# Assignment #8

## 1. Write a program to simulate user login.

### User Class:

**public** **class** User {

/\*\*

\* Setting empty username field.

\* Setting default password field.

\* Setting empty role field.

\*/

**private** String username;

**private** Password password;

**private** String role;

/\*\*

\* User constructor with default password.

\* **@param** username username as String

\* **@param** role role as String

\* new Password().getPassword() returns the default password from Password Class

\*/

**public** User(String username, String role) {

**this**(username, role, **new** Password().getPassword());

}

/\*\*

\* User constructor with modified password

\* **@param** username username as String

\* **@param** role role as String

\* **@param** pass password as String

\*/

**public** User(String username, String role, String pass) {

**this**.password = **new** Password(pass);

setUsername(username);

setRole(role);

}

/\*\*

\* Username getter

\* **@return** username as String

\*/

**public** String getUsername() {

**return** **this**.username;

}

/\*\*

\* Username setter

\* **@param** username username as String

\*/

**public** **void** setUsername(String username){

**this**.username = username;

}

/\*\*

\* Role getter

\* **@return** role as String

\*/

**public** String getRole() {

**return** **this**.role;

}

/\*\*

\* Role setter

\* **@param** role role as String

\*/

**public** **void** setRole(String role){

**this**.role = role;

}

/\*\*

\* Password getter

\* **@return** User password as String

\*/

**public** String getPassword(){

**return** password.getPassword();

}

}

### Password class:

**public** **class** Password {

/\*\*

\* Setting empty password field.

\*/

**private** String password = "Welcome1";

/\*\*

\* Default Password constructor

\*/

**public** Password() {}

/\*\*

\* Setting password.

\* **@param** password Password as string

\*/

**public** Password (String password) {

setPassword(password);

}

/\*\*

\* Password setter. Sets password if complexity checks are met.

\* **@param** password Password as string

\*/

**private** **void** setPassword(String password) {

**if** (isPasswordValid()) {

**this**.password = password;

}

}

/\*\*

\* Password getter.

\* **@return** User password

\*/

**public** String getPassword() {

**return** **this**.password;

}

/\*\*

\* Breaks password string to character array

\* **@param** password Password as string

\* **@return** Array of characters

\*/

**private** **char**[] PasswordToChar(String password) {

**return** password.toCharArray();

}

/\*\*

\* Checks if password is valid, meeting all password complexity

\* **@return** boolean true or false

\*/

**private** **boolean** isPasswordValid() {

**if** (isPasswordLenght(password) && hasPasswordNumber(password) &&

hasPasswordCapital(password)) {

**return** **true**;

}

**return** **false**;

}

/\*\*

\* Checks password length

\* **@param** password Password as string

\* **@return** boolean true or false

\*/

**private** **boolean** isPasswordLenght(String password) {

**if** (password.length() >= 8) {

**return** **true**;

}

**return** **false**;

}

/\*\*

\* Checks if password has a Number

\* **@param** password Password as string

\* **@return** boolean true or false

\*/

**private** **boolean** hasPasswordNumber(String password) {

**for** (Character c : PasswordToChar(password)) {

**if** (Character.*isDigit*(c)) {

**return** **true**;

}

}

**return** **false**;

}

/\*\*

\* Checks password has a Capital letter

\* **@param** password Password as string

\* **@return** boolean true or false

\*/

**private** **boolean** hasPasswordCapital(String password) {

**for** (Character c : PasswordToChar(password)) {

**if** (Character.*isUpperCase*(c)) {

**return** **true**;

}

}

**return** **false**;

}

}

### Main class:

**import** java.nio.file.AccessDeniedException;

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.Map.Entry;

**import** java.util.Scanner;

**public** **class** TestCase {

**public** **static** HashMap<User, Password> *userMap* = **new** HashMap<User, Password>();

**public** **static** **void** main(String[] args) **throws** AccessDeniedException {

// Instantiating default password to use

Password pass = **new** Password();

// Instantiating user objects

User user = **new** User("Bob", "user");

User user2 = **new** User("Jim", "user");

User user3 = **new** User("Liz", "super user");

// Adding users to map

*userMap*.put(user, pass);

*userMap*.put(user2, **new** Password("SuperMegaPassw0rd"));

*userMap*.put(user3, **new** Password("JavaIsTheBest1"));

// Ask the user for username and password and verifies if they are correct

**while** (**true**) {

System.***out***.println("Enter user name:");

Scanner scanner = **new** Scanner(System.***in***);

String input = scanner.next();

// Output username and password to Array

String[] userPass = *FindUserAndPassword*(input);

// Verify if the username is in the array on position 0.

// If not ask again, else fall through

**if** (!*VerifyUserExists*(userPass)) {

System.***out***.println("User does not exist.\n");

**continue**;

}

// Test if the password is correct.

// If password match output to screen and exit

// Else throw an exception and display Access Denied message.

**try** {

System.***out***.println("Enter password:");

input = scanner.next();

**if** (*VerifyPassword*(userPass, input)) {

System.***out***.println("Welcome, " + userPass[0] + "!");

}

**break**;

}

**catch** (AccessDeniedException e) {

System.***out***.println("Access Denied!");

**break**;

}

}

}

/\*\*

\* Check if the user specified exists in the HashMap

\* **@param** input input as Array. username is string and is first element in Array

\* **@return** true if user exists. false if user does not exist.

\*/

**public** **static** **boolean** VerifyUserExists(String[] input) {

**if** (input[0] == **null**) {

**return** **false**;

}

**else** {

**return** **true**;

}

}

/\*\*

\* Verifies if the user password is correct

\* **@param** input input as Array. Password is string and is second element in Array.

\* **@param** pass password as String. Entered by user.

\* **@return** true if passwords match

\* **@throws** AccessDeniedException if passwords do not match an exception is thrown.

\*/

**public** **static** **boolean** VerifyPassword(String[] input, String pass) **throws** AccessDeniedException {

**if** (input[1].equals(pass)) {

**return** **true**;

}

**else** {

**throw** **new** AccessDeniedException(**null**);

}

}

/\*\*

\* Method to check if user exists in the HashMap

\* **@param** user username as String from user input

\* **@return** array of username and password as string

\*/

**public** **static** String[] FindUserAndPassword(String user) {

String[] userAndPass = **new** String[2];

// Iterate through the map using Entry Set

Iterator<Entry<User, Password>> entrySetIterator = *userMap*.entrySet().iterator();

**while** (entrySetIterator.hasNext()) {

Entry entry = entrySetIterator.next();

// Type casting needed for the User and Password objects

User u = (User)entry.getKey();

Password p = (Password)entry.getValue();

**if** (user.toLowerCase().equals(u.getUsername().toLowerCase())) {

userAndPass[0] = u.getUsername();

userAndPass[1] = p.getPassword();

}

}

**return** userAndPass;

}

}

## 2. Factorial recursive function

**public** **class** Factorial {

**public** **static** **void** main(String[] args) {

// Factorial can only be calculated on positive integers

System.***out***.println(*Factorial*(0));

}

**public** **static** **int** Factorial(**int** i) {

// If the number we are trying to find the factorial is 0 or 1, factorial is always 1

**if** (i == 0 || i == 1) {

**return** 1;

} **else** {

// Calculate factorial by multiplying the number to the one before it

**return** i \* *Factorial*(i - 1);

}

}

}

## 3. Explain the difference between stack and heap

Stack and heap are both stored in the RAM. The stack has a memory space reserved for the process by the OS. The heap has a memory space managed by the OS and is shared among other processes. Used to get additional space at runtime. Memory space can change the size of for the heap. Memory space designated for the stack cannot be changed once created. Each method has its own stack. A method’s stack exists only during the lifetime of the method, from the calling of the method until the return. Most of the time the stack is used to store methods, variables, either local to the method variables and primitive type variables. There is an application stack that holds all the method stacks. The heap holds objects that are either shared amongst methods or are needed for a longer period of time. The stack only holds references to the objects in the heap. The methods are popped out of the stack as soon as they finish execution. And all the local variables, arguments and object references that have been part of the method are destroyed too. Objects in the heap are left in the heap as long as there are references in the stack, pointing to them. When no pointers exist for a particular object in the heap it is marked for deletion and is taken care of by the garbage collector.

Primitive types that are stored in the stack are: Boolean, short, byte, char, int, long, float. Primitive fields are stored in the heap as part of the object.   
Strings are immutable and are stored in the heap too.

There are exclusions of cores, where objects can be stored in the stack, instead of the heap. This is controlled by the Java Virtual Machine. If an object is created inside a method without using the keyword “new”, it will be placed in the stack, because the object is short lived and will be deleted as soon as the method is discarded.

If there is no enough space on the stack to handle the memory assigned to it the stack overflow exception occurs. It usually happens when a lot of nested functions are being called, or because of an infinite recursive function.   
The heap can resize the memory it has allocated.

There is a potential issue with the heap and it is caused by freeing up chunks of memory. The memory can get fragmented. When excessive fragmentation occurs new memory might not be allocated, even if there is enough memory available, because there is no single big block of memory available.

The way the stack is built it cannot get fragmented. The stack is quicker than the heap, because of the way it works, and that it does not have to reallocate memory.