Programming in Java

Assignment 1 (Formative)

University Information System Tools

Marks available: 100*

Date issued: 04/10/19

Deadline: 14/10/19, 17:00

*This is a formative assignment for all students. That means you must submit your work but it will not contribute to your final module mark. You will get feedback on your work.

Introduction

If you are new to programming, and to Java, make sure you have fully understood the 'Getting Started' document before attempting these exercises.

If you do not understand how to do the first exercises in this assignment (and you have worked through the 'Getting Started' document) then do the practice exercises on Canvas, and look at the solutions for those.

Submission instructions

Submit your work by the deadline above as a zip archive. Submit only .java source files in your archive. Do NOT submit .class files or any type of file. Please do not submit entire Eclipse (or other IDE) projects.

In your programs, do not output anything to the console unless explicitly asked to.

Where applicable, your class for each of the exercises must have the name shown in the exercise heading.

Each Java file must have a package declaration at the topic of it (these declarations must be on the very first line of the file). The final part of this declaration varies according to the exercise. Below is shown the declaration for the first exercise.

```
package com.bham.pij.assignments.gradechecker;
```

The declaration for the fourth exercise will be:

```
package com.bham.pij.assignments.shortidcreator;
```

Do not forget to add the semicolon at the end of this line. All characters in a package name should be lower case. You can add the package declaration as the last step before you submit your work but make sure you enter it correctly, make sure you put the semi colon at the end, and make sure your file still compiles when you have added the package declaration.

In each of these exercises, you are not restricted to only creating the *required* methods. If you want to, you can create others. However, you must create **at least** the required methods.

Do not make the methods required by these exercises `static`.

Do not use any of the Java collections for these exercises, e.g. `ArrayList`.

You must not use *regular expressions* in your solution. Use of regular expressions in your solution will result in a mark of zero. You may use the `String split()` method if you wish but do not pass it anything other than a single character as a delimiter.

IMPORTANT: Do NOT put any code that gets user input in the required methods below. If you do, your work can't be tested. This means that you should not use `Scanner` in any required method or in any method called by a required method, etc. Please ask if unsure about this.

Exercise 1: `GradeChecker` [10 marks]

You may use the `GradeChecker.java` file on Canvas for this exercise. If you do that, you simply need to complete the two required methods. You do not need to use that file. You can write your own if you prefer.

Write a program that asks the user to enter a student grade and then outputs either "Pass." for a pass grade or "Fail." for a fail grade. The pass mark is 40 out of 100. A grade of less than 40 is a fail and a grade of 40 or more is a pass. If the input grade is not valid (according to the definition below under the method `isValid()`), the program should output the message "Invalid input" and end.

Your program should have a class called `GradeChecker` with a `main` method.

Your `GradeChecker` class must have the following methods:

```
public boolean isValid(int grade)
```

This method returns true if the input grade is in the range [0,100] (0 to 100 inclusive), and false if not. This method must do nothing else apart from this. For example, this method must not get the user input or do any other processing.

```
public boolean isPass(int grade)
```

This method returns `true` if the input grade is 40 or above, and `false` if not. This method must do nothing else apart from this. This method does **not** validate the input grade.

Write a class called `GradeChecker` and inside that class put a `main` method. Your class must also have the two other methods listed above.

Exercise 2: `MeanGrade` [10 marks]

Do **not** use any of the Java collections for this exercise, e.g. `ArrayList`.

Create a program that will compute the mean grade of four input grades. Each grade represents the mark for one module, thus four module marks are to be entered. The user should be prompted to enter four seperate grades. The user should press the enter key after each grade.

The program should then compute and output the mean of the grades entered. The program should not allow more than four grades to be entered.

If an input grade is not valid, the program should quit with an error message.

When designing your program, consider how you would extend it to handle a larger number of grades, if requested. For example, how would your program handle 20 grades? 100? Your program only needs to handle **four** grades but do think about this design question.

Your program should have a class called `MeanGrade` with a `main` method.

Your `MeanGrade` class must have the following method:

```
public double computeMean (int[] grades)
```

This method computes the mean of the grades in the array `grades`. Note that it returns a `double`. This method should **not** assume that the array has `length` four[1]. If the input array contains an invalid value, this method should return the value -1.0.

Remember that, in Java, you can find the length of an array as follows. For the array `grades`, the expression `grades.length` evaluates to the size (length) of the array. This value will always be the size that the array was created with and **not** a count of how many values have been set in the array.

Exercise 3: `ResultChecker` [10 marks]

Create a program that outputs a result for a student based upon the mean grade of **eight** module grades and a project mark.

The rules for the marks are the same as for the above exercises. The pass mark is 40 and a valid grade is in the range [0,100]. These rules apply to individual module grades and to the project.

The result is computed as follows:

If any individual module grade or the project grade is not valid then the result is “ERROR”.

If any individual module grade or the project grade is less than 40, the result is “FAIL”.

If the result is not “ERROR” or “FAIL”, and if the mean grade or the project grade is less than 50, the result is “PASS”.

[1]The reason for this is that you might wish to use this method in a different application in which the number of values is not restricted to four.

If the result is not “ERROR” or “FAIL”, and if the mean grade and the project grade are 50 or more, the result is “MERIT”.

Your program should have a class called `ResultChecker` with a `main` method.

Your `ResultChecker` class must have the following method:

```
public String getResult(int[] grades, int projectGrade)
```

This method returns one of the `String` values above based upon the input grades and the project grade. Thus, the return value of this method will be one of {“ERROR”, “FAIL”, “PASS”, “MERIT”}.

Exercise 4: `ShortIDCreator` [10 marks]

Create a program that receives a student’s name as input and then outputs a three-character ID for that student. This three-character ID would be used to create a longer, unique ID but that is not a part of this particular program.

Prompt the user to enter a student’s name. You can assume that the student’s name will have either two or three parts to it, i.e. it will either be *first-name last-name* or *first-name middle-name last-name*[2].

The ID is created as follows. If the user enters a three part name, the ID number is created by taking their initials, i.e. the first character of each part of the name, converted to lower-case characters where necessary. For example, the name “David Robert Jones” would lead to an ID of *drj*.

If the user enters a two-part name, the ID is again created by taking their initials, converted to lower-case characters where necessary, but this time substituting an “x” character for the missing middle name. For example, the name “Richard Starkey” would lead to an ID of *rxs*.

All upper-case letters must be converted to lower-case in the ID.

Your class must contain the following method:

```
public String createID(String input)
```

This method receives the input name as a single `String` and returns the generated ID as a `String`.

If the input string is not valid then this method should return the value `null`. The input is not valid if:

- It is `null`.
- It is empty.
- It does not contain either two or three parts.

Exercise 5: `PasswordChecker` [15 marks]

Do not use *regular expressions* in your solution for this exercise.

[2]I appreciate that this is inadequate for some names but the restriction is needed to minimise the complexity of the task.

Write a program that checks to see if an input password is valid or not. In this exercise, a valid password has the following characteristics:

- It contains at least eight characters.
- It contains at most 12 characters.
- It contains only alphabetic characters, digits and the underscore ('_') character.
- It does not start with a digit.
- It has a mix of upper-case and lower-case characters.

Your class must contain the following method:

```
public String checkPassword(String input)
```

This method receives a possible password as input and returns only the `String` "OK" if the password is valid. If the password is not valid then you should return the reason(s) for that as the return value. The following strings should be returned for the given errors:

If the password is too short, return "TOO SHORT".
If the password is too long, return "TOO LONG".
If the password contains illegal characters, return "WRONG CHARACTERS".
If the password starts with a digit, return "LEADING DIGIT".
If the password does not have a mix of upper and lower-case characters, return "NOT MIXED CASE".

Do not change these return strings in any way. Return exactly those strings.

Exercise 6: `ShortAddressCreator` [15 marks]

Do not use *regular expressions* in your solution for this exercise.

Write a program that can create a 'short address' from a longer one. The program should be able to receive an address that has a fixed format but is of arbitrary length, and convert that to a specified shorter format.

All input addresses will have the following format:

address-line-1, address-line-2, ..., address-line-n, postcode

Note that there is a space character after each comma.

All postcodes will have the following format:

adddaa.

Where *a* is an alphabetic character and *d* is a digit[3].

[3]This is not a realistic format since it excludes many real postcodes

For example, the address *56 Clapham Gardens, Stoke Newington, London, N145PX* has the given format. As does *3 Acacia Ave, E178PU*.

The short address that your program generates must have the following format:

address-line-1 postcode

Note that there is no comma in this string but there is a space between *address-line-1* and *postcode*.

For the examples above, the short addresses would be:

56 Clapham Gardens N145PX and *3 Acacia Ave E178PU*.

Your class must contain the following method:

```
public String createShortAddress(String input)
```

This method receives a long address as input and returns the generated short address. If the input is `null` or if it is empty, this method must return `null`.

If the input is incorrectly formatted, i.e. it does not have the exact format shown above, the method should return `null`.

Exercise 7: `FullIDCreator` [15 marks]

Do **not** use any of the Java collections for this exercise, e.g. `ArrayList`.

A full student ID includes the short ID that can be created by the program above, appended with a four-digit integer number. Full student IDs are unique.

The rules for the creation of the short ID part of the ID are the same as above. The four-digit integer part is used to make the full ID unique. It works as follows. If a particular short ID has never been created, the short ID is appended with "0000". If a particular short ID has been created before, it is appended with a four-digit integer that makes the overall full ID unique. Consider the following examples.

Imagine the very first student short ID created is "drj". Since this is the first ID that has been created, it can be appended with "0000" to make a unique full ID. Thus the full id is "drj0000".

Consider that the next short ID created is "bxb". This short ID has also never been created before, so it can be appended with "0000" to make "bxb0000".

Now imagine that the next short ID created is ="drj" (because people can have the same initials). "drj" has already been used so it cannot be appended with "0000" since that will create a duplicate ID. The sensible approach would be simply to append it with "0001". Now the full ID is "drj0001" and is unique.

The next time "drj" is encountered its integer part would be "0002", hence the full ID would be "drj0002", and so on.

Thus, the integer parts of the full ID should be incremented every time a duplicate short ID is encountered.

The maximum integer ID part is “9999”. You can assume that no more than 10000 student IDs will ever be created.

Note that full IDs are still `String`s even though they contain digits.

Write a program that can create these full IDs. Your program should have the following method:

```
public String createFullID(String input)
```

This method receives a student name, in the two-part or three-part format from Exercise 4, and returns the full ID, which is created as described above. If the input name is not well formed (i.e. it does not follow the two-part or three-part format descibed above), this method should return `null`.

Exercise 8: `LegacyCleaner` [15 marks]

Do not use *regular expressions* in your solution for this exercise.

The (ficticious) university has been using a student records system since the 1990s. It is now badly out of date and error prone. It has been possible to export some data from this old system in the form of strings. These strings contain some student details.

Your task is to take these exported strings and put them into a standard format containing the following student details: name, id, result, postcode. All of these details use the same format as described in the previous exercises.

Not all of the records exported from the old system are complete. Not all of them have the data in the same order. Some of them have corrupted data.

You are assured that, whatever the state of the data in an individual string, the elements of the string **that exist** are separated by commas. Here are some examples of the kind of strings you will be given to work on:

“Elvis Presley,,B712XY”

“David Robert Jones, drj1435, PASS, N165PT”

“Archibald Leach, axt2889, FAIL,”

“Michael Jagger, mxj0991, MERIT”

“pct6761, FAIL, SW123A”

“,,,,”

“,,,M149BX”

“FAIL, Basil Brush, N112JP”

“Karen8 Smith, uiXm_71289df_, kxs3865”

As you can see, in the input strings some elements are empty. There are also possibly corrupted elements (as in the last example above). If an element is empty you are not expected to try and generate an element for that. If an element is corrupted, i.e. it does not follow any of the assumptions outlined below, you are not expected to try and correct it. You do not have to validate the contents of the input further than is described below. For example, if the name field is "John Winston Lennon" and the id field is "zbd3455" you can return both of those in your result. You do not have to validate the ID against the name.

Your program should take each input string and put it into the following order:

{name, id, result, postcode}

You can make the following assumptions:

- Nobody's name is FAIL, PASS or MERIT.
- All names consist only of alphabetic characters and space characters.
- IDs are always 7 characters long and consist only of alphabetic characters and digits.
- Postcodes are always 6 characters long and consist only of alphabetic characters and digits.

Your program should try and find an element in the input string that matches the name, id, result and postcode, using the assumptions above. As stated earlier, you are **not** expected to *correct* any entry.

The limited range of assumptions means that, potentially, your program could identify something as an ID that is not actually an ID (for example). A corrupted element may have the form of an ID but not actually be one (in the system). You do not have to worry about this. You can't do anything about that anyway, based on the information you have been given.

Your program must have a method with the following signature to clean the strings (one by one):

```
public String[] clean(String input)
```

This method returns an array with elements in the order {name, id, result, postcode}. The elements will have been discovered in the input string. If any particular element is not found, its value in the array should be set to `null`. Any unnecessary spaces in the input string (i.e. those that are not in the name element) should be removed.

If the input string is null or empty, or the input string contains no commas, or if the input string contains only commas, this method should return `null`.