# Week 4 Discussion

Wednesday, 1/29/20

# Reminders

PA 4 due **Thursday, 2/6**

Quiz during discussion on **2/12**

# Today’s agenda

**PA 4 Overview**

- 2048
- Random class
- Enum

Required vs optional methods

# Demo

Link to game

# Config.java

- This file contains constants and settings for playing the game.
- You can modify the values of some of the constants for testing purposes but **DO NOT ADD** any additional constants.

# Enumeration (Enums)

- An `Enum` is a special Java type used to define collections of constants.
- An `Enum` can contain constants, methods etc.
- They are different from `static final` constants.

```
public enum Level {
  HIGH,
  MEDIUM,
  LOW
}

Level level = Level.HIGH;


// assign some Level constant to it
Level level = ...
if( level == Level.HIGH) { ... }
else if( level == Level.MEDIUM) { ... }
else if( level == Level.LOW) { ... }
```

```
for (Level level : Level.values()) {
  System.out.println(level);
}

Output:
HIGH
MEDIUM
LOW
```

# Direction.java

- This file contains an `Enum`
- Four direction constants
  - DOWN  (0)
  - LEFT  (1)
  - UP    (2)
  - RIGHT (3)
- What does LEFT (1) mean?
  - Define a constant named LEFT and initialize it with the constructor that takes in a single int.

# java.util.Random

- Allows you to generate pseudo-random numbers
- How can we test our code if it is generating random values?
  - Pass in your own seed
- Example:
  - `Random rand = new Random(123);`
  - `System.out.println(rand.nextInt(10)); // outputs 2`
  - `System.out.println(rand.nextInt(10)); // outputs 0`
  - `System.out.println(rand.nextInt(10)); // outputs 6`
- You are free to specify a seed in your code to test your methods.
- We will **not** be deducting points for using your own seed for your Random objects.

# GameState.java

- You will write this class from scratch.
- Requirements:
  - Class declaration: public class **GameState**
  - Must follow all the method signatures in the writeup
- No restrictions on instance variables
  - Suggestions from writeup:
    - private Random rng;
    - private int[][] board;
    - private int score;

# Constructor [REQUIRED]

- **public GameState (int numRows, int numCols)**
  - Create a board (must be a 2d array) with `numRows` rows and `numCols` columns.
  - Set your starting score to 0.
  - If random object is created here, `RANDOM_SEED` in `Config.java` may be helpful

# Getters/Setters [REQUIRED]

- `int[][] getBoard()`
  - Return a **deep copy** of the board
- `void setBoard(int[][] newBoard)`
  - Make a **deep copy** of the board and set the board
- `int getScore()`
- `void setScore (int newScore)`

# `toString()` [REQUIRED]

- Code is given in starter code for consistency

- Note: the `board` variable should be whatever your `getBoard()` returns
- If you do not represent your board in this way, you should change all instances of `board` to `getBoard()`.

# Generating tiles

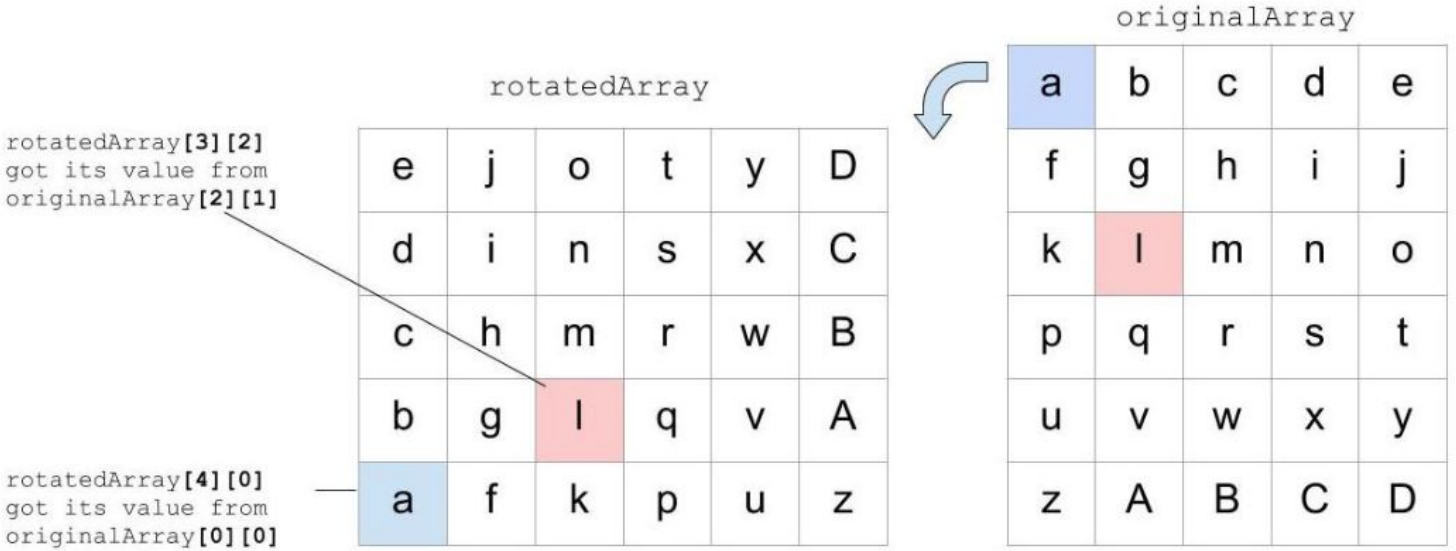
- `int rollRNG(int bound)` [Optional]
  - Return a random integer between 0 (inclusive) and bound (exclusive)
- `int randomTile()` [REQUIRED]
  - Either 2 or 4 tile will be added.
  - Hint: In Config.java, `TWO_PROB` means the probability out of 100 times. How can we use the Random object for this?
- `int countEmptyTiles()` [REQUIRED]
- `int addTile()` [REQUIRED]
  - Randomly choose a tile location and place a tile
  - **What happens if the board is full?**
  - Return the tile value you just added

# Movement

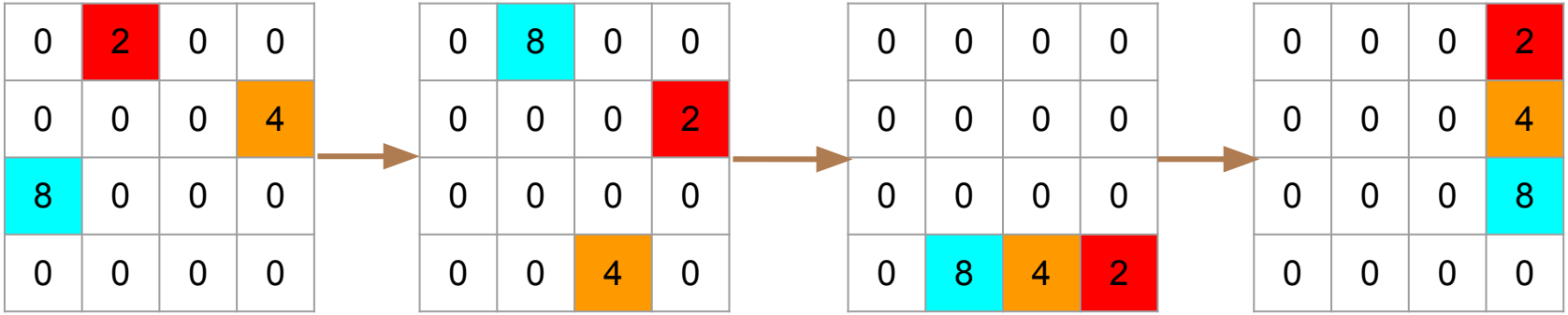
# `move(Direction dir)` [REQUIRED]

- Move the board in the direction `dir`.
- If movement is **successful**, add a tile using `addTile()`
- Multiple ways to move the board
  - Some ideas:
    1. Implement `canSlideDown()`, `slideDown()`, board rotation
    2. Using equals to implement `canSlideDown()`, `slideDown()`
- Feel free to define any private helper methods

# rotateCounterClockwise() [Optional]



# Why rotate?




how do we move right?

rotate the board counterclockwise 3 times

slide down

rotate the board counterclockwise 1 time

You can follow this process to move in the other directions

# `isGameOver()` [REQUIRED]

- Return true if the board cannot be moved in any direction
- `canSlideDown()` and `rotateCounterClockwise()` can be helpful

# canSlideDown() [REQUIRED]

- Checking whether the board can slide down
- Two cases where sliding is possible:
  - If two tiles can be combined. Two tiles can be combined if they share the same value and there are no other tiles between them (empty tiles can be between them).
  - If there is an empty tile below a non-empty tile.
- How? A loop? Or a smarter way you can think of?

# slideDown() [REQUIRED]

- Slide the board down and return whether the sliding was successful (if it changed the board).
- Empty spaces can be represented as 0s.
- Tiles slide all the way, unless it makes contact with other tiles or reaches the edge of the box.
- Few things to check:
  - Combine if two equal tiles are side by side
  - Once one tile is merged, it cannot merge with another tile.
  - Move if there is empty space
  - Update the score
- 4 4 4 4 -> 0 0 8 8

# Worksheet