DUE DATE: FEBRUARY 16, 2018 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 lose **2% of the marks per hour** late.
- If you submit a question multiple times, 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**.
- To be eligible to earn full marks, your Java programs must compile and run upon download, without requiring any modifications.

Assignment overview

In this assignment, you will implement the "2048 Puzzle" which was quite a hit a few years ago (starting in 2014). You can try playing the game at **`2048game.com`**.

Warning: There is source code available for this game in a variety of languages. **Do not use any of it** since 1) we already know about it, 2) you are automatically guilty of academic dishonesty if you do, 3) you will not get any benefit from doing the assignment, and 4) this assignment is structured in a way that the available code won't work anyway.

Phase 1: Shifting One Line

The 2048 board is a square 2-dimensional grid of "tiles". Each place in the grid might be empty, or contain a tile showing a number that is a power of 2 (2, 4, 8, 16, etc.). In this assignment, a simple **`int[][]`** array will be used, with the value 0 representing an empty place.

The main operation involved in playing the game is to shift a line of tiles in some direction. In Phase 1, only a 1-dimensional line of tiles will be used, and the tiles will only be shifted to the left (from higher index values toward 0).

Define a **`Board2048`** class. For now, it should contain only a **`public static boolean alterOneLine(int[])`** method. This method should accept an **`int[]`** array representing one line of tiles, and alter the array according to the rules and examples given below. It should return **`true`** if the array was changed in any way, and **`false`** if it remained exactly the same as it was before. It should handle any size of array. The rules can be implemented and tested one at a time.

Rule 1: All the tiles should be shifted to the left as far as they will go. (Empty spaces (0's) do not count.) For example:

```
> int[] test = {0,2,0,0,8,0,4};
> Board2048.alterOneLine(test)
true
> test
{ 2, 8, 4, 0, 0, 0, 0 }
> int[] test = {0,0,0,0,8,2,4};
> Board2048.alterOneLine(test)
true
> test
{ 8, 2, 4, 0, 0, 0, 0 }
> int[] test = {2,4,2,0,0,0,0};
> Board2048.alterOneLine(test)
false    (the line did not change)
> test
{ 2, 4, 2, 0, 0, 0, 0 }
```

Rule 2: If two tiles containing the same number are shifted into adjacent positions, they are merged into a single tile with double that value.

```
> int[] test = {2,0,2,4,0,0,4};
> Board2048.alterOneLine(test)
true
> test
{ 4, 8, 0, 0, 0, 0, 0 }     (the two 2's merged, and the two 4's merged)
> int[] test = {2,0,4,2,0,0,4};
> Board2048.alterOneLine(test)
true
> test
{ 2, 4, 2, 4, 0, 0, 0 }     (nothing merged – the 2's are not adjacent, nor are the 4's)
```

Rule 3: If a new tile is created by merging two other tiles (Rule 2) then it is "locked" and may not be changed again in this shift operation. (But it may be changed in future calls to **`alterOneLine`**.) Pairs of tiles must always be merged from left to right.

```
> int[] test = {2,0,2,2,0};
> Board2048.alterOneLine(test)
true
> test
{ 4, 2, 0, 0, 0 }     (The first pair of 2's on the left merge, not the last pair.)
> int[] test = {2,0,2,2,0,2};
> Board2048.alterOneLine(test)
true
> test
{ 4, 4, 0, 0, 0, 0 }     (Two pairs of 2's merge, but the 4's are locked and do not merge.)
> int[] test = {2,0,2,4,0,8};
> Board2048.alterOneLine(test)
true
> test
{ 4, 4, 8, 0, 0, 0 }     (The 2's merge but the new 4 that results is locked.)
> Board2048.alterOneLine(test)
true
> test
{ 8, 8, 0, 0, 0, 0 }     (Another shift of the same array will merge the 4's.)
> Board2048.alterOneLine(test)
true
> test
{ 16, 0, 0, 0, 0, 0 }     (A third shift of the same array will merge the 8's.)
```

You can use the supplied **`TestPhase1.java`** program to run the tests shown above.

Phase 2: Shift The Board

Now modify the **`Board2048`** object so that it will hold a 2-dimensional grid of tiles, which can be shifted in any direction.

1) Add instance variables to the **`Board2048`** class so that a **`Board2048`** object will contain a square grid of **`int`** values of any size. (The number of rows must always be equal to the number of columns.)

2) Add **`public static final int`** constants **`UP`**, **`DOWN`**, **`LEFT`**, and **`RIGHT`** to represent the four possible directions. It doesn't matter what values you use to represent them.

3) Provide a constructor with one **`int`** parameter that specifies the desired size of the grid. (The normal game is played on a 4x4 grid but your program should handle any size.) The grid will always be square. The grid should initially be blank (all 0's).

4) Provide a second constructor with one **`int[][]`** parameter that specifies a particular grid of values to be used. You may assume that the supplied array will be square, but it may be any size. The values in the grid may be any numbers, not only powers of 2.

5) Write a standard **`toString`** method which will return a multi-line **`String`** representation of the grid. Use tab characters (**`\t`**) between the columns and newline (**`\n`**) characters between the rows. Use a '-' for a blank space.

6) Write a method **`public int[] extractLine(int i, boolean vertical, boolean reverse)`** which will return an **`int[]`** array containing one particular row or column of the grid. If **`vertical`** is **`true`** it should return column i. If **`vertical`** is **`false`** it should return row i. If **`reverse`** is **`false`** it should return the values in the usual order (left to right or top to bottom), but if **`reverse`** is **`true`** it should return them in the reverse order. The array that is returned should be a newly-created object. For example:

```
> Board2048 test = new Board2048(new int[][]{{1,2,3},{4,5,6},{7,8,9}});
> test
1     2       3
4     5       6
7     8       9
> test.extractLine(1,false,false) //row 1, normal order
{ 4, 5, 6 }
> test.extractLine(1,false,true) //row 1, reverse order
{ 6, 5, 4 }
> test.extractLine(1,true,false) //column 2, normal order
{ 2, 5, 8 }
> test.extractLine(2,true,false) //column 2, normal order
{ 3, 6, 9 }
> test.extractLine(2,true,true) //column 2, normal order
{ 9, 6, 3 }
```

7) Write a method **`public void insertLine(int[] line, int i, boolean vertical, boolean reverse)`** which will work exactly like **`extractLine`**, but in reverse. The given **`int[]`** array will be copied into the specified row or column of the grid, in normal or reverse order. You may assume that the provided array is the correct size. For example:

```
> Board2048 test = new Board2048(new int[][]{{1,2,3},{4,5,6},{7,8,9}});
> test
1     2       3
4     5       6
7     8       9
> test.insertLine(new int[]{10,11,12},1,false,false); //Change row 1 to 10,11,12
> test
1     2       3
10    11      12
7     8       9
> test.insertLine(new int[]{13,14,15},2,true,true); //Change column 2 to 15,14,13
> test
1     2       15
10    11      14
7     8       13
```

8) Write a method **`public Board2048 shift(int direction)`** which will shift the entire grid in the indicated direction, producing a ***new*** **`Board2048`** object. The existing grid in the existing **`Board2048`** object ***should not be affected***. This method must use **`extractLine`**, **`alterOneLine`**, and **`insertLine`** to do all of the work. If the shift direction is LEFT or RIGHT, all ***rows*** should be shifted to the left or to the right, and if the shift direction is UP or DOWN, all of the ***columns*** should be shifted up or down. For example:

```
> Board2048 test = new Board2048(new int[][]{{2,0,2,4},{0,0,0,4},{2,2,2,2},{0,0,0,0}});
> test
2  -  2  4
-  -  -  4
2  2  2  2
-  -  -  -
> Board2048 test2 = test.shift(Board2048.LEFT);
> test2
4  4  -  -
4  -  -  -
4  4  -  -
-  -  -  -
> test //should be unchanged
2  -  2  4
-  -  -  4
2  2  2  2
-  -  -  -
> Board2048 test3 = test2.shift(Board2048.UP);
> test3
```

```
8  8  -  -
4  -  -  -
-  -  -  -
-  -  -  -
```

You can use the supplied **`TestPhase2.java`** program to run a short test on these functions.

Phase 3: Play The Game

Now add additional methods to make a playable version of the game.

1) Write a method **public int numEmpty()** which will find and return the number of empty spaces in the grid.

2) In every turn of the game, after the board has been shifted, a new tile with the value 2 or 4 is randomly added to the grid. Write a method **`public void newTile()`** which will add this new tile to the grid. *Every empty space must have an equal probability of being chosen as the place for the new tile.* **Hint:** You can use the **`numEmpty()`** method to choose an appropriate random number. If there are N empty places, pick a random number from 1 to N (or from 0 to N-1), and then put the new tile in the Nth empty place. You may assume that there will be at least one empty place in the grid. The value 2 should be used for the new tile 90% of the time, and the value 4 should be used 10% of the time.

```
> Board2048 test = new Board2048(3);
> test
-  -  -
-  -  -
-  -  -
> test.newTile();
> test
-  -  2
-  -  -
-  -  -
> test.newTile();
> test
-  -  2
-  4  -
-  -  -
```

3) Provide a public **`boolean equals(Board2048)`** method which will return **`true`** if two boards have identical grids.

4) Provide a **`public boolean validMove(int direction)`** method which will return **`true`** if a shift in the given direction is a valid move. The move is valid if it will cause any change at all in the grid. A move is invalid if the grid will remain unchanged. Hint: Use your **`equals`** method.

5) Provide a **`public boolean gameOver()`** method which will return **`true`** if the game is over. The game is over if there are no empty places in the grid, and there are no valid moves.

You can use the supplied **`TestPhase3.java`** program to play a complete game, using the console and text input. The program accepts the characters i,j,k,l or w,a,s,d to move up, left, down, and right, respectively. You can change the constant **`GAME_SIZE`** (line 4) to a smaller value to make it easier to test the **`gameOver`** method.

Phase 4: Add Undo and a GUI.

It is very useful to have an "undo" function in this game, and a real graphical user interface. In this phase you will add these to your game.

1) Add an accessor method **`public int[][] getMatrix()`** to your **`Board2048`** class, so that other classes will be able to obtain and display the data in the grid. You do not need to make a copy. Returning a reference to the existing matrix is good enough.

2) The provided class **`GraphicsWindow2048`** will provide a graphical user interface for your game. It requires the **`Board2048`** class and the **`Game2048`** class (described below). It contains a normal **`main`** method which you can run. It uses the same i,j,k,l or w,a,s,d keys to make moves, as it did for Phase 3 (but you don't have to press Enter any

more) and it uses the z key for "undo". You can look at the code to see how a "real" Java user interface works, but you are not expected to understand the code in this file. It provides two instance methods which you will have to use:

a. **`void displayBoard(int[][])`** will display the data in the given matrix in a graphical window. You should call this method whenever your board changes.

b. **`void displayMessage(String)`** will display the given message in the centre of the window, using transparent text. You can use this to display "Game Over" or similar messages. An empty String or a null will remove any previously displayed message.

3) Implement a new class **`Game2048`** which will interact with the **`GraphicsWindow2048`** class, and also provide an undo function. Start with the supplied **`Game2048`** file which contains just enough code to allow it to be compiled. Remove the temporary code and add your own.

a. Provide instance variables which are suitable for storing a list of **`Board2048`** objects. A simple partially-full array can be used. You can use an **`ArrayList`** if you know how to use them. When a game is played, a complete list of all of the boards should be stored, so that you can go back to any point in the game by using an undo. (This is why the **`shift`** function in Phase 2 always created a new **`Board2048`**).

b. Add a **`GraphicsWindow2048`** instance variable which will contain a reference to the window being used to play your game.

c. Provide a constructor with the parameters **`(GraphicsWindow2048, int)`**. The first parameter is a reference to the window for your game, and the second is the desired size of the board. You should create the first board, including the first tile. You should display the initial board.

d. Provide a method **`public void tryMove(int direction)`** which will attempt to make the indicated move. If it is an invalid move, it should do nothing. If it is a valid move, it should make the move, add a new tile to the board, and display the new board. It should keep track of the new board in your list of boards. If the game is now over, it should display a "Game Over" message in the window.

e. Provide a method **`public void undo()`** which will undo the previous move, going back to the previous board. This should work any number of times, so that every move in the entire game could be undone. But you should not be able to go beyond the initial position created in the constructor.

Hand in

Submit your two Java files (**`Board2048.java`**, **`Game2048.java`**). ***Do not submit .class or .java~ files!*** You do ***not*** need to submit the **`TestPhaseN.java`** files or the **`GraphicsWindow2048`** file that were given to you. If you did not complete all four 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.

The test files supplied are ***not*** the only tests that the marker will run. Make sure you test your code completely.

Make sure none of your files specify a package at the top!