DUE DATE: OCTOBER 23, 2019 AT 11:59PM

Instructions:

- You must complete the "**Blanket Honesty Declaration**" checklist on the course website before you can submit any assignment.
- Only submit the **java files**. Do ***not*** submit any other files, unless otherwise instructed.
- To submit the assignment, upload the specified files to the **Assignment 2** folder on the course website.
- Assignments must follow the **programming standards** document published on UMLearn.
- After the due date and time, assignments may be submitted but will be subject to a late penalty. Please see the ROASS document published on UMLearn for the course policy for late submissions.
- If you make multiple submissions, only the **most recent version** will be marked.
- These assignments are your chance to learn the material for the exams. Code your assignments independently. We use software to compare all submitted assignments to each other, and **pursue academic dishonesty vigorously**.
- Your Java programs must compile and run upon download, without requiring any modifications.

Assignment overview

In this assignment, you will build a version of the game Bejewelled. Bejewelled is a 'match 3' game, where you must match 3 or more of the same gem by moving the gems on the board. The gems can only move 1 cell up or down, left or right, and swap with neighbouring gem when moving. Upon matching, the gems that match disappear, leaving a gap, and the gems above the gap fall, filling the gap. If you are unfamiliar with the game, you can plan an online version here.

Keep all of your methods short and simple. In a properly-written object-oriented program, the work is distributed among many small methods, which call each other. There are few methods in this entire assignment that should require more than 15 lines of code, not counting comments, blank lines, or {} lines.

Unless specified otherwise, all instance variables must be **`private`**, and all methods should be **`public`**.

Your assignment should be DRY – don't repeat yourself. If you notice you have code that does the same function in two places, consider making a private method that can be called to do the work. If you see 3 or more lines of code that do the same thing in two different methods, consider making a **`private`** method that does that task. This is often called a 'helper' method.

Testing Your Coding

We have provided sample test files for each phase to help you see if your code is working according to the assignment specifications. These files are *starting* points for testing your code. Part of developing your skills as a programmer is to think through additional important test cases and to write your own code to test these cases.

Phase 1: The Grid

First, create the game board, named class Grid.

The **`Grid`** class should have:

- An instance variable to hold the board, which is a 2D array of **`char`**.
- A method **`fillBoard()`**, which fills the board with random gems, overwriting whatever is in the board.
- A constructor **`Grid(int height, int width)`**, which creates a new game board filled with random gems.
- A standard **`toString`** method, which returns a text representation of the board.

To choose random gems, you can add this to your Grid class:

```
public static char[] gems = {'r', 'g', 'b','y'};
```

```
public static char getRandomGem() {
	int choice = (int)(Math.random() * gems.length);
	return gems[choice];
}
```

Phase 2: Grids from files

We want to be able to save our progress. To do this, we will save our progress to a file, and be able to read these files.

The save files have a special format. The top line is the dimensions of the saved board: number of rows, then number of columns. The rest of the file is the contents of the board. Example:

```
2 3
yrb
bry
```

Is a valid file.

To do this, implement these methods in the Grid class:

- A new constructor, **`Grid(char[][] someGrid)`**, which creates a grid with the given data. You do not need to check if the gems are valid (in the set of letters provided).
- A factory method called **`Grid createGrid(String filename)`**, which reads the data the text file specified by 'filename', and returns a Grid object with that data. This method throws **`IOException`** if the data is not valid. Special exception messages include:
  - Bad dimensions for the saved file
  - Not enough rows of data to read
  - Not enough columns of data to read

  Your error messages should have as much detail as possible. See the sample output for what is expected. This method is quite long, due to the error handling, and will exceed the usual "15 lines of code" rule-of-thumb.
- A **`save(String filename)`** method, that writes the current board to the specified file, in the save file format used. If there is an exception, it should print out the error message, and return null.

Sample output:

```
yr
by
bb
rr

java.io.IOException: No dimensions to read
java.io.IOException: The size values were not numeric!
java.io.IOException: There was not enough rows! Saw 5 need 10
java.io.IOException: There was not enough colums. Saw 2 need 12
yr
by

These should be the same:
rryryb
yybgbg
yrrbrb
yyryrr
brgbgr


rryryb
yybgbg
yrrbrb
yyryrr
```

```
brgbgr
```

Phase 3: Extracting data

Add the following methods to extract data from your Grid class. You will use these later in the program.

- Method **`char[] extractRow(int rowNum)`** that extracts the row specified by the passed integer. 0 will fetch the first row, 1 the second (index-by-zero rules).
- Method **`char[] extractColumn(int colNum)`**, which extracts a single column, much like extract row.

Phase 4: Checking for gems in a row

Write a method named **`static char[] replaceSets(char[] input)`** in Grid that is passed a 1D array of gems, and checks if there are set of 3 or more gems in a row. Return a new array that has the gems that are in a row replaced with 'x' characters.

Phase 5: Merge

We have methods to find if there are sets of gems. Now, write the method **`replaceAll()`** in Grid. This method uses the methods created in phase 4 to mark any sets of 3 with x's.

There are some subtleties to this method. Consider:

```
bbb
bgg
bgg
```

There is a set of 3 in both the first row, and first column. If you set the first row to all x's, then check the column, then column 0 would be xbb, and the program would not set column 0 to all x's. A solution to this is to create a new 2D array with the sets replaced by x's, while looking at the original to see if there are sets to be replaced.

Phase 6: Drop

Create method **`boolean drop()`** in class Grid. This method 'drops' the gems that are above cells that have an x in them. In Bejewelled, as gems disappear, the gems above them drop down. This method does the dropping step.
The drop method returns **`true`** if any gems have moved.

In any spots that are now empty, place random gems. This will be the gems that are 'falling from the top'.

Phase 7: Swap

Create 4 **`public static final`** variables in class Grid: **`UP`**, **`DOWN`**, **`LEFT`**, and **`RIGHT`**. You can set these to any numbers you like but they all must be different numbers.

Create the method **`void swap(int row, int col, int direction)`**. This will swap the gem specified by row and column (indexed at 0) in the direction specified. Throw appropriate exceptions for out-of-bounds gem selection, and movement. Throw **`IndexOutOfBoundsException`** for any strange things that may happen.

Phase 8: Bonuses: Is that swap allowed? Ending the game?

+10%: In Bejewelled, moves are only allowed if it causes a match. Update the game to enforce that rule.

+20%: The game is over when there are no more moves. Update the game, and TestPhase7b.java to enforce this. Visit every gem, and see if it can move up/down/left/right. Continue the game if there is at least 1 valid move left.

If you accomplish these bonus items, write that you did so in the comment section of the hand-in.

Hand in

Submit your one Java file (**`Grid.java`**). ***Do not submit .class or .java~ files!*** You do ***not*** need to submit the **`TestPhaseN.java`** files that were given to you unless you attempted the bonus. If you did not complete all phases of the assignment, use the Comments field when you hand in the assignment to tell the marker which phases were completed, so that only the appropriate tests can be run. For example, if you say that you completed Phases 1-2, then the marker will compile your files with appropriate test cases. If it fails to compile and run, you will lose ***all*** of the marks for the test runs. The marker will ***not*** try to run anything else, and will ***not*** edit your files in any way. ***Make sure none of your files specify a package at the top!***