# Project 4 Testing Preface

---

**Potential Extra Credit**

As indicated in the canvas comments, these are the things I added to my code to make it more appealing visually and experimentally:

- Reset and Quit Guestures
- Keyboard input for movements (base behavior exists still)
- Immutable board and text
- Title and Instructions sections in game
- Colorful background

---

This testing document was written by making use of the markdown editor *Obsidian*. This document will frequently reference my testing file *SlideGameTest.java*.

---

**My methodology**

I worked through all of these tests making use of the Test 0, Test 1, Test Many, and Test First, Test Middle, Test Last guidelines. As per the instrcutions of the project, I will not be including the actual code in this document.

---

I used Maven to run, compile, and test all of the files in this project. This is a project manager for java, and it should be noted that the code compiles and tests in DrJava, as per the guidelines of this course. If for some reason you get a compiler error, I almost guarantee it is nothing wrong with my code. If you are not using a project manager, I believe removing the package declarations at the top of files should do the trick. I think adding my code to a

package called 'projectFour' would also work. **I ask that you do not take of points for compilation errors, as everything compiles on my end and all tests run without issue.**

---

## I used the following imports in my testing file:

```
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Arrays;
import java.util.List;
import java.util.Map;

import javafx.application.Application.Parameters;

import org.junit.*;
import static org.junit.Assert.*;
```

The imports here are all necessary to prevent thrown exceptions and proper class overrides (Paramters specifically). I do not use Arrays, List, or Map outside of the Parameters nested class and its tests, and I do not bother implementing complex versions of the getUnamed or getNamed methods in the Parameters class that I was forced to override.

---

## Organization:

This projects organization differs from the previous one. While I wrote fewer testing methods, the real length of the file comes from the use of java refelction. I used javadoc to comment any helper methods or nested classes I used, though I did not provide documentation for the tests. I felt there was no reason to write them, as the Unit tests will be explained here instead. The helper methods do not have tests written for them, as they are tested proactively when used to test expecteed vs actual results. I was unable to test certain GUI methods, and I will list them in the following section.

- Also, at the start of many of my unit tests, i declare empty boards to prevent Null Pointer Exceptions.

*I tried my best to indicate which line tests were on, but that proved to be somewhat difficult as I made small changes to the format of the java file.*

---

### Reflection:

As I mentioned above I used Java reflection to test private methods. I will not be listing them in the testing document. They are repetitive and it would be a waste of my time. They are fully commented in the testing file, however.

---

### Fields:

Finally, I will not explain any fields in this file, as they are all Method declarations for Java reflection.

# Untested Methods

---

I will be using **(...)** to indicate methods with arguments.

- **getGameGrid() & setGameGrid(...)**: These methods are purely for gameGrid organization with respect to the game's GridPane, and are not tested as such.
- **getColor()**: Method is used only when declaring stage node's background color
- **start(...)**: For obvious reasons. The start method is used only to start the application and declare children of the Stage.
- **updateGameBoard(...)**: This method accesses the gameGrid directly, and declares button text to the corresponding board number, as long as it is non-zero. This is pure GUI manipulation, and is not tested.
- **reset()**: Very simply resets the current gameBoard as if the game hasn't started yet. Makes use of other methods, and has no real implementation itself that hasn't already been omitted or tested.
- **setButtonActions(...)**: Makes use of plenty of anonymous classes in order to set the clicking behavior for various buttons as required by the project.

- **handleKeyPress(...)**: Exclusively handles key presses in such a way that the Stage, Scene, and grid of buttons is manipulated based on different inputs. No testing is done as it works specifically with GUI components.
- All `MoveOnAction` methods are untested. This private nested class handles the logic more generally in order to apply it to various places in the code. It invokes plenty of other methods that are tested separately.
- **getMoveArea()**: This method is used exclusively for describing the rules and button presses to the user to the left of the board. It handles important formatting and text generation, which is directly linked to the GUI preventing easy testing.
- **buttonColor():** This method is used for changing color of the buttons. It was not tested with JUnit because there is no way to see if the CSS color I entered is valid without running the JavaFX application.
- All of the setter methods in `Logic` are not tested, as they are invoked with otehr tested methods.
- All `'Behavior'` methods other than **slideLeft() & slideRight()** are untested. While this may seem strange, closer looks into the merging methods in the `Logic` class explain why I made this choice. The remainder of the sliding method directly implement the movement methods of the `Logic` class, which all individually modify the current game board, and just need to be invoked to work.
  *Any other getter/setters or GUI specific methods that I failed to list here were not tested because of redundancy and lack of ability, respectively.*

# Actual Method Testing

---

JUnit tests will be labeled accordingly, and I will omit any private helper methods as they do not contribute to the actual testing document.

## boardDimTest()

---

*LINE 178*

- I first tested null. I designed my code to not accept null arrays through the board dimension setter, so it defaults to a 4 x 4 board, which can be observed in this first test.
- I then tested zero and one. These methods, when invoked with arguments less than 4, set the corresponding dimension to 4 by default. This can be observed in the Test 0 and Test 1

labeled assertions.

- I then tested two variations on many, with one being a square array and the other being a rectangle. I guaranteed the methods worked for these, as well as the getter methods, by using them in conjunction with the theoretically correct board dimensions.
- Finally, I tested dimensions that were incomplete, with only 1 dimension. I designed the code to bypass these types of inputs, setting the game's board to the default 4x4.
  *It should be noted that I went more in depth with related tests in the parseArgs method, as they accomplish similar things and reference similar fields.*

# copyBoardTest()

---

*LINE 195*

- First I tested null, which returns null by design to avoid null pointer exceptions.
- Then I tested a 1 x 1 board of zeros using the **zeroBoard(...)** helper method in the testing class. The boards should be equal, as the name of this method suggests.
- Then I tested two larger arrays, one that was a square and one that was a rectangle. This inadvertently tests the ascendBoard method, guaranteeing that this type of board works when copied, and that non-square arrays also work.

# parseArgsTest()

---

*LINE 230*

- I tested arguments that are non-numeric. This should set the dimensions to the default 4 x 4 as dimensions cannot be letters, obviously.
- I tested one non-numeric argument to if the method works if the user misinputs either dimension. In either position, the invalid dimension is set to 4, by design.
- I tested unreasonable dimensions where the values are numeric but are less than the minimum length. With negative arguments, the board's dimensions are set to the default.
- I tested 0 for both dimensions, which should also result in the default because a 0 x 0 array is technically null, but that has no place in the game if the user messes up.
- I then tested zero in either position once again in order to fully fool-proof the method. These tests ensures that the user cannot enter 0 in either dimension position.
- I then tested a series of reasonable arrays, labeled test default and test many (up to exceeding maximum). All of these arrays worked exactly as intended, setting the game board to the

inputted values.

- I tested values exceeding my maximum in three different ways. Two had the value in one spot only, and the only had exceeding values in both slots. I designed the method this way because I did not want to have the board lag to much as the methods scale poorly with size. These inputs all set the boards exceeding dimensions to 11, which I set to be the case.
- Finally I tested unformatted inputs, one with too many arguments and the other with too few. In the case with too many arguments, the board only accounted for the first two values, which was exactly what I wanted. For those with one argument, it is considered for the first slot and then the second slot is set to the default. these were both observed by these tests.
  *As you can see, the purpose of parseArgs is to prevent user error entirely.*

# gameBoardTest()

---

*LINE 327*

- I first tested a null array for dimensions. When the empty board method is invoked with null dimensions, it returns null fittingly.
- I tested dimensions of { 0, 0} to show that the same behavior is exhibited. 0 for dimensions means nothing, so the board returned by the game board maker is also null here.
- I then tested the smallest possible dimensions of 1 x 1. The board created should just have a nested array with 0 in it, though it is unusable in any actual play. This was confirmed in this test.
- I then tested two actual possible sizes to guarantee that the empty board creator works when properly invoked. I chose to omit further exhaustive testing as this behavior is confirmed by the board dimension setters. The proper behaviors were exhibited here, as expected. Empty boards of specified length were all properly created.

# setPositionTest()

---

*LINE 377*

- First I tested an index out of bounds of the array. I cannot have exceptions being thrown all the time in my code, so I set my method to not alter anything and simply return the inputted array. This is why the first test works successfully, and why I implemented it in the first place.
- I then tested 0, which here was setting an already 0 spot to 0, changing nothing and returning the inputted array.

- Then I tested 1 by changing one spot to something different than one thing given. I manually changed the position and then invoked the method. The position in the testing board changed and matched that returned by the method invoked.
- Finally, I changed many locations in the testing array. For each place in the array I changed, I invoked the **setPosition** method on that same index with the same replacement value. Comparing the two boards yielded a true result, as expected.

# randomIndexTest()

---

*LINE 417*

- First I tested a board with one open slot in it. I then repeatedly ran the **randomIndex()** method in order to ensure that the index returned was always the empty index that I determined at the start of the test.
- Then I tested a board with many zeros in it, but not all. To ensure that the random method grabbed random slots in the board that are non-zero, I ran the method 100 times and compared the value in the board evaluated at the index to 0. Each iteration was successful.
- Then, I tested a board with mostly non-zero numbers. I then repeated the same looping methodology on this board, and confirmed that each random index provided was indeed zero.
- Then I did the same thing with a board full of zeros. Each possible spot is 0, so all of the loop iterations should return 0 when evaluated the way described above.
- Finally, I tested a board with no zeros. When this is the case, an exception is thrown, which is confirmed by the Unit test not yelling at me when it is the last method called in the test.

# flipHorizontalTest()

---

*LINE 507*

- First I tested an empty board that is 1 x 1 in length. Flipping a 1 x 1 array simply returns that array by design. Invoking the horizontal flip here changed nothing and the result was equivalent to the input.
- Then I tested an ascending board (when I refer to this I mean each position counts up from 1) of 2 x 2 size. Performing a horizontal flip on this array places the first nested array in place of the second nested array, and vice versa. This is confirmed by retrieving the game board after invoking the method and asserting their equality.

- Then I tested a model 5 x 5 board in ascending order. I went through and wrote out the expected array, as seen in the testing document. The manual array and that returned by invoking the method were equal, indicating that the method worked.
- Then I tested the 2 different rectangular arrays. After following the exact same process as that described above, I was able to confirm that the method works for not only square arrays, but also rectangular arrays.

# flipVerticalTest()

---

*LINE 598*

- I tested the exact same arrays as seen in the **flipHorizontalTest** method. This will be a common theme for the rest of this testing document. I manually wrote out the expected arrays, and compared them with that returned by the invoked methods. This indicated successful code for the method.
- Finally, I tested an even larger array and followed the exact same process as seen above. This larger array matched that returned by the invocation.

# transposeTest()

---

*LINE 704*

- I tested the exact same arrays as seen in **flipVerticalTest**. I went through and manually transposed the game boards, setting each column to the respective row and each row to the respective column. When comparing these expected results with that returned by the invocation of this method, the boards were equivalent which was clearly a success.
  *This test was a great use of the **assertBoardEquals** helper method I created to test game boards. Comparing both the length and element contents allowed testing rectangular arrays to go very smoothly.*

# mergeRowLeftTest()

---

*LINE 874*

- First I tested a null row. The row returned should also be null because you cannot perform operations on a null object without throwing an unwanted exception.
- Then I tested an empty row which yields the same result as above for the exact same reason.
- Then I tested a row with one element. Shifting a row with no empty positions results in the same row, as confirmed by the test.
- Then I tested an actual row with 4 elements and no empty spaces. Performing the merge on this row results in the exact same row as inputted, which is supported and confirmed by the test.
- Then I tested a row of all zeros. The same thing should happen here as in the above test, which is that the returned row is identical to the input. This is supported by the testing class.
- Then I tested a row with an empty space at the first position. This caused all of the rows in the row to the left without combining any elements, which is confirmed by the test.
- Then I tested a row with an empty space in a middle position. The same result is expected as above since no elements can be combined.
- Then I tested a row with an empty space at the end, which results in the exact same row for the same reason.
- I then tested 3 different rows in a very similar format to those above. The only difference is that the rows had only one number in them and that is the default number 1. The rows follow the same expected formula as above, and the method worked properly.
- Then I tested a row with two combinable elements in it in the first positions. Merging this row should result in a 2 on the far left and the rest of the row as zeros since the array only has two 1s which are combinable.
- Then I repeated the test above but placed the two elements in the middle of the array. I also varied this test by running it again with a space between the middle elements, showing that the merge method works properly when zeros are 'in the way' of the merge.
- I then tested a row with the combinable elements in the last two spots, which showed that the method can merge elements in the last column, which I originally struggled with implementing.
- Then I tested the two big examples from the instructions. These examples are great for ensuring that the code works as intended by the instructor, which it does as I copy and pasted the result from the given.

# mergeRowRightTest()

---

*LINE 1020*

- Every single invoked test was the same here. I only changed the invocation method name and the expected result. Most of the expected results are simply reversed, except for the last few where I went through and thought out the results based on each element moving to the right as far as it can individually.

# mergeUpTest()

---

*LINE 1169*

- First I tested null and empty (0 x 0 ) boards. In both of these cases, the expected board is null because merging an empty board does nothing to the board itself.
- Then I tested a 1 x 1 board. Since the board has only one element in it, the result is the same as the input since nothing can change.
- Then I tested a 2 x 2 array of zeros. Since zeros fill the board, the shift 'occurs', but nothing changes because the merging algorithm effectively omits zeros.
- Then I tested a column board with no empty spots. Just like the above array, nothing can change because merging up on a non-zero board does nothing.
- Then I tested three different column scenarios with the 0 filled in three different spots, first middle and last. In all 3 cases, the expected column is just all of the elements push up to the top since no merging can happen.
- Then I tested many different variations of large arrays. In the first cases, I considered boards where there are the zeros in similar spots to the previous test. Then I went through and manually combined any possible pairs and compared my manual result to the outputted result from the method invocation. All of these were equivalent, indicating that my method works correctly.
- Finally, I tested variant of example one and two. These variants exist for the same reasons they do in **mergeRowLeftTest**, which is to ensure that my methods align with the given examples from the professor.

# mergeDownTest()

---

*LINE 1381*

- All test boards here are the same as they are in **mergeUpTest**. The only difference is that of the expected boards. I went through the mergeable and shiftable boards and compared them to that returned by the invoked method. I could do this because this method in the actual

game doesn't implement anything different that an upwards merge. the expected boards match those outputted by the method, indicating that my way of transforming the board to merge elements works.

# diagonalExtractorTest()

---

*LINE 1585*

- First I tested an array of null length. In this case the diagonals of a null array must be null, so null is returned as confirmed by the assertion.
- Then I tested a 0 x 0 array which results in the exact same result of a null since diagonals of a null array are also null.
- Then I tested a single value in an array. The diagonal in a single value array is just that value, which is confirmed by the test shown by Test an array of 1 length.
- Then I tested a square array with 2 rows and 2 columns full of zeros. There should be three rows in the resulting array, looking like a uniform normal distribution since the array is a square. This is shown in the testing class and matches up with the resulting array from invocation.
- Then I transitioned into multiple "test many" iterations. All of these tests cover the possibilities of (row = col), (row > col), and (row < col). This is curtail for confirming that the diagonalization works correctly, since the method works correctly and matches with my predictions regardless of dimensions. I used a combination of empty arrays and ascending arrays in order to make sure the method works for all elements and not just zero.

# diagonalReconstructorTest()

---

*LINE 1706*

- First I tested 5 possible null cases. In each case, an input is formatted or entered incorrectly, or its that the input is null itself. When any of these are the case, the result is always null since the method cannot reconstruct the array based on diagonals if not given all of the proper information. This is confirmed by all 5 tests.
- Then I tested an array with one element but using real yet improper dimensions. In this case, the input array should be returned without error, effectively bypassing the dimension check.
- Then I tested two more null cases when where proper full diagonal arrays are present but the dimensions are not present or not formatted correctly. This and the tests above test the

conditional branches in the method. All of these test passed, indicating correct conditional calling and return behavior.

- Then I tested many once again by using the results from the extractor. I simply reversed the position of the array (i. e. expected becomes actual, actual becomes expected). I matched the array with its corresponding full array and compare this expected array with the actual one outputted by the method invocation. Regardless of the type or length of array, the diagonals were perfectly reconstructed like nothing ever happened to them.
  *This is the final new behavior test. Everything else is extremely repetitive, and I will make plenty of references in the following explanations*

# mergeTopRightTest()

---

*LINE 1864*

- First I tested all the null, 0 length, and 1 length implementations since these should all be in the methods to prevent thrown exceptions.
- I implemented the exact same tests as I used in the testing method for **mergeUp**. When I did this, I was sure to manually shift the arrays such that they matched up with this given slide. Since this method uses transformations on the game board, these tests confirmed that my transformation order and thought process was correct. All of the tests passed perfectly, indicating that the method invokes the proper methods in the proper order while properly changing/preserving the game state when necessary.

# mergeBottomRightTest()

---

*LINE 2067*

- First I tested all the null, 0 length, and 1 length implementations since these should all be in the methods to prevent thrown exceptions.
- I implemented the exact same tests as I used in the testing method for **mergeUp**. When I did this, I was sure to manually shift the arrays such that they matched up with this given slide. Since this method uses transformations on the game board, these tests confirmed that my transformation order and thought process was correct. All of the tests passed perfectly, indicating that the method invokes the proper methods in the proper order while properly changing/preserving the game state when necessary.

# mergeTopLeftTest()

---

*LINE 2290*

- First I tested all the null, 0 length, and 1 length implementations since these should all be in the methods to prevent thrown exceptions.
- I implemented the exact same tests as I used in the testing method for **mergeUp**. When I did this, I was sure to manually shift the arrays such that they matched up with this given slide. Since this method uses transformations on the game board, these tests confirmed that my transformation order and thought process was correct. All of the tests passed perfectly, indicating that the method invokes the proper methods in the proper order while properly changing/preserving the game state when necessary.

# mergeBottomLeftTest()

---

*LINE 2513*

- First I tested empty and column only boards with the method invocations. In both of these cases, the boards are returned unmodified by the methods since there is no real modification to make.
- I implemented the exact same tests as I used in the testing method for **mergeUp**. When I did this, I was sure to manually shift the arrays such that they matched up with this given slide. Since this method uses transformations on the game board, these tests confirmed that my transformation order and thought process was correct. All of the tests passed perfectly, indicating that the method invokes the proper methods in the proper order while properly changing/preserving the game state when necessary.
  *In the diagonal testing methods, all column arrays are not changed since there is no way to move elements along diagonals that do not exist*

# slideLeftTest()

---

*LINE 2740*

- First I tested empty and column only boards with the method invocations. In both of these cases, the boards are returned unmodified by the methods since there is no real modification

to make.

- I implemented the exact same tests as I used in the testing method for **mergeUp**. When I did this, I was sure to manually shift the arrays such that they matched up with this given slide. Since this method uses transformations on the game board, these tests confirmed that my transformation order and thought process was correct. All of the tests passed perfectly, indicating that the method invokes the proper methods in the proper order while properly changing/preserving the game state when necessary.

# slideRightTest()

---

*LINE 2955*

- First I tested empty and column only boards with the method invocations. In both of these cases, the boards are returned unmodified by the methods since there is no real modification to make.
- I implemented the exact same tests as I used in the testing method for **mergeUp**. When I did this, I was sure to manually shift the arrays such that they matched up with this given slide. Since this method uses transformations on the game board, these tests confirmed that my transformation order and thought process was correct. All of the tests passed perfectly, indicating that the method invokes the proper methods in the proper order while properly changing/preserving the game state when necessary.
  *Please note that all of the recent sliding/merging tests were all basically copy and pasted and then I manually changed the result to accommodate the difference in invocation*

> ***That concludes the testing portion of this project***
>
> ```
> I appologize for funky formatting and length
> ```

Spell check does not work all the time in code blocks, so I apologize for minor errors or copy-paste issues, as some of the tests were very repetitive.