Com S 227
Spring 2021
Assignment 3
300 points

Due Date: Tuesday, April 6, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm April 5)

10% penalty for submitting 1 day late (by 11:59 pm April 7)

No submissions accepted after March 3, 11:59 pm

(Remember that Exam 2 is THURSDAY, April 8.)

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Note: Our second exam is Thursday, April 8, which is just two days after the due date for this assignment. ***It will not be possible to have the assignments graded and returned to you prior to the exam****.*

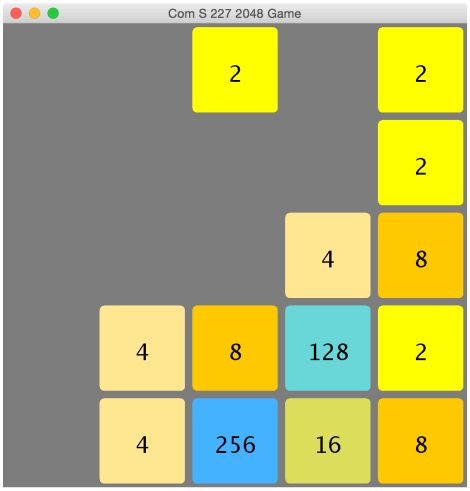
Please start the assignment as soon as possible and get your questions answered right away!

Introduction

The purpose of this assignment is to give you some practice writing loops, using arrays and lists, and, most importantly, to get some experience putting together a working application involving several interacting Java classes.

There are two classes for you to implement: `Powers` and `ShiftUtil`. As always, your primary responsibility is to implement these classes according to the specification and test them carefully.

These two classes can be used, along with some other components, to create an implementation of our version of a game known as "2048." If you are not familiar with the game, do not worry, it is not complicated. (It is, however, highly addictive, so watch out.)



Score: 2576

The game consists of a grid of moveable *tiles*. Each tile contains a number that is a power of 2. We will think of the grid as a 2D array of integer values, where the value 0 means there is no tile in that cell. There are only four possible operations: to "shift" the grid up, down, left, or right. What this means is that the tiles are *shifted* in the indicated direction, and that neighboring tiles with the same value are *merged*. For example, referring to the screenshot above, after shifting the grid upwards, it might look like this:

Score: 2588

Note that there is also a new tile on the left side. Whenever any tiles are moved, a new tile is generated in a random empty position, having either value 2 or 4.

The object of the game is to end up with a tile that has value 2048.

There are many versions online you can try out to get an idea of how it works; for example, as of the moment this document is being written, there is one you can play at http://www.coolmath-games.com/0-2048. *Please note that this and other versions you find may not correspond precisely to the game as specified here.*

The two classes you implement will provide the "backend" or core logic for the game. In the interest of having some fun with it, we will provide you with code for a GUI (graphical user interface), based on the Java Swing libraries, as well as a simple text-based user interface. More details below.

The sample code includes a documented skeleton for the two classes you are to create in the package **`hw3`**. The additional code is in the packages **`ui`** and **`api`**. The **`ui`** package is the code for the GUI, described in more detail in a later section. The **`api`** package contains some relatively boring types for representing data in the game. There are a few examples of test cases for you to start with located in the default package.

You should not modify the code in the **`api`** package.

Specification

The complete specification for this assignment includes

- this pdf,
- the online Javadoc, and
- any "official" clarifications posted on Canvas and/or Piazza

Overview of the ShiftUtil class

The class **`ShiftUtil`** consists of a collection of methods that implement the basic logic of shifting a single row or column. The idea is to carefully isolate the trickiest parts of the algorithm so that it can be developed and tested independently. This is a "utility" class and it should have **no instance variables**. Logically all the methods could have been specified as static methods; however we have chosen not to do so in order to facilitate testing.

In particular, **`ShiftUtil`** only operates on a one-dimensional array and only shifts to the left. We will later see how these simpler operations can be used easily by the **`Powers`** class to shift in any direction.

What does it mean to "shift" an array in this context? We will refer to an array cell containing a zero value as "empty" and a cell with a nonzero value as "nonempty." To shift the array basically means to a) *shift* values to the left so that all nonempty cells are to the left of empty ones, and b) *merge* two neighboring nonempty cells when they have the same value, giving the resulting cell twice the old value. The details are given more precisely in the javadoc for the **`ShiftUtil`** class, which describes the required algorithm in pseudocode. It is important to note that in **`ShiftUtil`** we work only from the left: if there are three neighboring cells with the same value, only the first two are merged. Also, merges do not cascade; a new merged value cannot be merged again during the same operation.

Here are some examples of performing the **`shiftAll`** operation:

- for [2, 0, 0, 4, 0, 2], result is [2, 4, 2, 0, 0, 0]
- for [2, 0, 2, 4, 0, 4], result is [4, 8, 0, 0, 0, 0]
- for [0, 2, 2, 2, 0, 0], result is [4, 2, 0, 0, 0, 0]
- for [2, 2, 2, 2, 2, 0], result is [4, 4, 2, 0, 0, 0]

The **`shiftAll`** operation is further broken down into smaller steps that can be implemented and tested incrementally. The basic step is to identify a single shift or merge operation that places a new value in a given cell of the array. A shift could be a simple move of a value from one cell to another, or it could be a merge of two cells. After identifying a shift, the array has to be updated before identifying and performing additional shifts.

For example, consider the array [2, 0, 0, 4, 0, 2].

1. Start with index 0, which has a 2. What value should go there? We find the next nonempty cell to its right, which is a 4. It can't be merged with the 2, so there is nothing to do, the 2 just stays at index 0.

2. Next, go to index 1. This cell is "empty" (contains a zero). What value should go there? We find the next nonempty cell to its right, which is the 4 at index 3. However, since we started with an empty cell, it is possible that there is another 4 to the right that should be merged. So we find the next nonempty cell to the right of index 3, which is a 2 at index 5. The 2 cannot be merged with the 4. Therefore, we just move the 4 at index 3, down to index 1. We now have [2, 4, 0, 0, 0, 2].

3. Next, go to index 2. This cell is empty. What value should go there? We find the next nonempty cell to its right, which is the 2 at index 5. Again, look for a nonempty cell to the right of index 5, to see whether they should be merged. There is no nonempty cell to the right of index 5. Therefore, we move the 2 at index 5 down to index 2. We now have [2, 4, 2, 0, 0, 0].

4. Next, go to index 3. This cell is empty, and there is no nonempty cell to its right, so do nothing. The same happens for indices 4 and 5. (Clearly the loop could be short-circuited once we discover that there are no nonempty cells on the right.)

As an example containing some merges, consider [2, 0, 2, 4, 0, 4]

1. Start with index 0, which has a 2. What value should go there? We find the next nonempty cell to its right, which is the 2 at index 2. These cells have the same value, so we *merge* indices 0 and 2 into index 0, obtaining [4, 0, 0, 4, 0, 4].

2. Next, go to index 1. This cell is empty. What value should go there? We find the next nonempty cell to its right, which is the 4 at index 3. Since we are starting with an empty cell, we also find the next nonempty cell to the right of index 3, which is the 4 at index 5. These cells have the same value, so we merge indices 3 and 5 into index 1, obtaining [4, 8, 0, 0, 0, 0].

3. Next, go to index 2. This cell is empty, and there is no nonempty cell to its right, so do nothing. The same thing happens for indices 3, 4, and 5.

We describe each shift/merge with an instance of the **`Shift`** class, which contains (among other attributes) the old index of the cell to be moved, possibly an additional index for a second cell that is to be merged, and the new index (which in case of a merge, might be the same as the old index). The **`Shift`** class actually serves two purposes. The first is to enable us to separate the logic for *finding* a move from the logic for *updating* the array. The algorithm for **`shiftAll`** is thus implemented in terms of the following basic operations of **`ShiftUtil`**:

`public int findNextNonempty(int[] arr, int start)`
- *returns the index of the next nonempty cell on or after* **`start`**

`public Shift findNextPotentialShift(int[] arr, int index)`
- *returns the shift to perform in order to obtain the correct value at* **`index`**, *without modifying the array*

`public void applyOneShift(int[] arr, Shift shift)`
- *modifies the array by performing the given shift*

See the javadoc for **`ShiftUtil`** for more details and pseudocode for the **`findNextPotentialShift`** method. The logic of **`shiftAll`** is simple to describe in terms of the basic operations above:

for each index i in the array
    find the next potential shift at position i
    if there is a potential shift
        apply the shift to the array
        add the shift to the result list
return the list of shifts performed

The second purpose of the **`Shift`** class is to support the development of a user interface. In order to have the potential of animating the movement of tiles in the game, we need to supply the UI with a description of the moves being made. More on this later.

Overview of the Powers class

The **`Powers`** class encapsulates the state of the game. This includes an n x n grid representing the tiles, the current score, an instance of **`ShiftUtil`**, and an instance of **`Random`** for generating new tile positions and values. A zero value at a given row and column is interpreted as an "empty" cell. A likely implementation would use a 2D array of **`int`** for the grid, but there is no particular requirement to do so, as long as **`getTileValue`** returns the correct integer values.

Basic game play: the doMove() method

The basic play of the game takes place by calling the method

```
public void MoveResult doMove(Direction d)
```

The type **`Direction`** is just a set of constants for indicating which direction to collapse the grid:

```
Direction.LEFT
Direction.RIGHT
Direction.UP
Direction.DOWN
```

Instead of just using integers for these four values, **`Direction`** is defined as an **`enum`** type. You use these values just like integer constants, but because they are defined as their own type you can't accidentally put in an invalid value.

Tip: add the line

```
import static api.Direction.*;
```

to the top of your Powers file. Then you can refer to the four constants without having to type "**`Direction.`**" in front of them.

The **`MoveResult`** type contains a list of **`Descriptor`** objects plus a **`TilePosition`** object describing a new tile being added to the grid, if any. Each **`Descriptor`** object encapsulates one of the **`Shift`** objects returned from the **`shiftAll`** method of **`ShiftUtil`**, but it also includes a row or column index, and the **`Direction`** for the move. Since it contains a record of all changes to the grid, the **`MoveResult`** can be used by a user interface to produce an animation of the tiles.

Implementing doMove()

The **`doMove`** method will make use of the algorithms implemented in **`ShiftUtil`** to shift the entire grid in the indicated direction. However, the **`shiftAll`** method in **`ShiftUtil`** only collapses to the left. How do we do the other three directions? The simple solution is to define a method that copies a row or column from the grid into a temporary array. We can copy any row or column in either direction. The method is:

```
public int[] getRowOrColumn(int rowOrColumn, Direction dir)
```

For example, suppose we have the grid,

0	2	0	0
0	4	0	0
0	0	0	0
0	8	0	0

Then a call to **`getRowOrColumn(1, Direction.DOWN)`** would return the array [8, 0, 4, 2]. Note that the **`rowOrColumn`** argument is a *row* index for directions **`LEFT`** and **`RIGHT`**, but is a column index if the direction is **`UP`** or **`DOWN`**. There is a corresponding method

```
public void setRowOrColumn(int[] arr, int rowOrColumn, Direction dir)
```

that takes the given array and copies its elements into the grid, in the given direction.

With those two methods in mind, you can see how the basic logic of **`doMove`** might work:

for each row/column index
get the array for that row/column, in the given direction
call shiftAll on the array
set the array back into the grid in that row/column, in the given direction
create a descriptor for each shift object and add it to the MoveResult
if any shifts actually took place, generate a new tile too

Scoring

Whenever two cells are merged, the new (combined) value is added to the score.

Generating new tiles

Each time a **`doMove`** operation actually moves one or more cells, a new tile should appear in the grid. The purpose of the **`generateTile()`** method is to select a new position and value for the tile using the game's instance of **`Random`**. The requirement is that

- all empty positions are equally likely, and
- the new tile's value is 2 with 90% probability and 4 with 10% probability

The row, column, and value are returned as an instance of the simple class **`TilePosition`**.

The text-based UI

The **`util`** package includes the class **`ConsoleUI`**, a text-based user interface for the game. It has a **`main`** method and you can run it after you get the required classes implemented. The code is not complex and you should be able to read it without any trouble. It is provided for you to illustrate how the classes you are implementing might be used to create a complete application. Although this user interface is clunky, it has the advantage that it is easy to read and understand how it is calling the methods of your code. It does not use the the **`MoveResult`** object returned by the **`doMove()`** method, so you can try it out before you have that part working.

The GUI

There is also a graphical UI in the **`ui`** package. The GUI is built on the Java Swing libraries. This code is complex and specialized, and is somewhat outside the scope of the course. You are not expected to be able to read and understand it. However, you might be interested in exploring

how it works at some point. In particular it is sometimes helpful to look at how the UI is calling the methods of the classes you are writing.

The controls are the four arrow keys.

The main method is in **`ui.GameMain`**. You can try running it, and you'll see the initial window, but until you start implementing the required classes you'll just get errors. All that the main class does is to initialize the components and start up the UI machinery. The class **`GamePanel`** contains most of the UI code and defines the "main" panel, and there is also a much simpler class **`ScorePanel`** that contains the display of the score.

You can configure the game by setting the first few constants in GameMain: to use a different size grid, to attempt to animate the movement of the tiles, or to turn the verbose console output on or off. Animation requires that the **`MoveResult`** object returned by the **`doMove()`** method be completely valid. You can still try out the UI with animation off.

The interesting part of any graphical UI is in the *callback* methods. These are the methods invoked when an event occurs, such as the user pressing a button. If you want to see what's going on, you might start by looking at **`MyKeyListener`**. (This is an "inner class" of **`GamePanel`**, a concept we have not seen yet, but it means it can access the **`GamePanel`**'s instance variables.)

If you are interested in learning more, there is a collection of simple Swing examples linked on Steve's website. See http://www.cs.iastate.edu/~smkautz/ and look under "Other Stuff". The absolute best comprehensive reference on Swing is the official tutorial from Oracle, http://docs.oracle.com/javase/tutorial/uiswing/TOC.html . A large proportion of other Swing tutorials found online are out-of-date and often wrong.

Testing and the SpecChecker

As always, you should try to work incrementally and write tests for your code as you develop it.

Do not rely on the UI code for testing! Trying to test your code using a UI is very slow, unreliable, and generally frustrating. The code for the UI itself is ten times more complex than the code you are implementing, and it is not guaranteed to be free of bugs***. In particular, when we grade your work we are NOT going to run either of UIs, we are going to test that each method works according to its specification.***
Note also that when we test your code, we are going to test your ShiftUtil separately and use a known, valid implementation of ShiftUtil when testing your Powers class.

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specifications, ask questions when things require clarification, and write your own unit tests.

Since the test code is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

This SpecChecker will also offer to create a zip file for you that will package up the two required classes. Remember that your instance variables should always be declared **`private`**, and if you want to add any additional "helper" methods that are not specified, they must be declared **`private`** as well.

Importing the sample code

The sample code includes a complete skeleton of the two classes you are writing. It is distributed as a complete Eclipse project that you can import. It should compile without errors out of the box. *However, neither of the UIs will not run correctly until you have implemented the basic functionality of* **`Powers`**.

1. Download the zip file. You don't need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for "Select archive file".
4. Browse to the zip file you downloaded and click Finish.

If you have an older version of Java or if for some reason you have problems with this process, or if the project does not build correctly, you can construct the project manually as follows:

1. Extract the zip file containing the sample code (*Note: on Windows, you have to right click and select "Extract All" – it will not work to just double-click it*)
2. In Windows Explorer or Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse
4. In the Eclipse Package Explorer, navigate to the src folder of the new project.
5. Drag the **`hw2`**, **`ui`**, and **`api`** folders from Explorer/Finder into the **`src`** folder in Eclipse.

Getting started

At this point we expect that you basically know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification.

You can find the example test cases below in the files **`SimpleTestsUtil.java`** and **`SimpleTestsGame.java`**, located in the default package of the sample code. *Don't try to copy/paste from the pdf.*

Please remember that these are just examples to get you started and you will need to write many, many more tests to be sure your code is working correctly. (You can expect that the functional tests we run when grading your work will have well over 100 test cases, for example.)

Don't rely on the GUI for testing your code. Doing so is difficult and frustrating because the GUI itself is so complex. Rely on simple, focused test cases that you can easily run in the debugger or pythontutor.

You can start with either **`Powers`** (see step #7) or **`ShiftUtil`**, but of course the fundamental **`doMove`** method is dependent on **`ShiftUtil`**, so you won't be able to get it all working until your **`ShiftUtil`** is working. Be sure to familiarize yourself with the simple types in the **`api`** package, which you will need to be able to use.

1. Let's start (for no particular reason) with **`ShiftUtil`**. As discussed in the "Overview of the **`ShiftUtil`** class" section above, **`ShiftUtil`** is designed as a set of steps toward implementing the fundamental **`shiftAll`** algorithm in an incremental, testable way. Read that section carefully before you begin, and also read the javadoc for **`ShiftUtil`**, which includes some pseudocode to further specify the algorithm. *Remember, this is a "utility" class and should have no instance variables. Every method can compute its result based only on the given arguments.*

You might first be sure you can find the index of the next nonempty cell:

```
int[] arr = {2, 0, 0, 4, 0, 2};
int index = util.findNextNonempty(arr, 0);
System.out.println(index);
System.out.println("Expected 0");
index = util.findNextNonempty(arr, 1);
System.out.println(index);
System.out.println("Expected 3");
```

2. Then, think about applying shifts to an array. You can check your work by constructing some **`Shift`** objects to use. For example:

```
int[] test = {2, 0, 0, 4, 0, 2};
int[] expected = {2, 4, 0, 0, 0, 2};

// move index 3 to index 1, tile value is 4
Shift shift = new Shift(3, 1, 4);
util.applyOneShift(test, shift);
System.out.println("Result:   " + Arrays.toString(test));
System.out.println("Expected: " + Arrays.toString(expected));
System.out.println();

test = new int[]{2, 0, 2, 4, 0, 4};
expected = new int[]{2, 0, 2, 8, 0, 0};

// merge index 3 and 5 to index 3, (current) tile value is 4
shift = new Shift(3, 5, 3, 4);
util.applyOneShift(test, shift);
System.out.println("Result:   " + Arrays.toString(test));
System.out.println("Expected: " + Arrays.toString(expected));
```

3. The key algorithm is in **`findNextPotentialShift`**. Work through a few examples with pencil and paper, using the pseudocode in the Javadoc. Notice that since the **`Shift`** class has a **`toString`** method, we can just pass a **`Shift`** object to **`System.out.println`** and it will automatically invoke the **`toString`** method to give us some readable output.

```
// using arr = {2, 0, 0, 4, 0, 2};
Shift move = util.findNextPotentialShift(arr, 0);
System.out.println(move);
System.out.println("Expected null");// no possible shift to index 0

move = util.findNextPotentialShift(arr, 1);
// should be: move index 3 to index 1, tile value 4
Shift expectedShift = new Shift(3, 1, 4);
System.out.println("Result:   " + move);
System.out.println("Expected: " + expectedShift
```

4. Then see if we can identify a potential shift involving a merge.

```
arr = new int[]{2, 0, 2, 4, 0, 4};
shift = util.findNextPotentialShift(arr, 3);
// should be: merge index 3 and 5 to index 3, tile value 4
expectedShift = new Shift(3, 5, 3, 4);
System.out.println("Result:   " + shift);
System.out.println("Expected: " + expectedShift);
System.out.println();
```

5. Finally, put together the **`shiftAll`** method using the pieces above:

```
test = new int[]{2, 0, 0, 4, 0, 2};
expected = new int[]{2, 4, 2, 0, 0, 0};
util.shiftAll(test);
System.out.println("Result:   " + Arrays.toString(test));
System.out.println("Expected: " + Arrays.toString(expected));
System.out.println();

test = new int[]{2, 0, 2, 4, 0, 4};
expected = new int[]{4, 8, 0, 0, 0, 0};
util.shiftAll(test);
System.out.println("Result:   " + Arrays.toString(test));
System.out.println("Expected: " + Arrays.toString(expected));
System.out.println();
```

6. In the tests above, notice we are ignoring the **`ArrayList<Shift>`** that is supposed to be returned by **`shiftAll`**. Initially, you could just return null, and the tests above should work. In fact, the implementation of **`doMove`** can be mostly completed without that list. The list of **`Shift`** objects is only used in **`doMove`** to construct the **`MoveResult`** object that it returns.

When you are ready to check the return value of **`shiftAll`**, you could try something like this:

```
test = new int[]{4, 0, 2, 4, 0, 4};
expected = new int[]{4, 2, 8, 0, 0, 0};
ArrayList<Shift> shifts = util.shiftAll(test);
System.out.println("Result:   " + Arrays.toString(test));
System.out.println("Expected: " + Arrays.toString(expected));
System.out.println();
System.out.println("Return value: ");
if (shifts != null)
{
  for (Shift s : shifts)
  {
    System.out.println(s);
  }
}
```

The expected output is something like this, where the results can be in any order. (Since the **`Shift`** class has its own **`toString`** method, we can just pass a **`Shift`** directly to System.out.println.):

```
Move 2 to 1
Merge 3 and 5 to 2
```

7. When you move on to the **`Powers`** class, we are back in the world of object-oriented programming. You can start by defining instance variables for the key aspects of the game state: the grid (a 2-dimensional int array, most likely), an instance of **`ShiftUtil`**, a **`Random`** object, and the score, for example. Make sure that the easy methods such as **`getTileValue`** and **`setValue`** work correctly. The constructor is also supposed to place two values randomly in the grid, which you would presumably do with the **`generateTile`** method (see item 12), but you can skip that for now if you want.

```
ShiftUtil util = new ShiftUtil();
Powers g = new Powers(4, util);
System.out.println(g.getSize());  // expected 4
g.setValue(2, 3, 42);  // sets a value at row 2, column 3
System.out.println(g.getTileValue(2, 3)); // expected 42
```

8. Then, one of your first problems will be: how do I test the methods **`getRowOrColumn`** or **`setRowOrColumn`**? Here is one possible approach to get started. We can use the static method **`printGrid`** from the **`ConsoleUI`** to easily examine the grid. Here is an example:

```
ShiftUtil util = new ShiftUtil();
Powers g = new Powers(4, util);

int[][] test =
  {
    {2, 2, 0, 2},
    {0, 4, 4, 0},
    {0, 4, 0, 0},
    {0, 8, 0, 2}
  };

// copy the test array into the game grid
for (int row = 0; row < test.length; row += 1)
{
  for (int col = 0; col < test[0].length; col += 1)
  {
    g.setValue(row, col, test[row][col]);
  }
}

int[] result = g.getRowOrColumn(1, Direction.DOWN);
int[] expected = new int[] {8, 4, 4, 2};
System.out.println("Result:   " + Arrays.toString(result));
System.out.println("Expected: " + Arrays.toString(expected));
```

9. Then, to test **`setRowOrColumn`**, just do the opposite, for example:

```
g = new Powers(5, util);
int[] arr = {1, 2, 3, 4, 5};
g.setRowOrColumn(arr, 3, Direction.LEFT);
ConsoleUI.printGrid(g);  // verify
int[][] expected2d =  {
  {0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0},
  {1, 2, 3, 4, 5},
  {0, 0, 0, 0, 0},
};
ConsoleUI.print2dArray(expected2d);
```

(Note that, if you have fully implemented the constructor, you may also see one or two nonzero cells that were initialized by your constructor and placed randomly in the grid.)

10. Once you have implemented **`ShiftUtil`**, you can try out the **`doMove`** method. Notice that you can do this incrementally. For example, to start out, you can ignore the list of **`Shift`** objects returned from **`shiftAll`** method of **`ShiftUtil`**. You'll eventually need them to construct the **`MoveResult`** object, but at the beginning, you can just return null. The updates to the grid can still be done correctly, and you can actually play the game in the console UI, which ignores the **`MoveResult`**s anyway. Try some test cases like this:

```
g = new Powers(4, util);
int[][] test2 =
  {
    {8, 8, 0, 8},
    {0, 4, 4, 0},
    {0, 32, 0, 0},
    {0, 16, 0, 8}
  };
for (int row = 0; row < test2.length; row += 1)
{
  for (int col = 0; col < test2[0].length; col += 1)
  {
    g.setValue(row, col, test2[row][col]);
  }
}

MoveResult result = g.doMove(Direction.RIGHT);
int[][] expectedAfterMove =
  {
    {0, 0, 8, 16},
    {0, 0, 0, 8},
    {0, 0, 0, 32},
    {0, 0, 16, 8}
  };
```

```
System.out.println("Result: ");
ConsoleUI.printGrid(g);
System.out.println("Expected: ");
ConsoleUI.print2dArray(expectedAfterMove);
```

(Remember that your output might contain an extra 2 or 4 that was randomly placed by generating a new tile, if you have implemented that part yet.)

11. To verify the `MoveResult`, just iterate over the `Descriptors` and look at them.

```
ArrayList<Descriptor> descriptors = result.getMoves();
for (Descriptor d : descriptors)
{
  System.out.println(d);
}
System.out.println("New tile: " + result.getNewTile());
```

The output should be something like this (the descriptors can be in any order, and of course the new tile position is random):

```
Merge 0 and 2 to 0 (row 0 RIGHT)
Move 3 to 1 (row 0 RIGHT)
Merge 1 and 2 to 0 (row 1 RIGHT)
Move 2 to 0 (row 2 RIGHT)
Move 2 to 1 (row 3 RIGHT)
New tile: Position (2, 1) value 2
```

12. At some point, you'll need to implement `generateTile`. There are easy ways and hard ways to do it. Fundamentally you need to count the empty cells of the grid and select one at random. One strategy would be to create a one-dimensional array of `TilePosition` representing the empty cells in the grid, and select one of those.

Style and documentation

Roughly 10% of the points will be for documentation and code style. The general guidelines are the same as in homework 2. However, since the skeleton code has the public methods fully documented, there is not quite as much to do. Remember the following:

- You must add an @author tag with your name to the javadoc at the top of each of the classes you write.
- You must javadoc each instance variable and helper method that you add. Anything you add must be `private`.
- Since the code includes some potentially tricky loops to write, **you ARE expected to add internal (//-style) comments, where appropriate**, to explain what you are doing inside the longer methods. A good rule of thumb is: if you had to think for a few minutes to

figure out how to do something, you should probably include a comment explaining how it works. Internal comments always *precede* the code they describe and are indented to the same level.

- Keep your formatting clean and consistent.
- Avoid redundant instance variables
- Accessor methods should not modify instance variables

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`hw3`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`hw3`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

If you have a question that absolutely cannot be asked without showing part of your source code, change the visibility of the post to "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any announcements from the that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Assignments page on Canvas, before the submission link will be visible to you.

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw3.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw3`**, which in turn contains two files,

`Powers.java` and **`ShiftUtil.java`**. Please LOOK at the file you upload and make sure it is the right one!

Submit the zip file to Canvas using the Assignment 3 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", which can be found as link #9 on our Canvas home page.

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw3**, which in turn should contain the files **Powers.java** and **ShiftUtil.java**. You can accomplish this by zipping up the **src** directory of your project. **Do not zip up the entire project**. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and NOT a third-party installation of WinRAR, 7-zip, or Winzip.