Project 1: User Access Manager

CMSC 204

Fall 2025

Assignment Overview

In this project, you will build a basic access control system that manages user accounts and enforces access control rules. The system uses a `List<UserAccount>` to store and manage all user records. It supports loading existing users from a file and allows commands to add, remove, and verify users interactively. If access is denied multiple times, the account is locked.

You will implement appropriate exception handling, track failed attempts, and provide meaningful output for both successful and error cases.

Learning Objectives

By completing this project, you will:

- Implement and use custom exception types to handle specific error conditions
- Use static utility methods for string hashing and encryption
- Model real-world password security protocols

Project Requirements

To receive full credit, your submission must meet the following technical requirements:

- **UserAccount class**:
  - Implements: `equals()`, and `toString()`
  - Methods to get an encrypted password, increment/reset failure count and check lock status
- **UserAccessManager class**:
  - Field: `List<UserAccount>` to store accounts
  - Methods: `addUser`, `removeUser`, `verifyAccess`, `loadAccounts`
- **Exception Handling**:
  - Define and use:
    - `DuplicateUserException`
    - `UserNotFoundException`
    - `InvalidCommandException`
    - `AccountLockedException`
    - `PasswordIncorrectException`
  - Provide meaningful error messages without stack traces
- **Utilities**
- The following methods are provided in a separate `Utilities.java` file:

  - `encryptPassword`
  - `readAccountFile`

- **Console Interaction**:
  - Support the following commands: `add`, `remove`, `verify`, `load`, `exit`
  - Provide clear, user-facing output and handle invalid input gracefully
- **Password Security**:
  - Store only encrypted password strings
  - Compare passwords by hashing input with `encryptPassword` and comparing to stored values
- **Code Quality**:
  - Use clear formatting and inline comments
  - Avoid duplication and maintain consistent naming conventions

Classes and Responsibilities

UserAccount class

Represents a user.

Fields:

- `String username`
- `String encryptedPassword`
  (store only the hashed version of the password)
- `int failureCount` (starts at 0)
- `boolean locked` (starts as `false`)

Behavior:

- Methods to:
  - increment failure count
  - reset failure count
  - check if locked
  - `checkPassword(String password) throws AccountLockedException, PasswordIncorrectException`

    checkPassword Behavior:
    * If account is locked: throw `AccountLockedException`
    * If `password` parameter does not match the user's `encryptedPassword`: throw `PasswordIncorrectException`
    * Otherwise, return `true`
- Overrides `equals()` and `toString()`

  equals(): Two `UserAccount` objects are considered equal if they have the same username. Password, lock status, and failure count are not considered.

  toString(): The `toString()` method should return the username as a simple string (e.g., `"alice"`).

UserAccessManager class

Holds the list of accounts and performs actions.

Fields:

- `List<UserAccount> accounts`

Methods:

- `void loadAccounts(String filename) throws FileNotFoundException`
- `void addUser(String username, String encryptedPassword) throws DuplicateUserException, InvalidCommandException`
- `void removeUser(String username) throws UserNotFoundException, InvalidCommandException`

  Note: In `addUser()` and `removeUser()` if either `username` or `encryptedPassword` is null or empty, throws `InvalidCommandException`.
- `boolean verifyAccess(String username, String encryptedPassword) throws UserNotFoundException , AccountLockedException, . InvalidCommandException`

Verification Behavior:

- if username is empty: throw `InvalidCommandException`
- If username not found: throw `UserNotFoundException`
- If account is locked: throw `AccountLockedException`
- If `encryptedPassword` does not match the user's `encryptedPassword`:
  - Increment failure count
  - Lock account if 3 or more failures
  - throw `PasswordIncorrectException`
- If user exists, the password is correct and account not locked:
  - Reset failure count
  - Return `true`

Custom Exceptions

You must define the following custom exceptions:

- `DuplicateUserException`
- `UserNotFoundException`
- `InvalidCommandException`
- `AccountLockedException`
- `PasswordIncorrectException`

All exceptions must include meaningful messages and **must not** result in stack traces.

Accounts File and Encryption Scheme

User Accounts File Format

The system loads user account information from a text file provided as a command-line argument. Each line of this file must contain a single user record with the following format:

```
username encryptedPassword
```

The `username` is a string with no whitespace, and the `encryptedPassword` is a SHA-256 hash of the user's password in lowercase hexadecimal format. Lines are separated by newline characters. Any line that is blank or does not contain exactly two space-separated tokens is ignored. An example `accounts.txt` file is provided as part of the project files.

Example:

```
alice ef92b778bafe771e89245b89ecbc08a44a4e166c06659911881f383d4473e94f
bob   f53b9d4e1d98b9cc2c7403f0f64c5e8cd0f27c9287b6feff7750fcd4e0f5b1e8
charlie 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
```

These hashed values must already be generated using the provided encryption method. Plaintext passwords should never appear in the file.

Password Encryption Scheme

To securely store and verify user passwords, the system uses a hashing function based on the SHA-256 algorithm. Passwords are not stored in plaintext. Instead, the following method is used to encrypt passwords:

- The password string is encoded in UTF-8.
- It is hashed using the SHA-256 algorithm.
- The resulting byte array is converted to a lowercase hexadecimal string.

This hash is used for all password comparisons. Note that no salt is used in this scheme, and the same input password will always produce the same hash.
A static method `encryptPassword(String password)` is provided in the `Utilities` class to perform this transformation.

Why SHA-256?

Overview of SHA-256 SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function developed by the National Security Agency (NSA) and published by NIST as part of the SHA-2 family. It transforms any input string (such as a password) into a fixed-length 256-bit hash, represented as a 64-character hexadecimal number.

A key property of SHA-256 is that it is one-way: it is computationally infeasible to reverse the hash to discover the original input. Even a small change in the input (such as changing one character) results in a completely different output hash.

Why We Use It In this project, we use SHA-256 for the following reasons:

- **Security:** SHA-256 is considered secure against known attacks and is widely trusted in industry.
- **One-way function:** Passwords are not stored in plaintext. Even if someone gains access to the account file, they cannot easily recover the original passwords.
- **Consistency:** The same input always produces the same output, which is important for verifying password input against stored hashes.
- **Availability:** SHA-256 is built into the Java standard library via the `MessageDigest` class.

Limitations

While SHA-256 is a secure hash function, it is not specifically designed for password hashing. For high-security applications, functions like bcrypt, scrypt, or Argon2 are preferred because they incorporate salting and are intentionally slow to resist brute-force attacks.

However, for this educational project, SHA-256 provides a good balance of simplicity and security, and illustrates key principles of credential protection.

Input Specification

The program accepts one command and its arguments on each line. The valid commands are:

- add [username] - prompts for a password and then adds that user account to the list.
- remove [username] - attempts to remove username from the list of accounts
- verify [username] - prompts for a passwords and then validates the user's access to the system
- load [filename] - loads user account information from the specified filename
- exit - quits the system

All commands are case-sensitive and must include non-empty argument values. If a command includes an empty username or password, the system must throw `InvalidCommandException`.

`load` command details

The `load` command expects a filename as a command-line argument. This file will contain a list of users and passwords, one per line:

```
alice ef92b778ba...
bob 2c9341ca4c...
charlie 5f4dcc3b5a...
```

Each line contains a username and an encrypted password, separated by whitespace.

Sample Session

```
$ java UserAccessManager
User access manager ready.
User Access Manager> load accounts.txt
Unable to load file: accounts.txt
User Access Manager> add alice
Password: password
User Access Manager> add bob
Password: abc123
User Access Manager> add bob
Password: Hard*Password_1
User 'bob' account already exists.
User Access Manager> verify alice
Password: test
Incorrect Password
User Access Manager> verify alice
Password: testing
Incorrect Password
User Access Manager> verify alice
Password: another_test
User 'alice' account is locked.
User Access Manager> verify alice
```

```
Password: foo
User 'alice' account is locked.
User Access Manager> exit

Process finished with exit code 0
```

Output Format

All output must match the examples above. Messages should be clear and must identify the exception type when errors occur. Stack traces must not be printed.

Constraints and Notes

- Store all users in an `List<UserAccount>`. `List` is an `Abstract` type, you are free to use any of the subclasses of `List`. Do **not** use other `Collection` types.
- On failed access attempts:
  - Failure count increases
  - If it reaches 3, account is locked
- On successful access:
  - Failure count is reset to 0, unless the account is already locked
- Locked accounts cannot be accessed even with correct credentials
- Assume all commands will have the correct number of arguments, but check that values are non-empty
- Invalid or malformed commands should throw `InvalidCommandException`

Testing Notes

The provided GFA (Good Faith Attempt) JUnit tests represent the minimum requirements to validate the basic functionality of your project. If the project due date has passed, your submission must pass these tests in order to be considered for course credit.

In addition to the GFA tests, any GUI code and public JUnit tests are provided to assist with basic verification of your implementation. However, these do not cover all functionality. You are expected to design and run additional tests to ensure the full correctness and robustness of your project.

Student Tests

You must write a suite of JUnit tests that thoroughly verify your implementation. Your tests should include:

- Adding new users with valid usernames and passwords
- Verifying access for existing users with correct and incorrect passwords
- Ensuring that duplicate usernames are not accepted and trigger an appropriate exception
- Removing users and confirming they are no longer accessible afterward
- Triggering account lockout by providing incorrect passwords multiple times and verifying that access is denied thereafter
- Handling edge cases, such as:
  - Verifying or removing nonexistent users
  - Adding or verifying users with empty strings or null inputs

    - Attempting actions after an account has been locked

- Ensuring that the toString method returns the expected string format
- Testing proper exception handling for all required custom exceptions

Running the Provided JUnit Tests

To verify that your implementation meets the project requirements, we have provided a set of automated JUnit test cases. These include:

- `UserAccessManagerPublicTest.java`
- `UserAccessManagerGFATest.java`
- `CommandLineIntegrationTests.java`

These tests cover both functional correctness and integration behavior. You are expected to pass all tests prior to submission.

Using Eclipse (Recommended)

To run the tests in Eclipse:

1. Make sure your project uses a Java JDK (not a JRE).
2. Right-click your project folder in the **Package Explorer**.
3. Choose **Build Path** → **Configure Build Path**, and ensure that the `junit-5` library is included under **Libraries**. If not, click **Add Library** → **JUnit** → **JUnit 5**.
4. Make sure the test files (`UserAccessManagerPublicTest.java`, etc.) are in the `src` folder or a designated `test` folder on the build path.
5. To run a test class:
    (a) Right-click the file (e.g., `UserAccessManagerPublicTest.java`) in the Package Explorer.
    (b) Select **Run As** →**JUnit Test**.
6. Use the **JUnit tab** to see which tests passed or failed. Double-click a failing test to jump to the relevant line.

Test Coverage Notes

- The `UserAccessManagerGFATest` performs minimal testing to qualify the submission as a Good Faith Attempt.
- The `UserAccessManagerPublicTest.` focuses on basic expected behavior.
- The `CommandLineIntegrationTests` simulate realistic user input and output flow.

If you are failing tests, read the output messages carefully and verify that your logic matches the specification exactly. You are encouraged to write additional tests as you develop your solution.

Deliverables

Before submitting your project, ensure your code is free of syntax errors. Submissions that do not compile will receive zero points.

Design Documents UML and/or Pseudo-Code

Implementation Only submit files that you have created or modified, do not submit unmodified files that were provided in the project download. Place all your .java files inside a src folder. Include the entire doc folder with Javadoc for your own classes.

Summary Write-Up A 2-3 paragraph write-up (eg. LearningExperience.doc)

Submission Packaging

You will submit two compressed `.zip` files:

Main Project Files All student created or modified project files and data
  Filename: `LastNameFirstName_AssignmentX.zip`

- **src/** directory containing `.java` files created or modified by the student
- **doc/** directory containing student created Javadoc files
- **LearningExperience.doc** reflection and write-up
- **Design Documents** all design related documents

MOSS files Only the student created or modified source code files
  Filename: `LastNameFirstName_AssignmentX_Moss.zip`

- **source files** only the `.java` files created or modified by the student

Grading Rubric

Criteria	Points
Exception Implemention	40%
`UserAccount` Implementation	30%
`UserAccessManager` Implementation	30%
Student JUnit tests	-5%
Code Quality, Style	-5%
Doumentation (Javadoc, Design & Write-up)	-10%

Provided Method: `readAccountFile`

To simplify file input, the a static method to read a text file containing account information is provided for you. This method reads usernames and encrypted passwords from a file and adds them to the user access manager provided as a parameter using `addUser`.

This method is contained in Utilities.java and must be imported/referenced from there
Notes:

- Each line of the file should contain a username and an encrypted password, separated by whitespace.
- Blank lines are ignored.
- If a username is already in the list, a warning is printed and the program continues.

Provided Method: `encryptPassword`

You are also provided with a helper method that generates an encrypted version of a password. It accepts as input any string, and then encrypts for use throughout the project. This is similar to standard Unix password hashing, but simplified — it uses SHA-256 and ignores salt.

This method is contained in Utilities.java and must be imported/referenced from there
Notes:

- Only encrypted passwords are stored in the system.
- When verifying access, encrypt the input password using this method and compare it to the stored encrypted password.
- This method returns a hexadecimal SHA-256 hash of the input.
- No salt is used.