Project 4: Tetris
Dhruva Karkada & Michael Yu

1 Overview and Structure
In the project we will implement a version of Tetris, adhering to specifications outlined by official Tetris rules. We implement both a TetrisPiece and TetrisBoard class that handle the logic of rotations, wall kicks, row clearing, and placing pieces, among other things. In addition, we implement a rudimentary Brain which can evaluate a Tetris board and determine the optimal next move. Our Brain can place between 100-200 pieces before failing. We hoped to improve our understanding of robust unit and integration tests and learn about the implementation details of making two different interfaces talk to each other in a simple manner.
2 Solution Design

2.1 TetrisPiece

We implemented the **Piece** interface with the **TetrisPiece** class. Each **TetrisPiece** is a doubly linked list of **TetrisPieces** that each refer to the next rotation of itself. Each **TetrisPiece** further has both a **center** and a **location**, which differ in that **center** refers to a relative local pivot point on the piece, while **location** refers to the absolute position on the board of the **center**, and therefore the entire piece. A **TetrisPiece** object is instantiated by passing in a string of integers representing points that make up the body of the piece. After this string is validated via regular expression, an array of **Points** is calculated and the **body** of the piece is created. We then use the body to do a one-time calculation of the **height** and **width** of the piece. We do the same for the other attributes that need to accessed in constant time, such as **skirt**. Afterwards, we create the circularly linked list.

We define a static list of template **Pieces** that define the correct “default” orientation that wall kicks and SRS are based on, according to the Tetris wiki. We start by rotating each piece until it matches one of the template strings. We then set the **TetrisType** of the Piece to its corresponding piece, and we also set its **rotationNum** to 0, which defines that it is in the default orientation. Our **rotationNums** go from 0 to 3, with each successive one defining another counterclockwise rotation. For example, 1 corresponds to L in SRS, and 3 to R. After we figure out what the piece is supposed to be, we hard code its **center** with a switch-case. Then, we simply rotate the piece three more times, each time recalculating the center. In this way, we get a linked list of **Pieces** that each have their own defined **center**, **rotationNum**, and all the other required attributes.

The **TetrisPiece** class also has its own helper methods, which are documented in the comments. One important one in particular is the **clone()** method, which creates a deep clone of the current piece. This is necessary for the **TetrisBrain** to function, since it needs to be able to create copies of the state of the board in order to test out different movement options without actually changing the original board.

2.2 TetrisBoard

The **TetrisBoard** class implements **Board**, and contains all the information about the state of the actual game. Most importantly, it contains a reference to the current **TetrisPiece** that is in play, and its absolute location. Using this information, it handles validating rotations and other movements, keeping itself in a valid state.

When we initialize a **TetrisBoard**, it checks to make sure that it will be greater than 4 blocks in both dimensions, because the board needs to be able to accommodate every orientation of the...
piece with the largest bounding box (the stick). It then initializes default fields, most importantly the **state** field which is a matrix of booleans and the actual reference to the state of the board.

The key method inside **TetrisBoard** is the **move()** method, which handles and executes commands. Given a command, **move()** will run through the switch case. For most actions (**Action.DOWN**, **Action.LEFT**, **Action.RIGHT**), the board will simply try moving the current piece in that direction and check if it is a valid move. However, in the **Action.DOWN** case, if moving the piece down by 1 will cause it to be in an invalid location, that means the piece has encountered a barrier and cannot move down anymore, and **place()** is called to set the place into the board.

Each time a piece is placed, the **TetrisBoard** will recalculate how filled each row and each column is, clearing rows if necessary. It also updates the corresponding instance fields so that everything can be read in constant time during gameplay.

The **Action.CLOCKWISE** and **Action.COUNTERCLOCKWISE** cases are a bit more tricky since they involve wall kicks. Inside **TetrisPiece**, we have another static array containing wall kick data, which are literally hard-coded points that the two cases loop through checking for a piece translation that results in a valid location. In the case that **Action.CLOCKWISE** or **Action.COUNTERCLOCKWISE** would result in an invalid piece location, we pull a list of **Points** to try from the **wallKickData** array based on the rotation of the current piece. Because of the way the data is defined, we only had to hard-code half of the cases due to symmetry. For example, a 0>R rotation has the same points to check as a R>0 rotation, except multiplied by -1.

Each time we test a point, we modify its location according to the wall kick data, then check if it is valid. If it is, we are done. If it is not, we reset the location and check the next point. If all are exhausted and nothing worked, then the piece is stuck and is placed.

Like the **TetrisPiece** class, **TetrisBoard** has its own **clone()** method that creates a deep clone of itself for the **Brain** to play around with different possible moves.

### 2.2 LessLameBrain2

The **LessLameBrain2** class is implemented similarly to the **LameBrain**, with the added functionality of considering rotations and weighing possibilities a little better. Just like the **LameBrain**, the **LessLameBrain2** also considers every dropping from x-value, but it also considers dropping each rotation from every possible x-value. This expands the number of possible moves by a factor of 4 (approximately).

An additional strength is the way it weighs different options. There are four main parameters that it looks at: height, number of new rows cleared, number of inaccessible holes created, and density of the structure.
1. The height is considered as a fraction of the total height, and as the current max height approaches the total height, the score suffers quadratically. This way, increasing the height at the bottom is not penalized as much as increasing the height at the top. The efficacy of this measurement is limited by the fact that we can’t know the “losing” height; we can only compare the current max height to the total possible height, not the height at which we lose.

2. The number of new rows cleared is weighted heavily and linearly.

3. The number of inaccessible holes is measured by seeing how many empty spaces are below placed blocks. Because this Brain implementation only uses DROP, there is no way for us to get to those blocks without clearing the rows above it. This number is weighted as a log function, because we wanted the initial holes to be weighted heavily, but subsequent holes to be counted that much.

4. The density of the structure is the percentage of occupied spaces in all the occupied rows (it doesn’t count rows that aren’t occupied). There is also a weight that favors high density in the bottom rows as compared to the top. This is equivalent to saying “try to move the piece to the lowest reasonably position possible.” The density is weighted quadratically.

And that’s it. It’s not a good brain, but our previous brain attempt LessLameBrain was too complex and we couldn’t debug it in time. More on that in the Reflection!

3 Testing

Upon first running SanityTests.java, we discovered that our implementation was not implemented in the same was as the sanity tests were expecting. For example, we originally did not have the absolute location of each TetrisPiece calculated at instantiation, and instead calculated it only once it was actually added to the board. In short, we ended up rewriting a lot of the code which somewhat (really) spaghettified it.

Then, we wrote our own tests. The tests were split into TetrisPiece unit tests, TetrisBoard unit tests, and one integration test that put everything together. Each test had final static fields of hard-coded TetrisPiece objects. We did not exhaustively test every combination of TetrisPiece and rotationNum, but did make sure to have variety in the pieces used in testing to give a greater sense of security that our code is actually kind of correct.

3.1 TetrisPiece Tests

Inside the TetrisPiece unit tests, we test the following methods:

```java
validatePiece()
equals()
bodyEquals()
rotate()
```
For `validatePiece()`, we came up with various problem strings that could be passed to the method, including negative numbers, not enough numbers, letters, and other unwanted inputs. We verified that the method returned `false` for these “bad” inputs. However, it should accept inputs that have extra whitespace either in between numbers or at the beginning/end of the string. These we verified that they returned `true`.

For the getters, we set the relevant property to some well-known value, then verified that the our expected value was the same value that the getter returned.

For `clone()`, it was critical that it actually produced a unique clone rather than a shallow copy. We tested by cloning a piece, then making modifications to the clone. We then verified that the original and the clone were indeed different (based on the `TetrisPiece equals()` method).

For `equals()`, we only care that two pieces have the same body, not that they are different memory references. We test this by creating pieces and comparing them to our static template pieces. Also, creating a piece with a body that doesn’t match any rotation of any template piece should result in a piece that also doesn’t equal any of the template pieces.

### 3.2 TetrisBoard Tests

`TetrisBoardTest.java` makes use of a base `testBoard` that is used to base the rest of the tests on. Inside the `TetrisBoard` unit tests, we test the following methods:

```java
Constructor
valid()
equals()
testMove()
getWidth()
getHeight()
dropHeight()
move()
getMaxHeight()
getRowsCleared()
getColumnHeight()
getRowWidth()
```
In general, tests that involved moving things and placing things in the actual board were done by cloning the base `testBoard` and then doing the manipulations. This was so we didn’t have to create extra instances of `TetrisBoards`. This also meant that our `clone()` method had to work 100%, since other tests relied on it.

We tested `clone()` in a similar manner to the way tested the `TetrisPiece clone()` method. We cloned boards, manipulated them, and checked that they were different from the original. Also, if we did the reverse of the manipulations (such as moving left, then moving right again), we checked that the clone was equal again to the original.

The constructor as mentioned previously requires an input of at least 4 by 4, or it will throw an exception. We tested by giving it bad arguments and catching the thrown exception.

Again much like in `TetrisPiece`, the getters were tested by setting the relevant value and reading the value, expecting it to be a well-known value.

To test `valid()`, we simply passed in points that we knew to either be inside or outside board and asserted the corresponding truth value. We tested negative numbers as well as positive.

Like in `TetrisPiece`, `TetrisBoards` should be equal as long as their states are equal. We created various boards of different sizes and containing different pieces, then verified that they were not equal to our base `testBoard`.

For wall kicks, we first moved the piece against the wall, then triggered the wall kick. We could determine by hand where the piece should move, then compared that result to where the piece actually ended up. We tested both stick and non-stick pieces.

### 3.3 Integration Tests

We based the integration test on the provided sanity tests. Our version runs through some basic initialization and progressive checks that the state of certain variables are ok before and after actions. We also tested that our implementation would clear multiple lines at once, and also that wall kicks occurred in an expected manner when everything was put together.

### 3.4 Brain Tests

We ran out of time for this one unfortunately.
4 Reflection

4.1 Assumptions and Limitations
This implementation of Tetris assumes only the 7 canonical pieces can be passed, allowing us to hard code values for the centers of each type of piece.

This implementation also generates many extra copies of basically everything, from TetrisPieces to TetrisBoards, especially when running the brain. This is not space efficient, but yielded a working solution. Given more times or fewer travelling days, we may have been able to come up with a way to minimize cloning.

4.2 Edge Cases
As mentioned previously, an important edge case is where the board is smaller than some of the bounding boxes of the pieces that could be input. This was addressed by creating checks inside of the TetrisBoard constructor.

Also, wall kicks could be considered a kind of edge case, but one that is very obvious and well documented.

Also, we didn’t Javadoc many things due to the very very rapidly approaching deadline, and we are very very sorry for the spaghetti code you may encounter.

4.4 Personal Reflection, Pair Programming
Again, we started off by brainstorming ideas and theory for what we wanted abstracted. As we each began to have a better idea of what we needed to do, we started to work alone on implementing different parts of the project, Dhruva on the TetrisPiece/TetrisBrain and Michael on wall kicks, testing, and random miscellaneous things that came up.

We attempted to implement an intelligent brain which would look at every possible path, but there were some super weird bugs that we couldn’t get rid of, or even figure out where they were. So with about 3 hours left to submit the project, we started from scratch and threw together some modifications to the original LameBrain and we’re submitting that.

4.5 Pair Programming Log

<p>| 9/30 - 10/2 | We are back in Houston, work a little independently |</p>
<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/3</td>
<td>Michael drives for 2 hours, Dhruva drives for 2 hours</td>
</tr>
<tr>
<td>10/4 - 10/5</td>
<td>Switch off driving every hour or so</td>
</tr>
<tr>
<td>10/6 - 10/9</td>
<td>Calhacks, calhacks recovery</td>
</tr>
<tr>
<td>10/10</td>
<td>Realize how screwed we are, start working independently</td>
</tr>
<tr>
<td>10/11</td>
<td>Some pair programming (2 hrs) Lots of indep (5 hours)</td>
</tr>
<tr>
<td>10/12 - 10/13</td>
<td>21 hours of independent programming oh yeah</td>
</tr>
</tbody>
</table>