PA3: 2048 Part 1(100 Points)

Due: 11:59pm, Thursday, January 29th

Overview

2048 is a single-player puzzle game created in March 2014 by 19-year-old Italian web developer Gabriele Cirulli, in which the objective is to slide numbered tiles on a grid to combine them and create a tile with the number 2048. Cirulli created the game in a single weekend as a test 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.

The original version of the game was programmed in JavaScript. You will be creating a Java version of the 2048 game that can be played in the terminal (text based). Here is a link if you don't know of this game http://2048game.com/. Your version of the game will have the exact same functionality as the original version of the game.

For Part 1 you will be implementing the infrastructure of the game. In Part 2 you will build upon Part 1 and design the movement logic of the game.

Setup

In all of the following, the > is a generic command line prompt (you do not type that). You will need to create a new directory named **pa3** in your cs8b home directory on ieng6.ucsd.edu and copy over files from the public directory.

```
> cd
> mkdir pa3
> cp ~/../public/pa3/* ~/pa3/
```

Your pa3 directory should now contain the following files:

// code files

Game2048.java	// do not change
Direction.java	// do not change
GameManager.java	// the class that will manage the gameplay
Board.java	// the 2048 board you will implement

// sample 2048 board files, will not be submitted

2048_4x4.board	// sample 2048 board with a grid size of 4
2048_6x6.board	// sample 2048 board with a grid size of 6
seed2015_0.board	// Files for testing
seed2015_1.board	seed2015_4.board
seed2015_2.board	seed2015_5.board
seed2015_3.board	seed2015_6.board

README (10 points)

You are required to provide a text file named **README**, NOT Readme.txt, README.pdf, or README.docx, etc. with your assignment in your pa1 t directory. There should be no file extension after the filename "**README**". Your README should include the following sections:

Program Description (3 points) :
Describe what the program does as if it was intended for a 5 year old or your grandfather. Do not assume your reader is a computer science major.

Short Response (7 points):

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 command sets auto-indentation in vim? What command turns off (unsets) auto-indentation in vim?
3. What is the command to undo changes in vim?
4. In vim, in command mode, how do you move forward one word in a line of code? Move back one word? (Not arrow keys)

Unix/Linux related Questions:

5. How can you remove all .class files in a Unix directory with a single command?
6. How do you remove a Unix directory? Define the commands you would run on the terminal when the directory is empty and when it is not empty. How do these command(s) differ?
7. What is the command to clear a Unix terminal screen?

Style (20 points)

You will be graded for the style of programming on this assignment. A few suggestions/requirements for style are given below. These guidelines for style will have to be followed for all the remaining assignments. Read them carefully.

- Use reasonable in-line comments to make your code clear and readable.
- Use class headers and method header blocks to describe the purpose of your program and methods. Also, use file headers.
- Every time you open a new block of code (use a '{'), indent farther. Go back to the previous level of indenting when you close the block (use a '}').
- Keep all lines less than 80 characters. Use 2-3 spaces for each level of indentation. Make sure each level of indentation lines up evenly.
- Use reasonable variable names.
- Use static final variables to make your code as general as possible.
- Judicious use of blank spaces around logical chunks of code makes your code much easier to read and debug.

- Do not use magic numbers or hard-coded numbers. This means that if you want to use a number other than 0, -1, or 1, you must give it a variable name. This is so that your values are understandable and also can be changed later if need be.
- Always recompile and run your program right before turning it in, just in case you commented out some code by mistake.

You will be specifically be graded on commenting, file headers, class and method headers, meaningful variable names, sufficient use of blank lines, not using more than 80 characters on a line, perfect indentation, and no magic numbers/hard-coded numbers other than zero, one, or negative one.

Direction.java and Game2048.java will not be graded on style.

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.

As mentioned earlier, your 2048 game will function exactly as the original game. So if you have any doubts as to how a corner case should be handled, take a look at the original game and see how it handles those corner cases.

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.

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 (N) is on the first line by itself. 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.

Direction Enum

The Direction.java file which is provided 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. For example, 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 known as "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. In other words, our Direction enum can only have a value of UP, DOWN, LEFT, or RIGHT.

If I wanted to create a Direction variable I could declare it by doing the following:

```
Direction dir;
```

If I then wanted to assign `dir` the value of UP I would do the following:

```
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.

Correctness (70 points)

Example Output - (note that your game won't move tiles yet)

```
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
    -    -    -    -
    -    -    -    -
    -    -    -    -

 > w
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
```

If You Input a Board:

```
> java Game2048 -i 2048.board
Welcome to 2048!
Loading Board from 2048.board
 Controls:
   w - Move Up
   s - Move Down
   a - Move Left
   d - Move Right
   q - Quit and Save Board

Score: 0
   -    -    -    -    -
   -    -    -    -    4
   -    -    -    -    -
   -    -    -    -    -
   -    -    -    2    -

>
```

---

5 points (both constructors)
GameManager.java: public GameManager(int boardSize, String outputBoard, Random random)

This is one of the constructors for the GameManager object. It will need to initialize all instance variables for the GameManger 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.

GameManager.java: public GameManager(String inputBoard, String outputBoard, Random random)

This is one of the constructors for the GameManager object. It will need to initialize all instance variables for the GameManager class. This constructor will load a board using the filename passed in via the inputBoard parameter. A null string will never be passed in for inputBoard, nor for outputBoard.

10 points
GameManager.java: public void play()

This is the main play loop for the 2048 game. This method will begin by printing out the controls used to operate the game. Then this method will proceed to print out the current state of the 2048 board to the console and then prompt the user for a command. If the user enters a valid move (w, s, a, d), we will proceed to check if that move is valid. If it is a valid move then we will perform that move (implemented in PA4 - you may assumed it happens successfully) and add a new random tile to the board. If that move is not valid then we will simply prompt the user for another command. A new tile will only be added to the board if the accepted move is valid. Before prompting the user for another command the updated board will be printed to the screen again. 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.

For this first part of 2048 you will not be required to implement the canMove(), isGameOver() and move() methods of the board class. However you will be required to use these methods while implementing play(). For right now these methods will do nothing and move() and canMove() will simply return true while isGameOver() will return false.

5 points
Board.java: public Board(int boardSize, Random random)

This is the constructor for the board when no input file has been selected. This constructor is passed in two parameters, an int boardSize and a Random random. You should assign these parameters to some of the public final variables at the top of your class. Then, you need to initialize the grid and score variables that are at the top of the class so that other methods can use them. After, you need to add a few starter tiles to the board. You will add a number of tiles based on the constant NUM_START_TILES. These tiles will be added to the board using the addRandomTile() method which you will be designing.

10 points
Board.java: public Board(String inputBoard, Random random)

This is a constructor for the board class which 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 initialized based on the input file. The format of the board file was described up above.

15 points
Board.java: public void saveBoard(String outputBoard)

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 a null string. This file may exist or not, but you do not need to worry about that as the object we list below will handle that for you.

As described above, but repeated again, here is what your outputted file should contain:

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

4 is the board size, 224 is the score, and the rest of the numbers are the representation of the actual board.

PrintWriter
Use a PrintWriter object to write to a file. The PrintWriter object contains methods which are eerily similar to some methods which you are already accustomed to, those being print() and println(). For more information on PrintWriter, you can refer to the textbook or to the javadocs.
The PrintWriter Javadocs: http://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html
You will want to create a PrintWriter object and use one of its methods to write to the file specified in the parameter. We would suggest not using the PrintWriter's method called write, but rather another method that PrintWriter has.

25 points
Board.java: public void addRandomTile()

This method is designed to add a random tile (either 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.

1. for all tiles on the board, count the number of available tiles (called count)
    a. if count is 0, exit.
2. get a random int called location between 0 and count-1
3. get a random int called value between 0 and 99
4. Walk the board row first, column second keeping count of the empty spaces you find. When you hit the location'th 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 accordingly:

```
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 green location 1 above. If location is 2, it should be placed in the green location above. Etc. **Be sure you are using the same numbering scheme (number open spaces row first, column second) as above. Be sure you only call random.nextInt(n) twice if there is an open location in the grid.**

Using randoms

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(0)

You can change 0 to be any integer to get different random values.

How To Test

We have included a few sample boards that can be used for reference for loading and saving boards (2048_4x4.board and 2048_6x6.board). In addition we included a sequence of boards which can be used to verify if your addRandomTile() method is functioning properly. To do this testing you will need to change the seed of the Random object in Game2048.java to be 2015. There are 7 files named seed2015_X.board from seed2015_0.board to seed2015_6.board. These represent the state of a new 4x4 board after X additional random tiles have been added to the board due to valid input from the user. You can use to diff utility to ensure that your boards match these examples.

For testing, you will always start from a new board using seed 2015. So your testing for this would take this form:
0. Start game, Save State. Check your save game against seed2015_0.board
1. Start game, make one valid move, save state. Check your save against seed2015_1.board
2. Start game, make two valid moves, save state. Check your saved state against seed2015_2.board
3. Repeat for 3-6 moves.

Make sure your output matches our output, including the screenshots. Double check the strings you print out, as well. If you have any questions about this, post on Piazza.

Turnin

To turnin your code, navigate to your home directory and run the following command:

```
> cse8bturnin pa3
```

You may turn in your programming assignment as many times as you like. The last submission you turn in before the deadline is the one that we will collect. Always recompile and run your program right before turning it in, just in case you commented out some code by mistake.

Verify

To verify a previously turned in assignment,

> **cse8bverify pa3**

If you are unsure your program has been turned in, use the verify command. We will not take any late files you forgot to turn in. Verify will help you check which files you have successfully submitted. It is your responsibility to make sure you properly turned in your assignment.

Files to be collected:

README
Game2048.java
Direction.java
GameManager.java
Board.java

The files that you turn in must be EXACTLY the same name as those above.

NO LATE ASSIGNMENTS ACCEPTED.
DO NOT EMAIL US YOUR ASSIGNMENT!
Start Early and Often!

Extra Credit

Extra credit will be given for starting PA4 (part 2 of PA3) early. You can earn up to a maximum of 5 points (5%) extra credit by finishing the methods below by the PA3 deadline. For this extra credit, these methods need to behave properly when tested.

Methods:	Extra Credit:
canMove(Direction d)	+3%
isGameOver()	+2%

THIS IS ONLY FOR THE EXTRA CREDIT

Board.java: public boolean canMove(Direction direction)

This method determines if the board can move in the passed in direction. A board can move in the passed in direction if any of its tiles can move into an empty space or combine with a tile of the same value as itself. 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 as the 8 can move into the 8 space to combine to form a 16.

The logic behind this method is that you are searching for any tile on the board that you can move in the direction passed in. A tile can move in a certain direction if it passes one of the above requirements stated at the beginning of this section. Once you find a tile that can move, you can return true. Otherwise, you must keep searching. If you never find a tile that you can move, then return false.

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. for example:

```
private boolean canMoveLeft();
```

You can then call this method if Direction.LEFT was passed into canMove()

Board.java: public boolean isGameOver()

This is a short method that will check to see if the game is over. The game is over once there are no longer any valid moves left. Once you determine that the game is over, print out the words "Game Over!" on a newline, and return true. If you determine that the game is not over, then just return false.