ADSB PS12: 2048

First things first:

We will be coding up our own, text-based version of the smash hit. Mucch like what we did for Asteroids, I would like you to first take a look at the actual game, and play it for 5 minutes, paying special attention to the way the “tiles” move. You will be writing 6 different classes, **Tile, Board, UserAction, Game2048View, Game2048Controller and Game 2048**. Some will have much more code than others, as you will see soon. Before you start, make sure you name your project “ADSB PS12 2048” and your package “mr.gonzalez.<YOUR CSID>”

Take a look at this lovely little video and get a little familiar with the idea of MVC. MVC is a design pattern that stands for model-view-controller. Here's the short version: It can be nice for us to segregate the **data/model** of a program, the **view** of a program (i.e. what users see and interact with) and the **controller** of a program (the logic that "drives" the program). Designing with this structure in mind means that we could later come back to our program and easily swap out our graphical elements with no problem.

Have Fun!

Task 1:Tile.java

This will be our representation of each numbered tile in the game.

Instance variables

Just one: its value, an int

Constructors

Self-explanatory. Calling the default constructor should create a zero tile. Trying to create a negative Tile results in a 0 tile.

Other methods

A Tile's string representation should just be a String containing the numerical value of the tile; the only exception is a zero tile, which should be represented by a single underscore, a la: "_"

Task 2: Board.java

This is our representation of an entire Board. It will always be a square.

Instance variables

The dimension of the board, a 2D array of Tiles, and the current score

Constructors

Score starts as 0 except in the constructor that takes a score. The default board side length should be 4. If given a filename as a parameter, you should look for that file in the working directory and load a board state from it. The file should be named something like "boardstate.tfe" and we will define a .tfe ("twenty forty eight") be of the following format:

```
<score>
<some dimension n>
<n rows of n columns, separated by spaces>
```

So, for example:

```
0
4
2 4 2 2
4 2 0 2
0 0 2 2
4 0 2 4
```

This constructor should help you in testing if you'd like to load a particular game "in the middle" without having to reproduce the steps to get there.

Other methods

- **loadBoardState** - likely to be called by the filereading Constructor.
- **reset** - sets all Tiles to 0; reset score to 0
- **countEmptySpaces** - count zero tiles
- **getScore** - getter for score
- **toString** - return the entire board as a String, each Tile separated by a tab.
- **isFull** - true if there are no zero tiles; false otherwise. Consider calling countEmptySpaces
- **addTile** - adds one random tile to the board in a random empty location. 90% of the time, the new tile should be a 2; 10% of the time, it should be a 4.
- **rotateCW** - returns void; rotates this Board's 2d Tile array 90 degrees clockwise. It should modify this board itself.
  Example:

  Rotating a board with values...

  ```
  512 256 64 16
  128 32 8 4
  8 16 4 2
  2 8 2 4
  ```

  ...should return a new Board holding these values:

  ```
  2 8 128 512
  8 16 32 256
  2 4 8 64
  4 2 4 16
  ```

Perhaps this seems an odd capability for us to want, but it's actually wonderfully useful. **Think**: Suppose we implement a leftward shift of all the tiles. Rather than having to re-implement a downward shift, we could just rotate once CW by 90 degrees and do a leftward shift, and then rotate until we were back to our original configuration.

This saves us some work, since now we don't need to implement separate "shift right," "shift down," and "shift up" methods. We just need a "shift left" method and a reliable "rotate" method that can transform a new problem (right/down/up) into a problem we've already solved (left). *(It also accomplishes the nice little goal of practicing with 2D arrays.)*

- **canMove** - examines whether or not there is at least one possible move on the board as it is. If there ARE possible moves, it returns true; else false.
  If the board is not completely full, you can definitely move. Else, you have to check whether there are any moves possible. Read the next hint if you're desperate: You know you can move if you find any adjacent Tiles that have the same value.

- **left** - check each row and perform a left shift. Make sure you start looking for pairs to merge starting from the left end of each row; if a Tile is used in a merge, then it cannot be used again in the same shift.
  Example: Given row 2 2 4 _, if you are a performing a left shift, the 2s would merge to form a 4 but then would NOT be eligible to merge with the pre-existing 4. You would NOT get 8 _ _ _. Rather, you'd get 4 4 _ _.

  Return false if nothing changed after the "move." Return true if something changed. *[h/t NS]*

- **right** - combine clockwise board rotations and a left shift to make this work. DON'T manually re-implement the shift. Return false if nothing changed after the "move." Return true if something changed. *[h/t NS]*
- **up** - combine clockwise board rotations and a left shift to make this work. DON'T manually re-implement the shift. Return false if nothing changed after the "move." Return true if something changed. *[h/t NS]*
- **down** - combine clockwise board rotations and a left shift to make this work. DON'T manually re-implement the shift. Return false if nothing changed after the "move." Return true if something changed. *[h/t NS]*
- **copyBoard** - return a deeply cloned copy of this Board object.

Task 3: UserAction.java

Now our model is done. We're about to define our View and our controller. We know that our View is going to take some input from the user, and that means we'll have to have a good way to pass that information about the user's actions between the view and the controller.

For now, let's do this using an enum. Click here to read about Java enumerated types. They're a way for us to define a quick set of constants. For 2048, let's define an enumerated type called UserAction. It should contain these constants: LEFT, RIGHT, UP, DOWN, QUIT, RESET, INVALID. Define your enum in its own .java file. Make sure that your project is set to compile at level 1.6 or higher. This will be very short.

Task 4: Game2048View.java

OK, time for the view. We know our view needs to have somewhere to store the board, and we know that when we first make the view, we'll give it the board as it is at that moment. We'll also need a method to let us update the board based on some parameter, and we'll need a method that gets a UserAction from the user and returns it. Finally, the view needs to be able to display the current state of the model.

Instance variables

The Board

Constructors

The constructor should always take a Board object representing the state of our model at the moment of instantiation.

Other methods

- **setBoard** - update our board instance variable with the most current version of the board.
- **getUserAction** - Prompt the user to type in "Next action: [W]Up, [A]Left, [S]Down, [D]Right, [Q]uit, [R]eset: " and return the appropriate UserAction enum value.
- **updateDisplay** - output the current sate of the board to the console. (i.e. print it out.)

Task 5: Game2048Controller.java

Almost done. The controller coordinates between our model and view.

Instance variables

The controller should keep track of both the board and the view.

Constructors

The controller should have a constructor that takes a Board object and a Game2048View object, which will both be instantiated from our main application class.

Other methods

- **runGame** - This is the beating heart of the game. It should probably run in a cycle resembling...
  - As long as I haven't lost...
    - get the UserAction from the view
    - Based on the action, update the model
    - Send the updated model to the view
    - Update the view's display

Game2048.java

Barely anything here. This is the main class, and it contains your main method. Three instance variables: a Board, a Game2048View, and a Game2048Controller.

Instantiate your Board, instantiate your View (and give it your board!), and then instantiate your controller, giving it both. Then call your controller's runGame method and let 'er rip!