PSA 3: 2048

Checkpoint Submission: **01/27/18**
Final Submission: **02/04/18**

Commendations on making it to PSA3! By this time, you have gained extensive experience writing code, testing code, and debugging code. You have written code towards a complete program, or finished implementing one through the cdmobile game, 2048. Make sure to read the whole document before starting. Please be at/view the discussion section.

Helpful Information:

- **Online Communication: Using Piazza, Opening Regrade Requests**
- **Getting Help from Tutor, TA, and Professor's Hours**
    - **Lab and Office Hours** (Always refer to this calendar before posting.)
- **Academic Integrity: What You Can and Can't Do in CSE 8B**
- **Setting Up the PSA**
- **How to Use Vim**
- **Style Guidelines**
- **Submitting on Vocareum and Grading Guidelines**

Table of Contents:

Part 1: Vim and Unix Commands [6 Points]

Part 2: 2048 Methods [70 Points]
- **A. Board's two constructors (10 points) - DUE by Saturday 1/27 Checkpoint**
- **B. Saving the Board (10 points in total)**
- **C. Adding tiles to the Board (10 points in total) - DUE by Saturday 1/27 Checkpoint**
- **D. Determining whether a move is possible**
- **E. Executing a move**
- **Board.java: public boolean move(Direction direction) [10 points]**
- **F. Checking if the game is over**
- **G. Two Constructors of the GameManager (10 points in total)**

Example Output

Part 3: Summary [4 Points]

Style Guidelines (Link) [20 Points]

Extra Credit: Record and Replay
- **Record Mode and Replay Mode**
- **Implementation Changes to Game2048.java**
- **Implement Recording:**
- **Implement Replaying:**

Submitting the Assignment (Link)

Appendix

Overview

2048 is a single-player puzzle game created in March 2014 by 19-year-old Italian web developer Gabriele Cirulli. He created the game in a single weekend to see if he could program a game from scratch. 2048 was an instant hit when the game received over 4 million visitors in less than a week.

We are implementing the game verbatim. This means it's important to know how to play the game!

http://2048game.com/

For now, our version of 2048 will be text-based and played in the Terminal. In a future assignment, we will actually implement the beautiful and reactive front-end as played in the web version.

Basic Game Operation

2048 is played on a simple grid of size NxN (4x4 by default), with numbered tiles (all powers of 2) that will slide in one of 4 directions (UP, DOWN, LEFT, RIGHT) based on input from the user. Every turn, a new tile will randomly appear in an empty spot on the board with a value of either 2 or 4 (determined randomly with a 90% probability of being a tile with value 2). Tiles will slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with a total value of the two tiles that collide. For example, if two tiles of value 8 slide into one another the resulting tile will have a value of 16. The resulting tile cannot merge with another tile again in the same move.

A score will also be kept based on the user's performance. The user's score starts at zero, and is increased whenever two tiles combine, by the value of the newly combined tile. For example, if two tiles of value 8 are merged together, the resulting tile will have a value of 16 and the user's score will be increased by 16.

For more information on the game, see the Appendix section of this document.

Part 1: Vim and Unix Commands [6 Points]

In a file called **README.md**, answer the following questions with short answers.

Vim related Questions:

1. How do you jump to a certain line number in your code with a single command in vim? For example, how do you jump directly to line 20? (Not arrow keys)
2. What is the command to undo changes in vim?
3. How do you search for all occurrences of a word in vim?
4. How do you fix the indentation of your code in vim in a single command?
5. How do you switch between two files that are opened in a single vim editor?
6. a. In vim, in command mode, how do you move forward one word in a line of code?
   b. Move back one word? (Not arrow keys)

Unix/Linux related Questions:

7. How can you remove all .class files in a Unix directory with a single command?
8. a. How do you remove a Unix directory when the directory is empty?
   b. How do you remove a directory when it is not empty?
9. List two commands to clear a Unix terminal screen.
10. What are swap files? Why do they exist? How do you restore from a swap file? How do you delete a swap file?

Part 2: 2048 Methods [70 Points]

You will need to implement the following methods in **Board.java** and **GameManager.java**. The Board class is intended as the fundamental board where tiles will be placed and moved around, while the GameManager class will connect the movements instructed by the player as input in the Board class.

Read the starter code thoroughly, understand the overall structure of the program before getting started.

Board.java contains these instance variables**:**

private final Random random;	// a reference to the Random object, passed in as a parameter // in Board's constructors
private int[][] grid;	// a 2D int array, its size being boardSize * boardSize
private int score;	// the current score, incremented as tiles merge in gameplay // see move() for how to increment this score

In Board.java, implement the following methods:

A. Board's two constructors (10 points) - DUE by Saturday 1/27 Checkpoint

`public Board(Random random, int boardSize)      [5 points]`

This constructor is called when we want to create a new board for a new game. You should initialize instance variables for the Board object, create the grid (a 2D array), and add NUM_START_TILES (which is 2) random tiles by calling addRandomTile();

`public Board(Random random, String inputBoard)   [5 points]`

This constructor loads a saved board from the file specified by the string inputBoard. inputBoard will never be a null string. The board size, score, and grid will all be need to be initialized based on the input board. You may assume the input file is in the correct format. The format of the board file is in described below.

Format of a board file:	Example:	Another example:
[Filename (not written inside the file)]	2048_4x4.board	2048_6x6.board:
[Grid Size]	4	6
[Score]	224	1212
[Tile Value] [Tile Value] [Tile Value] [Tile Value] ..	`2 16 0 0`	`4  2  64  4 0 2`
[Tile Value] [Tile Value] [Tile Value] [Tile Value] ..	`16  4 2 0`	`8  4   2 16 0 0`
[Tile Value] [Tile Value] [Tile Value] [Tile Value] ..	`4 32 0 0`	`4 16 128  8 0 0`
[Tile Value] [Tile Value] [Tile Value] [Tile Value] ..	`8  0 0 2`	`2  8   4  0 0 0`
..		`4 16   0  0 2 0`
		`0  0   0  0 0 0`

B. Saving the Board (10 points in total)

`public void saveBoard(String outputBoard)   [10 points]`

This method is designed to save your current board to a file. The parameter, String outputBoard, is the name of the file you want to save your board to. This parameter will never be null. This file may or may not exist, but you do not need to worry about that.

> *Use the PrintWriter Class as you did in PSA2 to write to a file.* Some useful methods might be print() and println(). For more information on PrintWriter, you can refer to the textbook or to the javadocs. Remember to **close** the file when you're done, otherwise none of the writes will actually be saved to the file!
>
> Refer to the example above for the format of the output board.

C. Adding tiles to the Board (10 points in total) - DUE by Saturday 1/27 Checkpoint

`public void addRandomTile()   [10 points]`

This method is designed to add a random tile (with value of 2 or 4) to an open spot on the game board. If there are no open tiles, it should just return without changing the board. To ensure your code can be tested, you need to follow the algorithm below carefully in the same order.

1. For all tiles on the board, count the number of available tiles (called count)
    a. if count is 0 then there is no room to add a tile, so return.
2. Get a random int called location between 0 and count-1
3. Get a random int called value between 0 and 99 **(MAKE SURE YOU DO STEP 2 BEFORE YOU DO STEP 3)**
4. Walk the board row first, column second keeping count of the empty spaces you find. When you hit the i'th (the random int you just got between 0 and count -1) empty spot, place your new tile
    a. if "value" is <TWO_PROBABILITY, place a 2
    b. else place a 4

For example, if your game board is the following

```
- - 2 -
- - 4 -
8 4 2 2
- 2 4 2
```

You should imagine the open tiles as numbered as shown in red below:

```
0 1 2 2
3 4 4 5
8 4 2 2
6 2 4 2
```

If location is 1, a random tile should be placed in the red location 1 above. Etc.

- Be sure you are using the **same numbering scheme** (**English reading order)** as above.
- Be sure you only call random.nextInt(n) **exactly twice** in addRandomTile()
- Make sure you are using the Random object from **the instance variable**.

You can find documentation for Random class here. One useful method is nextInt(int n).

D. Determining whether a move is possible

`Board.java:  public boolean canMove(Direction direction)  [5 points]`

This method returns true if there exists a tile that can move in the passed in direction. Return false if no tiles can move in that direction.

A tile "can move" in such direction if:

a. It can move into an empty space in that direction
b. It can merge with an adjacent tile with the same value in that direction

For example:

```
2  16   8   4
16  4   2   8
4  32  16   0
8   4   8   2
```

The above board can move right as the 16 can move into the 0 space.

```
2  16   8   4
16  4   2   8
4  32   8  16
8   4   8   2
```

The above board can move down and up as the 8 can move into the 8 space to combine to form a 16.

We recommend that you break the problem into smaller pieces. One way that you can do this is by writing a specify helper method for each direction, such as:

    private boolean canMoveLeft();

E. Executing a move

Board.java: public boolean move(Direction direction) [10 points]

This method will actually perform the main game logic and perform a move. You will return true if the move was successful and false if it was not. This method will be called in play().

You SHOULD call canMove() method ONLY inside the move() method and not inside the play() method.

There are various algorithms that can be used to perform the move() operation. For the move() method, we again suggest that you break the problem into smaller pieces and write helper methods so that you can focus on moving one direction at a time. You can learn more about the movements for the game by playing it. Your 2048 game MUST perform identically to the original game in all cases.

To give you a head start, play the game and see what happens to your first row when it is one of the following cases (you are about swiping horizontally):

```
[0 0 2 0];[2 2 2 2];[2 0 2 0];[2 2 0 2];[2 0 2 2];
```

F. Checking if the game is over

`Board.java:  public boolean isGameOver()  [5 points]`

This short method returns true if the board cannot move in any direction.

In GameManager.java, implement the following methods:

G. Two Constructors of the GameManager (10 points in total)

`public GameManager(String outputBoard, int boardSize, Random random)   [5 points]`

This constructor needs to initialize all instance variables for the GameManager class. This constructor will create a new board with a grid size corresponding to the value passed in the parameter boardSize.

* BoardSize will always be an integer greater than or equal to 2. A null string will never be passed in for outputBoard. So you don't have to consider this edge case.

`public GameManager(String inputBoard, String outputBoard, Random random)   [5 points]`

This constructor will load a board using the filename passed in via the inputBoard parameter. It will initialize all instance variables for the GameManager class. A null string will never be passed in for inputBoard, nor for outputBoard. So you don't have to consider this edge case.

H. Play Method of the GameManager (10 points in total)

`public void play()  [10 points]`

This is the main play loop for the 2048 game.

This method will begin by printing out the controls used to operate the game (to move left, right, up and down) - The method printControl does this and is already implemented for you. Then this method will print out the current state of the 2048 board and then prompt the user for a command. If the user enters a valid move (w,a,s,d), then we will perform that move and add a new random tile to the board. If that move is not valid, do not add a new random tile. Then print the updated board and prompt the user for another command.

If the user decides to quit with command (q) or the game is over, then we will save the board to the output file specified by the instance variable outputFileName and then exit the method. If any invalid commands are received from the user then you will need to print the controls again and then prompt the user for another command. If isGameOver() returns true you will need to print the string “Game Over!”.

So to summarize, you should complete the part of reading in user input, verifying input validity, and call appropriate methods at the right place to build a correct structure for the play method.

Example Output

```
Welcome to 2048!
Generating a New Board
  Controls:
    w - Move Up
    s - Move Down
    a - Move Left
    d - Move Right
    q - Quit and Save Board

Score: 0
    2    -    -    2
    -    -    -    -
    -    -    -    -
    -    -    -    -

 > a
Score: 4
    4    -    -    -
    -    -    2    -
    -    -    -    -
    -    -    -    -

 > f
  Controls:
    w - Move Up
    s - Move Down
    a - Move Left
    d - Move Right
    q - Quit and Save Board

Score: 4
    4    -    -    -
    -    -    2    -
    -    -    -    -
    -    -    -    -

 > q
```

Part 3: Summary [4 Points]

We want to read about **how you approached implementing the program** in your Java files. In README.md, below the Vim and UNIX answers, provide a description for each file you worked on. For example, Board.java is a file you worked on, but not Direction.java. There are two Java files you worked on, so there are two program descriptions to write about. The program descriptions may use CS terms.

We want to read **a summary about what the entire program can do**. Sure, we are programmers, but as people, what can we do now that we have these programs? Here, do not use any CS terms. Explain simply, like the reader is five years old.

Points will be allotted for both clarity and conciseness.

Style Guidelines (Link) [20 Points]

#1-9 on the website. We will introduce #10 in a future discussion section.

Extra Credit: Record and Replay [+10 Points]

We've finished implementing a program that behaves different depending on user inputs. Some of these types of programs take days to run. Other programs run forever, until the server crashes or the program is stopped manually. For example, websites are hosted on web servers, which run indefinitely. ieng6 runs indefinitely. But any computer can suddenly crash for any reason.

Being responsible for our programs, we want to know what happened when it crashed. Even better is the ability to recover the program up to the point when it crashed, exactly as it had before, based on user input.

For the extra credit, we'll implement a way for 2048 to **log** the user input into a **logfile** and literally **replay** the game using that same **logfile**. We simplify the use case: we will never test that the program crashes or has any negative performance issues. We will assume that the gameplay works fine. See this video for a demonstration. If you attempt this extra credit, you must state this in the README.md.

Record Mode and Replay Mode

[*Record Mode*] To start 2048 such that it begins to record, run the program: `java Game2048 --record logfile`

where logfile is the name of your file containing the logs.
Note that if we use the "--record" flag, we can still use "-s" to specify a board size, but not "-o" anymore.

[*Replay Mode*] To replay 2048 from the logfile, run the program as follows: `java Game2048 --replay logfile`

Note that using the "`--replay`" flag will nullify any other flags, and any future record will be saved to the same log file.

After 2048 finishes replaying back to the saved point, you will be able to continue playing the game. The log will continue to save the moves. When you quit and replay the same logfile, it will start from the beginning and replay both the original moves *and* the newly played moves. This can go on forever.

Implementation Changes to Game2048.java

In Game2048.java, uncomment lines labeled "Uncomment these lines..." to support "`--record`" and "`--replay`" flags

Implement Recording:

In GameManager.java, add the following constructor:

```
public GameManager(Random random, int boardSize, String outputRecord)
```

Implement this new constructor to take in a Random object, a board size, and a String for outputRecord, which is the name of a file we will be storing the board size and keystroke record (as opposed to outputBoard which stores the board)

Add code that will open a file for writing. Previous content in the log file should be overwritten. The name of the log file is given in the outputRecord variable. Write the board size and then a space. The result of that would be a String with a length of 2. Close the file.

Modify GameManager.play():

Currently, whenever play() receives a valid command, it will then move the board depending on the character passed in. In between, before calling move(), open the log file for **appending**, then write the command, then close the file. That means every time the player makes a move, the file is appended with that move.

For example, in one game of size 4, the player moves up, then down, then left, then outputRecord should look like:

```
4 wsa
```

Implement Replaying:

In GameManager.java, add the following constructor:

```
public GameManager(Random random, String inputRecord) throws Exception
```

Implement this new constructor to take in a Random object and a String for inputRecord. This constructor should:

1. Read the first number in inputRecord as GRID_SIZE, and construct a Board of that size
2. Read the next string, replay each character by moving the board to the corresponding direction, each move separated a half second interval. The following line is useful
   `Thread.sleep(500);` // pause for 0.5 seconds
3. After replaying steps, the program should go into play() normally so you can keep on playing the same game. The program should continue to append the player's moves into the log file such that when you replay the game again, it will replay the original moves with the new moves.

Modify GameManager.play():

1. If the game is started by reading from an inputRecord. At the end of the game, it should store the whole history back into the same inputRecord file. For example, if in the beginning of the game the program reads from inputRecord
   ```
   4 wsa
   ```

   Then the player moves up, then down, then right. Then the player puts 'q' to end the game, inputRecord should become:

   ```
   4 wsawsd
   ```

We will look at the functionality of the record and the replay, rather than the actual contents of the log file. Your log file can look different as long as the functionality is the same. If you attempt this extra credit, remember to state this in your README.md.

Submitting the Assignment (Link)

Checkpoint and Final Submission Deadlines

This assignment is due on **Sunday 02/04/18**. There is a **mandatory** checkpoint due on **Saturday 01/27/18**. **Parts 2A and 2C** will be graded by the version submitted for the checkpoint! Be certain to submit this checkpoint in the **PSA 3: Checkpoint Submission** on Vocareum. Incorrectly implemented methods submitted to the Checkpoint still need to be fixed for the Final Submission, but the grade for those methods are determined by the Checkpoint autograder.

Submission Files

Make sure your submission contains all files, compiles, and runs on the ieng6 lab machines. We grade via Oracle Java 8 on Vocareum and ieng6.

- `README.md`
- `Board.java`
- `GameManager.java`

Maximum Score Possible: 110/100 Points

Appendix

Command Line Arguments

Your 2048 game will have several different arguments that can be passed in via the command line. Game2048.java will handle all of the command line argument processing and will simply pass the proper arguments to the GameManager constructor that you will be writing. You will be able to pass in parameters to specify the size of the grid of the board, the name of the output file that should be used to save the 2048 game board upon exit, and the name of the input file that should be used to load an existing board that will be used for play.

Ex:
```
> java Game2048 -o my2048.board -i input2048.board
> java Game2048 -s 5
> java Game2048

-i <filename>       - used to specify an input board
-o <filename>       - used to specify where to save the board upon exit
-s <integer>        - used to specify the grid size
```

Any combination of these command line arguments can be used in any order or you may use none of them! You can also specify all three. You can assume that if an input file is specified that it will conform to common file format which will be covered later. **If no size is specified (or a value that is less than 2 is specified) then the default grid size of 4 should be used.** If both an input file and size is specified, then the board size specified by the command line argument (via -s) should be ignored and the board size that should be used will be loaded from the file. This means there is no reason to specify a board size as well as an input file. If no output file is specified then the default file “2048.board” will be used to store the board.

Saved Board File Format

Your 2048 game will have the capability of loading an existing board from a file as well as saving the state of the board before quitting. The following is the format of these board files.

[Grid Size]
[Score]
[Tile Value] [Tile Value] [Tile Value] [Tile Value]
[Tile Value] [Tile Value] [Tile Value] [Tile Value]
[Tile Value] [Tile Value] [Tile Value] [Tile Value]
[Tile Value] [Tile Value] [Tile Value] [Tile Value]

Here are a few examples:

2048_4x4.board

```
4
224
 2 16 0 0
16  4 2 0
 4 32 0 0
 8  0 0 2
```

2048_6x6.board

```
6
1212
4  2  64  4 0 2
8  4   2 16 0 0
4 16 128  8 0 0
2  8   4  0 0 0
4 16   0  0 2 0
0  0   0  0 0 0
```

Note that the size of the board (Grid Size) is on the first line of the file. On the next line is the value of the score. The rest of the file contains the saved state of each of the tiles on the board. Each tile value is separated by a single space and each row of the board appears on its own line. **There is no space after the first two lines, a space after each row of the board's lines, and a newline after the last row of output!**

Direction Enum

The provided Direction.java file contains the definition of the Direction enumeration type, which will be used to represent the direction of a move in our 2048 game.

Enums are lists of constants. When you need a predefined list of values which do not represent some kind of numeric or textual data, you should use an enum. In our project, we want to be able to represent 4 distinct directions (UP, DOWN, LEFT, RIGHT) that could be used as a move in our game.

Variables of an enumerated type are "type-safe." This means that an attempt to assign a value other than one of the enumerated values, or `null`, will result in a compile error. The Direction enum can only have a value of UP, DOWN, LEFT, or RIGHT.

To create a Direction variable: `Direction dir;`

To assign **dir** the value of **UP**: `dir = Direction.UP;`

All values of an enumerated type can be accessed by using the following syntax:

```
EnumeratedTypeName.valueName
```

At some point we are going to be interested in determining what value an enumerated variable holds. To do this we can simply use the .equals() method that all objects have. So if I wanted to check to see if our variable dir is DOWN I could do the following:

```
dir.equals(Direction.DOWN)
```

This expression would return false since we previously set dir to hold the value UP. But if we instead did this:

```
dir.equals(Direction.UP)
```

Then the expression would return true since our variable dir indeed holds the value UP.

As an extra component the Direction enum also has two data fields (X and Y) to signify a direction unit vector. The x and y components of this vector can be retrieved via the getter methods (getX() and getY()). This vector represent the direction of motion with respect to the indexes of the rows and columns. This makes more sense if we take a look at a sample board with the rows and columns labeled on the grid:

```
Sample Board          Direction (X, Y)
   0  1  2  3         UP (0, -1)
0  -  -  -  -         DOWN (0, 1)
1  -  -  -  -         LEFT (-1, 0)
2  -  -  -  -         RIGHT (1, 0)
3  -  -  -  -
```

NOTE: You do not need to use the components of the Direction enums to complete the assignment. They simply exist for convenience sake.

For more information on Enumeration Types please refer to Appendix I in your textbook.

Using Random Class

You will use the nextInt(int n) method from the Random Java API:

http://docs.oracle.com/javase/7/docs/api/java/util/Random.html

int	**nextInt**(int n) Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

Note that the Game2048 passes your constructor a reference to a Random object. By seeding this Random object with a fixed value, you will always get the same random numbers (in sequence). You will almost certainly want to use this for debugging and we will use this to test your programs for proper behavior. To set a random seed in the tester, you would replace the code below:

new Random() with:

new Random(<SEED>)   where <SEED> is an integer.