

Fixing

2048

How to Play 2048

- Consists of a 4x4 grid of tiles with numbers on them
- Use arrow keys to move tiles in a direction
- Tiles will keep moving until they hit a wall or another tile
- Every movement randomly spawns a new tile, value “2”, in an empty space
- Tiles of the same number that hit will combine and double the value
 - Ex. Moving an “8” tile up into another “8” tile results in a “16” tile
- The game ends in two cases:
 - GAME OVER: the tiles are stuck - no possible movements
 - VICTORY: when you combine enough tiles to get one valued “2048” (10 combinations)

512	128	32	8
64	8	4	
8	4		
2			

512	128	32	8
2	64	8	4
		8	4
			2

512	128	32	8
2	64	16	8
			2
			2



Provided Files that Are Fine

THESE FILES ARE ERROR FREE

- Start.java: The driver class that creates a new Board and begins playing
- Game.java: Interacts with the Board by inputting commands and printing it out so the user can see it
- Direction.java: Defines the directions that the Board tiles can move in
 - See next slide for an explanation of “Enum”
- BoardTest.java: The JUnit tests for testing Board.java
 - Do NOT remove test cases
 - Feel free to add your own

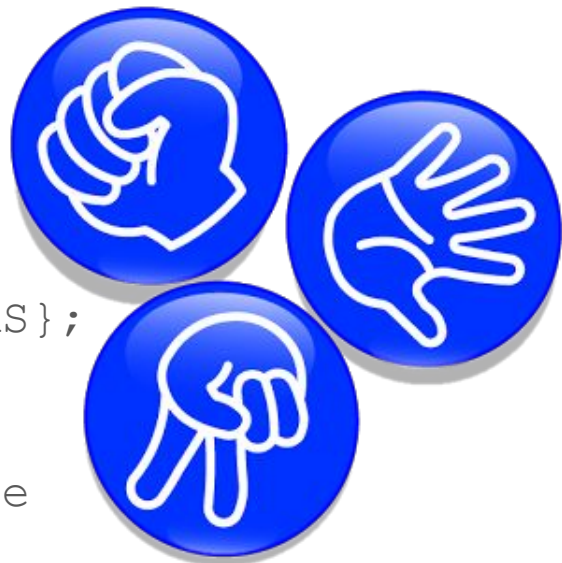
Enum Crash Course

- Similar to a boolean: can only be one of several values
 - Reminder: Booleans can only be true or false
- Enums can be the values you define it to have
- Example: ROCK, PAPER, SCISSORS

```
public enum playerHand {ROCK, PAPER, SCISSORS};
```

```
playerHand hand = playerHand.ROCK;  
if (hand == playerHand.ROCK) {...} // is true
```

```
switch(hand) { case ROCK: ...; case PAPER...; ...}
```



Running and Compiling Unit Tests

- To compile the UNIT TEST:

```
javac BoardTest.java
```

- To run the UNIT TEST:

```
java org.junit.runner.JUnitCore BoardTest
```

- To compile and run the GAME:

```
javac Start.java; java Start
```

JUnit Tests

- JUnit is a way to unit test a program
- Unit test: used in software development process in which the smallest testable parts of an application are individually and independently scrutinized for proper operation
- Unit tests allow you to automate testing
- Test a program method by method (methods are considered units)
- **Just because a program passes all its unit tests, that does NOT necessarily mean it is bug free**
 - There could be not enough test cases - the edge cases are not tested
 - The test cases themselves could be wrong

Make sure you are testing your code thoroughly throughout development!

assertEquals() is magic

- This method checks actual result vs expected result
 - For primitive types, assertEquals(a, b) checks if a == b
 - For objects, assertEquals(a, b) checks if a.equals(b)
-
- If test passes, nothing gets printed to terminal
 - If test fails, prints the test and both the expected and actual results

Example JUnit Program

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

public class ExampleTest {
    @Test
    public void testSomething() {
        int actual = SomeClass.someMethod();
        int expected = /* expected result of running method */
        assertEquals(actual, expected);
    }
}
```

Example JUnit Output

```
JUnit version 4.10
```

```
..E
```

```
Time: 0.01
```

```
There was 1 failure:
```

```
1) testFailed(ExampleTest)
```

```
java.lang.AssertionError: expected:<10> but was:<5>  
    at ...
```

```
FAILURES!!!
```

```
Tests run: 2, Failures: 1
```

- In the method `testFailed()` in `ExampleTest`, the expected value was 10, but the result was 5. Out of the two tests that were run, there was one failed test.

To Fix An Error

1. Find the specific test in the JUnit file
2. Figure out the input and the outputs, both expected and actual
3. Recreate the problem // sometimes not possible
4. Figure out which chunk of code is causing the problem
5. Fix the logic error in that chunk of code



Find the Test that Failed

- Given a failed test output, you can find the test that failed
- The test output will give you the expected and actual results

There were 14 failures:

```
1) testTransform(BoardTest)
```

```
arrays first differed at element [0]; expected:<4> but was:<2>  
    at ...
```

- So the test that failed was testTransform()
- But testTransform() has multiple assertEquals()
- **IF ONE FAILS, THE REST DO NOT RUN**
- Figure out which “assert” failed and fix it

Find the Expected and Actual Results

There were 14 failures:

```
1) testTransform(BoardTest)
```

```
arrays first differed at element [0]; expected:<4> but was:<2>  
    at ...
```

- Because this is a failed `assertArrayEquals()`, expected value at index 0 was 4, but result is 2
- Use this information to figure out exactly which `assertArrayEquals()` was the one that failed
- In `testTransform()`, the first and last `assertArrayEquals()` expect 4 at [0]
- So one of these two is failing (usually the first)
- HINT: In this case, the first “assert” fails so the rest do not run

Recreate the Problem

- Not always possible
- Helpful to visualize the bug, because JUnit can be vague
- Useful to see under what conditions a bug appears
- This is why it is important to have many tests that test specific things, instead of few tests that test multiple units

Figure Out Which Chunk of Code Is Wrong

- **HARD PART #1**
 - In this case look at `Board.transform()`
 - `Board.transform()` calls `Board.shift()`
 - Therefore, the bug could be in either `transform()` or `shift()`
-
- If a method does not call any other methods and it is wrong, the error must be in that method
 - If a method calls other methods and it is wrong, the error could be in either
 - **MAKE SURE** the basic methods are bug free before debugging methods that call them

Figure out the Logic Error

- **HARD PART #2**
- FORTUNATELY we already have unit tests that test `Board.shift()` aka `testShift()`
- We know those unit tests pass because no failure from `testShift()` is printed out
- Therefore, the bug must be in the logic of `Board.transform()` somewhere, because we know that `Board.shift()` is working fine
- Follow the logic of `Board.transform()`

Creating Your Own Test Cases

- MOST IMPORTANTLY make sure your expected value is correct
- Unit tests are no good if we check for the wrong value
- Ex. A unit test that insists `isOdd(3)` should return false

```
assertEquals(isOdd(3), false);
```

- We waste a lot of time trying to figure out what is wrong with `isOdd()`
 - The problem is actually in the test!
-
- Not required to make your own test cases
 - The game should work after all the fixes

Questions?

