Design Explanation

Batuhan BASAK

1 SystemSecurity Class

The SystemSecurity is an API that contains all the implementations. You can call SystemSecurity by two constructor methods. While the first constructor only takes username, and two passwords the other one take denomination numbers as parameter also.

```
// This is the first constructor.
SecuritySystem s1 = new SecuritySystem("sibelgulmez", "[rac()ecar]", 74);

// This is the second constructor.
// It also takes denomination numbers as last parameter.
SecuritySystem s12 = new SecuritySystem("sibel", "[rac()ecar]", 37, denominations);
```

The constructor methods do two things. First sets the username, passwords, and denominations. If denominations are not given as parameter it uses the default denomination numbers which are [4, 17, 29]. And then invokes a method called checkUserValid that checks whether all the credentials are satified or not. The denomination numbers are not stored as data field, details will be explained later. Thus there is an overloaded version of checkUserValid which takes denomination numbers as parameter also.

```
/**
 * Default constructor of class SecuritySystem.
 * Sets data fields and checks whether they are valid or not.
 * @param username The username.
 * @param password1 The first password.
 * @param password2 The second password.
 */
public SecuritySystem(String username, String password1, int password2) {
    this.username = username;
    this.password1 = password1;
    this.password2 = password2;
    checkUserValid();
}
```

```
/**
 * The constructor class which takes denomination numbers.
 * @param username The username.
 * @param password1 The first password.
 * @param password2 The second password.
 * @param denominations The denomination numbers.
 */
public SecuritySystem(String username, String password1,int password2,
                      int [] denominations){
    this.username = username;
    this.password1 = password1;
    this.password2 = password2;
    checkUserValid(denominations);
}
```

Because there are too many conditions to check, making entire conditions in one class is cumbersome. To make things easier to implement there are class in SystemSecurity class. Each class checks information. The information that username, password1, and password2. These inner classes are private and statics. Because they are private user can not access them, there is no reason for that, and because these are static they can not mutate or access from outer class. That makes easier to divide this big problem into smaller pieces without worrying about side effects.
Each instances are initialized in method checkUserValid and the methods exist in those classes are used respectively to check each condition.

```
/**
 * Checks whether user valid or not respect to the default denomination numbers.
 */
private void checkUserValid(){
    // Check whether username is valid or not.
    UsernameValidation usernameValidation = new UsernameValidation();
    if (!usernameValidation.checkIfValidUsername(username)){
        return ;
    }

    // Check whether password1 is valid or not.
    Password1Validation password1Validation = new Password1Validation();
    if (!password1Validation.checkAlphaCharactersCount(password1)){
        System.out.println("The password1 is invalid. It should have letters too.");
        return;
    }
    if (password1.length() < 8){
        System.out.println("The password1 is invalid. " +
            "It should have at least 8 characters.");
        return ;
    }
```

```
        if (!password1Validation.containsUserNameSpirit(username, password1)){
            System.out.println("The password1 is invalid." +
                "It should have at least 1 character from the username.");
            return;
        }
        if (!password1Validation.isBalancedPassword(password1)){
            System.out.println("The password1 is invalid. It should be balanced.");
            return;
        }
        if (!password1Validation.isPalindromePossible(password1)){
            System.out.println("The password1 is invalid." +
                It should be possible to obtain a palindrome from the password1.");
            return;
        }
        if (password1Validation.countBracketsCount(password1) < 2){
            System.out.println("The password1 is invalid. " + "
                It should have at least 2 brackets.");
            return ;
        }


        // Check whether password1 is valid or not.
        Password2Validation password2Validation = new Password2Validation();
        if (!password2Validation.isNumberInRange(password2)){
            System.out.println("The password2 is invalid. " +
                "It should be between 10 and 10,000.");
            return ;
        }
        if (!password2Validation.isExactDivision(password2)){
            System.out.println("The password2 is invalid. " +
                "It is not compatible with the denominations.");
            return ;
        }


        // Every case satisfied successfully.
        System.out.println("The username and passwords are valid. " +
        "The door is opening, please wait..");

    }
```

2 UsernameValidation Class

The UsernameValidation Class is an inner class in SystemSecurity class to check that whether username is valid or not. There two conditions for that. The first condition is lenght of username must bigger than or equal to one, and second is the username consists of alpha characters. The method isOnlyAlphaChar checks username's characters' types. And the method checkIfValidUsername checks the length, and invokes the method isOnlyAlphaChar to check character type.

```
/** Inner class for username validation system. */
private static class UsernameValidation{
    /**
     * Checks that whether username is valid or not.
     * @param username The username to check.
     * @return true if username is valid, else false.
     */
    private boolean checkIfValidUsername(String username){
        if (username.length() < 1){
            System.out.println("The username is invalid. " +
                "It should have at least 1 character.");
            return false;
        }
        if (!isOnlyAlphaChar(username)){
            System.out.println("The username is invalid. " + "
                It should have letters only.");
            return false;
        }
        return true;
    }

    /**
     * Checks that whether username consists of alphacharacters or not.
     * @param username The username.
     * @return true if it is only contains alphacharacters, else false.
     */
    private static boolean isOnlyAlphaChar(String username){
        if (username.length() == 0)
            return true;
        else if (!Character.isAlphabetic(username.charAt(username.length() - 1)))
            return false;
        else
            return isOnlyAlphaChar(username.substring(0, username.length() - 1));
    }
}
```

3 Password1Validation Class

The Password1Validation Class is the inner class of SystemSecurity class to check that whether password1 is valid or not. There are four condition to do that. First password1 must contain at least one character that username has. The method containsUserNameSpirit checks that whether this condition is satisfied or not. The second condition is that brackets in string password1 must pair each other. The method isBalancedPassword checks that. The third condition is password1 can be a palindrome when replacing the characters. The method isPalindromePossible checks that. The fourth condition is there must at least two brackects in password. The fourth condition is not checked directly but the count of brackets are count by the method called checkBracketsCount. All of these four methods check conditions apart from each other and return the result independently. Each method is called in outer class method checkUserValid as shown above.

3.1 containsUserNameSpirit

The method containsUserNameSpirit uses two stacks. One for storing characters in password and the other one for storing characters in username. The fillStack method is a helper method to fill stacks with characters of given string.

```
/**
 * Fills stack with characters of given string.
 * @param string The string.
 * @param stack The stack to fill.
 */
private void fillStack(String string, Stack<Character> stack){
    if (string.length() != 0){
        stack.push(string.charAt(string.length() - 1));
        fillStack(string.substring(0, string.length() - 1), stack);
    }
}
```

After stacks are filled, in a loop the characters inside stack which contains characters of the password is pop one by one for each iteration to check that whether current peek of the stack which contains characters of username and popped character are same or not. This steps executed in an outer while loop that pops the stack contains characters of the username to make sure each character in username is checked. If same characters are catched, the method returns true. If loop halts, resets the stacks and return false.

```
/**
 * Checks whether password1 contains any letter that given username contains.
 * @param username The username.
 * @param password1 The password.
 * @return true if it satisfies the condition, else false.
```

```
 */
private boolean containsUserNameSpirit(String username, String password1){
    if (usernameChars.isEmpty())
        fillStack(username, usernameChars);
    if (password1Chars.isEmpty())
        fillStack(password1, password1Chars);

    char c;
    while (!usernameChars.isEmpty()){
        c = usernameChars.peek();
        while(!password1Chars.isEmpty()){
            if (c == password1Chars.pop()){
                usernameChars.clear();
                fillStack(username, usernameChars);
                password1Chars.clear();
                fillStack(password1, password1Chars);
                return true;
            }
        }
        fillStack(password1, password1Chars);
        usernameChars.pop();
    }
    // reset stack.
    fillStack(username, usernameChars);
    return false;
}
```

3.2 isBalancedPassword

Initialises the password stack and calles helper method to check that whether brackets are matching or not.

```
/**
 * Checks whether brackets in string password1 pairs each other or not.
 * @param password1 The password string
 * @return true if open and closing brackets are matching, else false.
 */
private boolean isBalancedPassword(String password1){
    if (password1Chars.isEmpty())
        fillStack(password1, password1Chars);
    if (!checkBracketsMatching(password1)){
        return false;
    }
    return true;
}
```

The method checkBracketsMatching iterates over the characters of given string, stores each opening brackets in a stack and when he find a closing brackets looks for that peek character of the stack and closing brackets are pair or not. If they are pair then pop the top bracket from stack and look for the next one. For else case, because the top bracket at the opening brackets stack is not closed, the method returns false, which mean it is not balanced. If the each character is iterated and there isn't any problem oocured, the method return true, which means the string is balanced.

```
/**
 * Checks that whether brackets pairs in given string.
 * @param string The string to check.
 * @return true if brackets are pairing each other, else false.
 */
private boolean checkBracketsMatching(String string){
    if (string.isEmpty()){
        return false;
    }

    Stack<Character> brackets = new Stack<>();
    char c;
    for (int i = 0; i < string.length(); i++){
        c = string.charAt(i);

        if (c == '(' || c == '[' || c == '{'){
            brackets.push(c);
        } else if (c == ')' || c == ']' || c == '}'){
            if (brackets.isEmpty())
                return false;
            if ((brackets.peek() == '(' && c != ')') ||
                    (brackets.peek() == '[' && c != ']') ||
                    (brackets.peek() == '{' && c != '}') )
                return false;
            else
                brackets.pop();
        }
    }
    return true;
}
```

3.3 isPalindromePossible

The method isPalindromePossible is uses a data field called nonRepeatedString which stores set of the characters inside password. Because it stores a set it has no repeated characters. The method first gets the nonRepeatedString by helper method getNonRepeatedString. And then uses another helper method called getCountOfOddChars which counts number of characters which counts of them in string password1 is odd number. It returns true if count is less than or equal to one, else false. The idea is simple. You can't create a palindrom from a string by replacing its characters if there are more than one odd number of characters. For the simplicity of implementation, the method getNonRepeatedString ignores the bracktes in password and doesn't add them to nonRepeatedString. Each function used in the method isPalindromePossible is recursive functions respect to the homework requirements.

```
/**
 * Checks whether a palindrome can be obtained by replacing characters in
 * password string or not.
 * @param password1 The password string.
 * @return true if a palindrome is obtainable, else false.
 */
private boolean isPalindromePossible(String password1){
    nonRepeatedString = new String();
    getNonRepeatedString(password1);
    int oddCount = getCountOfOddChars(password1, nonRepeatedString);
    return oddCount <= 1;
}


/**
 * Returns the number of characters which in charSet and the count of it is odd in string s.
 * @param s The target string we count on.
 * @param charSet The character set we count.
 * @return The number of odd characters.
 */
private int getCountOfOddChars(String s, String charSet){
    if (charSet.isEmpty())
        return 0;
    int len = charSet.length();
    char c = charSet.charAt(len - 1);
    if (countChar(s, c) % 2 != 0)
        return 1 + getCountOfOddChars(s, charSet.substring(0, len - 1));
    return getCountOfOddChars(s, charSet.substring(0, len - 1));
}


/**
```

```
 * Initialises the string nonRepeatedString with alpha characters in string s
 * without repetition.
 * @param s The target string.
 */
private void getNonRepeatedString(String s){
    if (s.isEmpty())
        return;
    int len = s.length();
    char c = s.charAt(len - 1);
    if (c != '(' && c != '[' && c != '{' &&
            c != ')' && c != ']' && c != '}' &&
            nonRepeatedString.indexOf(c) == -1)
        nonRepeatedString += c;
    getNonRepeatedString(s.substring(0, len - 1));
}
```

3.4 checkBracketsCount

The method checkBracketsCount doesn't check whether count of brackets are satisfactory or not but returns the count of brackets in password. It's a very simple method that its base case is when the string is empty return 0, means starts counting from 0. The case that last character of string is a bracket returns addition of 1 and itself that parameter is substring of password that in range from 0 to *length of password1 - 1* which means that because we find a bracket increase the result by one and call itself to check second right most character in string. The last case is that last character is not bracket but string hasn't over yet. Simple do the same thing we do for the second case without increasing by one.

```
/**
 * Counts brackets in string password1
 * @param password1 The string
 * @return The count of brackets in string password1
 */
private int countBracketsCount(String password1){
    if (password1.isEmpty())
        return 0;
    int len = password1.length();
    char c = password1.charAt(len - 1);
    if (c == '(' || c == '[' ||c == '{' ||
            c == ')' || c == ']' || c == '}')
        return 1 + countBracketsCount(password1.substring(0, len - 1));
    else
        return countBracketsCount(password1.substring(0, len - 1));
}
```

4 Password2Validation Class

Password2Validation is the inner class of SystemSecurity class to check that whether password2 valid or not. There are two conditions for password2. The first one is that password must be in range [10, 10000] which is checked by method isNumberInRange. The other one is that password2 must satisfies the formula that

$$password2 = a \cdot < d1 > + b \cdot < d2 > + c \cdot < d3 > \tag{1}$$

where a, b, and c are coefficients and <dx> are the denomination numbers.

4.1 isNumberInRange

The method that checks whether the password2 is in range or not. If it is in range return true, else false.

```
/** The minimum number password2 can be */
private static final int MIN = 10;
/** The maximum number password2 can be */
private static final int MAX = 10000;

/**
 * Checks whether the number n is in range or not.
 * @param n The number which is checked.
 * @return true if number n is in range, else false.
 */
private boolean isNumberInRange(final int n){
    return MIN <= n && n <= MAX;
}
```

4.2 isExactDivision

Hence the user is able to set denomination numbers, there are two isExactDivision method, that one is overloading another by taking denomination numbers as parameter.

```
/**
 * Checks whether password2 is exact division of denomination numbers or not.
 * @param password2 The number.
 * @return true if it is exact division, else false.
 */
private boolean isExactDivision(int password2){
    int [] denominations = new int[3];
    denominations[0] = 4;
    denominations[1] = 17;
    denominations[2] = 29;

    return isExactDivision(password2, denominations);
}

/**
 * Checks whether password2 is exact division of denomination numbers or not.
 * @param password2 The number.
 * @param denominations The denomination numbers.
 * @return true if it is exact division, else false.
 */
private boolean isExactDivision(int password2, int [] denominations){
    return multA(0, password2, denominations);
}
```

The way of isExactDivision method works is similar to writing 3 nested for loops but in a recursive manner. Like creating a nested loop we have three variable like i, j, and k. In the implementation i call them a, b, and c, like in formula (1). The isExactDivision method calls method multA with giving first parameter as 0. That start a from 0.

4.2.1 multA

The base condition of multA, which is halting case for the recursion, is that $a \cdot denomination[0]$ must be less than password2. There should be a better upper bound but there won't be any change in terms of time complexity therefore I use this upper bound as base case. It basicly says that if the upper bound is reached and method couldn't find coefficients fit the equation return false. Else start second coefficient from 0 by calling method multB.

```
/**
 * First recursive function that iterates second coefficient
 * @param a The first coefficient.
 * @param password2 The password
 * @param denominations The denomination numbers.
 * @return true if it is exact division, else false.
 */
private boolean multA(int a, int password2, int [] denominations){
    if (a*denominations[0] > password2){
        return false;
    } else {
        return multB(a, 0, password2, denominations);
    }
}
```

4.2.2 multB

The base condition of the method multB is same with multA. But the else case is not calls the next coefficient like multA did but increase itself, the second coefficient.

```
/**
 * Second recursive function that iterates second coefficient
 * @param a The first coefficient.
 * @param b The second coefficient.
 * @param password2 The password.
 * @param denominations The denomination numbers.
 * @return true if it is exact division, else false.
 */
private boolean multB(int a, int b, int password2, int [] denominations){
    if (b*denominations[1] > password2){
        return false;
    } else {
        return multC(a, b+1, 0, password2, denominations);
    }
}
```

4.2.3 multC

The base case is c is reached to its upper bound. When the base case happens call multA with increasing a by 1 to call the method multA which is first called in method isExactDivision. The second case is the case that we found the coefficients that satisfies the equation. There is nothing to do left. Just return true. The else case increase c by 1 and continue calling method multC.

```
/**
 * Third recursive function that iterates first and third coefficient
 * @param a The first coefficient.
 * @param b The second coefficient.
 * @param c The third coefficient.
 * @param password2 The password.
 * @param denominations The denomination numbers.
 * @return true if it is exact division, else false.
 */
private boolean multC(int a, int b, int c, int password2, int [] denominations){
    if (c*denominations[2] > password2 ){
        if (a * denominations[0] <= password2) {
            return multA(a+1, password2, denominations);
        }
        return false;
    } else if (a*denominations[0] + b*denominations[1] + c*denominations[2] == password2){

        return true;
    } else {
        return multC(a, b, c+1, password2, denominations);
    }
}
```