EE 422C Project 2
Object Oriented Mastermind
Individual Assignment

See Canvas for due dates.

General Assignment Requirements

The purpose of this assignment is to design and implement an OO program with multiple classes to play a text based version of the board game called Mastermind. You are free to use whatever classes and methods from the JAVA 8 library you wish.

The first thing to do is - Read the Wikipedia article on the game of Mastermind at:
http://en.wikipedia.org/wiki/Mastermind_(board_game)
Here is a good example of what an interactive GUI for Mastermind would look like:
http://www.web-games-online.com/mastermind/index.php

The version of the game you implement will have the following properties.

- The computer will randomly generate the secret code.
- The player will try to guess the secret code.
- The player has 12 guesses to guess the code correctly. This number should be easily changeable by modifying your code.
- If the player does not guess the code correctly in 12 or fewer guesses they lose the game.
- The secret code consists of 4 colored pegs in specific position order.
- The valid colors for the pegs are blue, green, orange, purple, red, and yellow. Capital letters will be used in the examples to indicate colors. B for blue, R for red, and so forth. The number of colors should be changeable for full credit. The selection of the number of colors, like the number of guesses and the number of pegs, should be changeable through the provided class `GameConfiguration.java`, by changing the code and re-building.
- The results of a guess are displayed with black and white pegs. The Wikipedia article refers to the results as feedback.
- A peg in the guess will generate feedback of either: 1 black peg, 1 white peg, or no pegs. A single peg in the guess cannot generate more than 1 feedback peg.
- The order of the feedback does not give any information about which pegs in the guess generated which feedback pegs. In your feedback, you must give the number of black pegs (if any) first, and then the number of white ones, if any.
- The player's guesses must be error checked to ensure that they are of the correct length and only contain valid characters.

The output of the game should be a simple text based display on the console, similar to the following example:

Input / Output
Initially, before getting any input and before running any game print the greeting message. For each game print a single line asking the user if he wants to play a new game. If and only if you are in testing mode, print the secret code.

Then enter the part of the program asking for user guess inputs. Before asking for user input, print a blank line, then print the number of remaining guesses. Process the user's guess and print the number of black/white pegs. If the guess is not valid, print INVALID_GUESS. There is a special case of guess -- history -- in that case, print all previous valid guesses and their resulting pegs. If the game ends (losing or winning) print a blank line, then ask again for whether the user wants to play a new game. If user does not want to continue, print nothing and exit the program.
Input and outputs are **case-sensitive**. If you see any space in the sample dialogue below, it means a **singe space character,** and not any other form of white space (e.g. tabs, multiple spaces).
Do not use end of line character explicitly (no \n or \r\n etc.) as its interpretation depends on the underlying platform. Always use println; in any case if you want to explicitly use the end of line character, try System.getProperty("line.separator") , or simply use println with no input
Each line of output should have exactly one line separator. It means that no other blank line is allowed (except what we said about each guess and each result). Also, the last line of output should have an ending line separator as well. **We will try to ignore newline differences and ignore blank lines, but do the best you can to follow the directions.**
You can assume that except for invalid guesses, we (the player) will not give you erroneous inputs. For example, when you are expecting the user's response for a new game, we will always give Y or N. Invalid guesses also may only have error in the content; they won't have whitespaces or special characters.
Please carefully follow the sample user dialogue. Any failure to obey the rules, may cause loss of points.

Sample user dialogue
Console output is in `typewriter font`, and user input is **bold-underlined**. Do not change the text or format. If you pipe your output to a file instead of the console, the user input part will be missing in the file. In some lines we use // for adding comment.; the comments (including // part) should not be taken as sample output/input.

```
Welcome to Mastermind.//Initial greeting
Do you want to play a new game? (Y/N):
```
Y
```
Secret code: YRBY//Print this line iff we are in testing mode
//the blank line we talked about
You have 12 guess(es) left.
Enter guess:
```
OOOO
```
OOOO -> 0b_0w//0 black peg, 0 white peg

You have 11 guess(es) left.
Enter guess:
```
oooo
```
INVALID_GUESS//case sensitive

You have 11 guess(es) left.//Invalid guesses are not counted
Enter guess:
```
RRRRRRR
```
INVALID_GUESS//length
```

```
You have 11 guess(es) left.
Enter guess:
```
RRRR
```
RRRR -> 1b_0w//1 black peg, 0 white peg
```
(Etc. etc.,. . .)

```
You have 1 guess(es) left.
Enter guess:
```
HISTORY
```
OOOO -> 0b_0w
RRRR -> 1b_0w
RRRR -> 1b_0w
RRRR -> 1b_0w
RRRR -> 1b_0w
RRRR -> 1b_0w
RRRR -> 1b_0w
RRRR -> 1b_0w
RRRR -> 1b_0w
RRRR -> 1b_0w
RRRR -> 1b_0w//imagine that we did not make any new guess

You have 1 guess(es) left.
Enter guess:
```
RRRR
```
RRRR -> 1b_0w
You lose! The pattern was YRBY//of course, this line depends on the
secret pattern

Do you want to play a new game? (Y/N):
```
Y
```
Secret code: YRBY//Let's assume that it is a new code

You have 12 guess(es) left.
Enter guess:
```
YRBY
```
YRBY -> 4b_0w
You win!

Do you want to play a new game? (Y/N)://final line of output
```
N

Program structure

You must have a class in your program called `Game`**, which is nearly at the top-level** (a `Driver` class with `main()` should call `Game's` constructor, and is at the top level). It must have a constructor that takes a `boolean` value as its parameter. The `boolean` value is used for running the game in the testing mode until you are finished. If it is true, then the secret code is revealed for every game you play, as shown in the sample output. You should read in the first

argument to `main`, and if and only if the first argument exists and it is `1`, you should set testing mode to true; otherwise testing mode is false. Look up documentation on how to use the `String[] args` part of `main`. For example, we will call your program by saying (something like)
`java assignment2.Driver 1`
if we want to set the testing mode to `true`, and
`java assignment2.Driver` or
`java assignment2.Driver <something else>`
if we want to set the testing mode to `false`.

Your program must have a `Scanner` object connected to the keyboard that is passed to any methods as necessary. You must have only 1 `Scanner` object connected to the keyboard, and it should be created only once during your entire program. (Other Scanner objects (not connected to the keyboard) may be used, if necessary.) It can be created once in `main()`, and passed to `Game` as an additional constructor parameter (in addition to the test mode), or created once in the `Game` class's constructor, if the Game object is created only once in your program. For example, if the user finishes a game and wishes to play again, a new `Scanner` object should not be created. Creation of multiple `Scanner` objects breaks our grading script's functioning.

The `Game` class must also have a method named `runGame` that carries out the actual games. Your `main` method could create and uses a new `Game` object for each game played, or, what might be better, create one `Game` object that has a loop for multiple games. Do not use `System.exit()` to exit your program, as it will break our grading scripts.

To create the random code, use the provided code, rather than make your own. Import the `SecretCodeGenerator` class, and use it thus:
`SecretCodeGenerator.getInstance().getNewSecretCode()` to return a `String` code. Note that `SecretCodeGenerator` has a private constructor. As we will replace `SecretCodeGenerator` for each test case, do not change the signature of any parts of this class; also, get new secret code exactly once for each game. However, you may modify its body code to test your code. **Do not submit this class with your solution.**

How to proceed
Recall that when designing a program in an object oriented way, you should consider implementing a class for each type of entity you see in the problem. For example, in Mastermind there is a game board, pegs, colors, codes (the secret code and the player's guesses can both be considered the same data type, a code), feedback results, a computer player, a human player, and an over-all game runner. Some things are so simple you may choose not to implement a class for them. For example, the computer player doesn't do anything more than pick the secret code and answer the guesses. Maybe that is simple enough for the Game or Board class to do. Also you may use some pre-existing type, primitive or a class, to represent things. E.g. the guesses and results could be `String` objects.

After deciding what classes you need, implement them one at a time, simplest ones first. Test a class thoroughly before going on to the next class. You may need to alter your design as you implement and test classes. Remember Design a little - code a little - test a little. Repeat.

One of the criteria of the assignment is to break the problem up into smaller classes even if you think the problem could be solved more easily with ONE BIG CLASS. For this assignment you should have more than 1 class.

I recommend you work on this incrementally. Start with a design and try to get that to work. Have a working program at all times and add to it as you implement more features. This will avoid the assignment becoming an all or nothing affair. Even if you don't finish you will have a working version with some functionality ready to turn in.

Design deliverables for Part 1 of the assignment

The design to be turned in on paper during recitation will consist of the following:

- - For use case: Write down a paragraph describing which users will use this program, and which actions they may have.
- - For Class diagram: Write down a paragraph describing which classes you will create to finish this implementation, with a UML class diagram.
- - For Sequence diagram/flowchart: Give us 2 UML sequence diagrams (or flowcharts) that express the high level algorithm descriptions for a) the game runner, and b) formulating the reply to a given guess.

If you have made significant changes to the methodology described in these, resubmit these with your final code.

Warning *- You may not acquire, from any source (e.g., another student or an internet site), a partial or complete solution to this problem. You may not show another student your solution to this assignment. You may not have another person (TA, current student, former student, tutor, friend, anyone) "walk you through" how to solve this assignment. Review the class policy on cheating from the syllabus.*

Tip *– we may come back to this assignment later in the semester to put a fancy GUI on it, so design it with that in mind. I.e. encapsulate the user interface so it can be replaced.*

Package statement
Make sure that all your java files are in a package called assignment2.

Final check
We will be supplying you some grading scripts and sample testcases for your final check before submission.

Submission
For Part 1: Turn in your work during recitation. Your TA will inspect your work. You must attend recitation to get credit for this part. Make a directory named assignment2, put all your PDF files in it, zip it up, rename the zip file **Project2_EID.zip,** and upload it to Project 2, first part, on Canvas, if we ask you to do so. If we don't ask for an electronic version, just turn in your work on paper.

For Part 2: When you are done, you will have a directory named "`assignment2`" that contains your java files. Do not include your test files, and do not include `GameConfiguration.java` or `SecretCodeGenerator.java`. All java files must be in exactly 1 level under assignment2; nested directories are not accepted. Put any other file (such as updates to part 1) in that same directory. Then compress it to generate a **zip** file. We recommend that you make the zip file from the command line. Please note that compressing files in OSX may generate an unwanted directory called "__MACOSX__" or something similar. Please make sure that by extracting your zip file, no such directory appears (it is better to check the unzip on the command line). Finally, change the name of your zip file to **Project2_EID.zip.**
Disobeying the submission rules above may break the automated grading which **will cause you to receive a 0 grade or large penalty**.

CHECKLIST – Did you remember to:

- ☐ Re-read the requirements after you finished your program to ensure that you meet all of them?
- ☐ Make sure that you use the keyboard Scanner in the prescribed way?
- ☐ Make sure that you use the `SecretCodeGenerator` in the prescribed way?
- ☐ Make up your own testcases?
- ☐ Make sure that all your submitted files have the appropriate header file?

- ☐ Put all your source files in a package and folder named assignment2?
- ☐ Upload your solution to Canvas, remembering to include ALL your files in one zip file named Project2_EID.zip?
- ☐ Tried out your code on our grading scripts on kamek?
- ☐ Download your uploaded solution into a fresh directory and re-run all testcases?

Adapted from an assignment written by Mike Scott and Herb Kreisner.