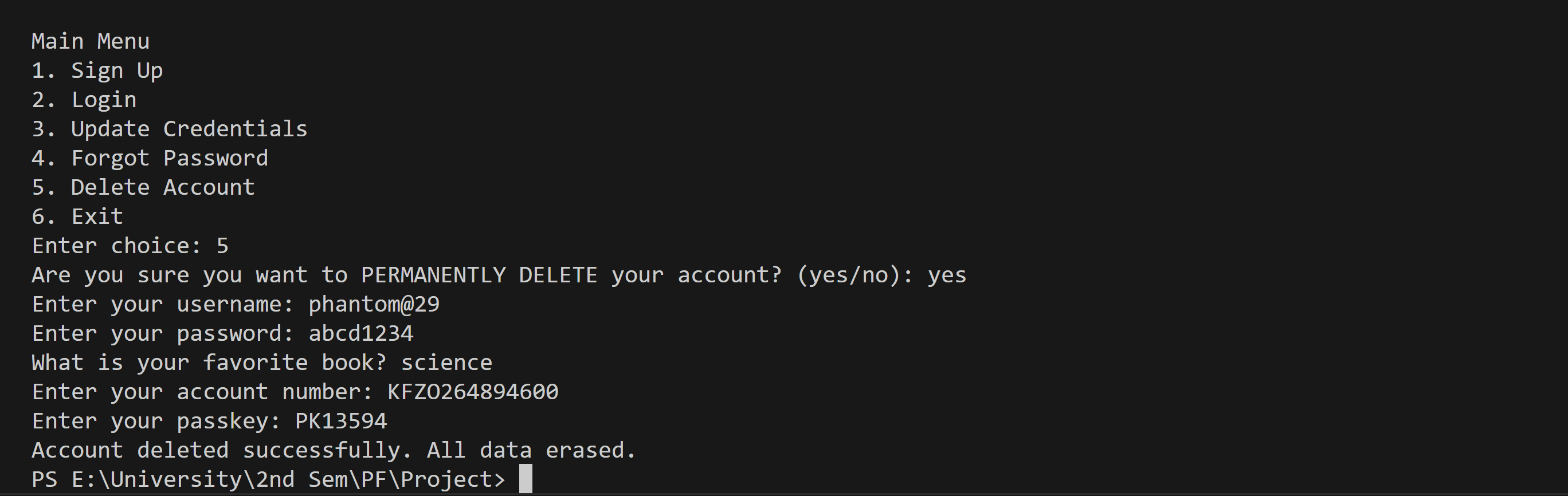
**withdraw()**

* **Purpose:** Allows user to withdraw funds.
* **Process:**
  + Prompts for withdrawal amount.
  + Validates that balance is sufficient.
  + Updates the balance variable if conditions are met.
* **Security:** Both require user to be logged in; all data is encrypted.



### Delete – deleteAccount()

* **Purpose:** Permanently deletes a user's account from memory.
* **Process:**
  + Confirms user intention.
  + Requests multiple credentials:
    - Username
    - Password
    - Security answer
    - Account number
    - Passkey
  + Verifies each using encrypted comparisons.
  + If verified, all user data variables (encryptedUsername, encryptedPassword, etc.) are reset to null or 0.0.
* **Security:** Uses **multi-factor verification** to ensure only the correct user can delete the account.



# 

# Encryption & Decryption Mechanism

VaultX uses a **custom Caesar Cipher-based encryption technique** to protect user data.

### 🔸 encrypt(String data)

* Each character in the string is shifted by a fixed number of positions (e.g., +3).
* For example:
  + A → D, b → e, 1 → 4
* Used for encrypting all sensitive user details before storing in memory.

### 🔸 decrypt(String data)

* Reverses the Caesar shift by the same amount (e.g., -3).
* For example:
  + D → A, e → b, 4 → 1
* Used internally when recovering or verifying user data.

// This method encrypts the input string using a simple Caesar cipher (shifts letters and digits by +3)

    private static String encrypt(String input) {

        String encrypted = "";  // Will hold the encrypted result

        // Loop through each character in the input string

        for (int i = 0; i < input.length(); i++) {

            char c = input.charAt(i);  // Get the current character

            // Encrypt uppercase letters (A-Z)

            if (c >= 'A' && c <= 'Z') {

                // Shift by 3 within the alphabet, wrapping around if needed

                encrypted += (char) ('A' + (c - 'A' + 3) % 26);

            }

            // Encrypt lowercase letters (a-z)

            else if (c >= 'a' && c <= 'z') {

                // Shift by 3 within the lowercase alphabet

                encrypted += (char) ('a' + (c - 'a' + 3) % 26);

            }

            // Encrypt digits (0-9)

            else if (c >= '0' && c <= '9') {

                // Shift by 3 within 0-9 range

                encrypted += (char) ('0' + (c - '0' + 3) % 10);

            }

            else {

                encrypted += c;

            }

        }

        return encrypted;

    }

## **Purpose of Encryption:**

* Prevents plain-text storage of critical information like passwords or account numbers.
* Adds a layer of **obfuscation**, making it difficult for unauthorized access even if memory is inspected.

# **Coherent Design Flow**

1. **Main Menu**:
   * Displays options: Sign Up, Login, Recover Password, Exit.
   * Loops until the user exits.
2. **User Authentication**:
   * Sign Up stores encrypted credentials.
   * Login decrypts and verifies credentials.
3. **Banking Interface (Post-login)**:
   * Options: Check Balance, Deposit, Withdraw, Logout.
   * File I/O ensures persistent state across sessions.

# 

# **Challenges and Solutions in the VaultX Project**

1. **User Authentication and Validation**
   * **Challenge:** Ensuring proper authentication before allowing sensitive operations.
   * **Solution:** Implemented login() function to verify credentials (username, password, security question, etc.) using encrypted data.
2. **Encryption and Decryption**
   * **Challenge:** Secure storage and handling of sensitive data like passwords and balances.
   * **Solution:** Used simple encryption (Caesar Cipher) to store encrypted data and decrypted it when needed, preventing plaintext exposure.
3. **File Operations for Data Persistence**
   * **Challenge:** Safe handling of user data in text files to prevent corruption or accidental loss.
   * **Solution:** Used temporary files during updates (e.g., account deletion) to ensure safe and consistent data manipulation.
4. **Account Deletion**
   * **Challenge:** Deleting an account while verifying user credentials and maintaining data integrity.
   * **Solution:** Implemented a multi-step process: verified user’s identity and reset all fields to null, then updated the files using temporary files.
5. **User Experience**
   * **Challenge:** Ensuring security without making the process overly complex for users.
   * **Solution:** Clear, concise prompts for verification and simple steps to guide users through sensitive operations like account deletion.
6. **File-Based Data Security**
   * **Challenge:** Protecting sensitive data stored in text files.
   * **Solution:** Stored encrypted data in files, ensuring files were updated safely and not exposed to unauthorized access.
7. **Preventing Unauthorized Access**
   * **Challenge:** Preventing unauthorized actions (e.g., balance check, account deletion).
   * **Solution:** Implemented login checks and multi-factor authentication for sensitive actions to ensure only authorized users can perform them.

# 

# **Conclusion: What I Learned from the VaultX Project**

The VaultX project taught me several key lessons about secure software design and user data management:

1. **Security and Encryption**: I learned the importance of securing sensitive data (passwords, balances) using basic encryption techniques (Caesar cipher). This was a fundamental step in ensuring data protection within the application.
2. **User Authentication**: The project emphasized the need for multiple layers of verification (username, password, security question, etc.) to prevent unauthorized access to accounts.
3. **File Handling**: I gained experience in managing user data in text files, ensuring data integrity while performing operations like updates or deletions.
4. **Account Deletion**: I learned the importance of providing clear confirmation prompts and proper authentication before allowing sensitive operations, such as account deletion.
5. **User Experience**: Creating clear prompts and feedback for users helped me understand how important it is to maintain a smooth, intuitive interface when performing complex tasks.
6. **Data Validation**: I recognized how essential it is to validate user inputs to maintain system integrity and prevent errors.
7. **Basic Encryption**: Even simple encryption methods can provide a level of security, reinforcing the importance of protecting user data.