Com S 227
Spring 2018
Assignment 3
300 points

Due Date: Wednesday, March 28, 11:59 pm (midnight)
"Late" deadline: Thursday, March 29, 11:59 pm

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 Monday, April 2, which is just a few 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****.*

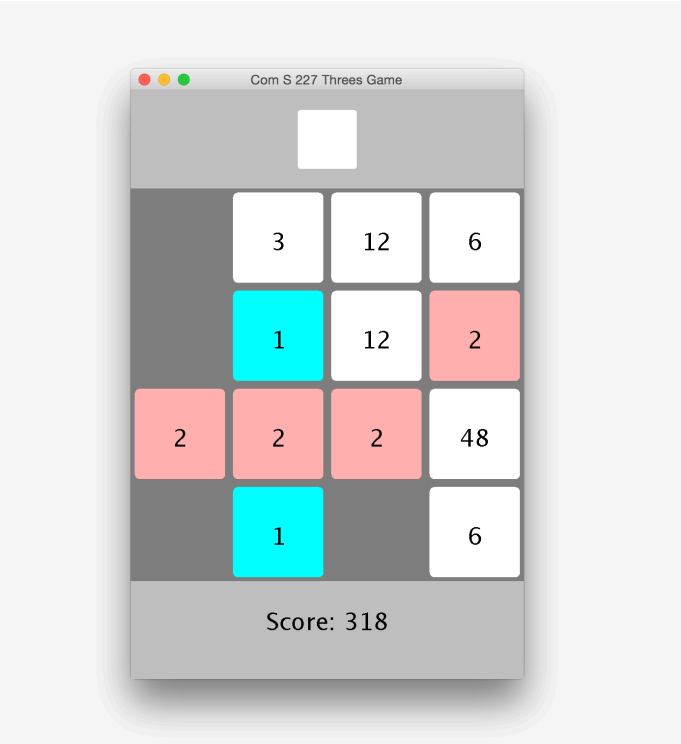
Please start the assignment as soon as possible and get your questions answered right away!

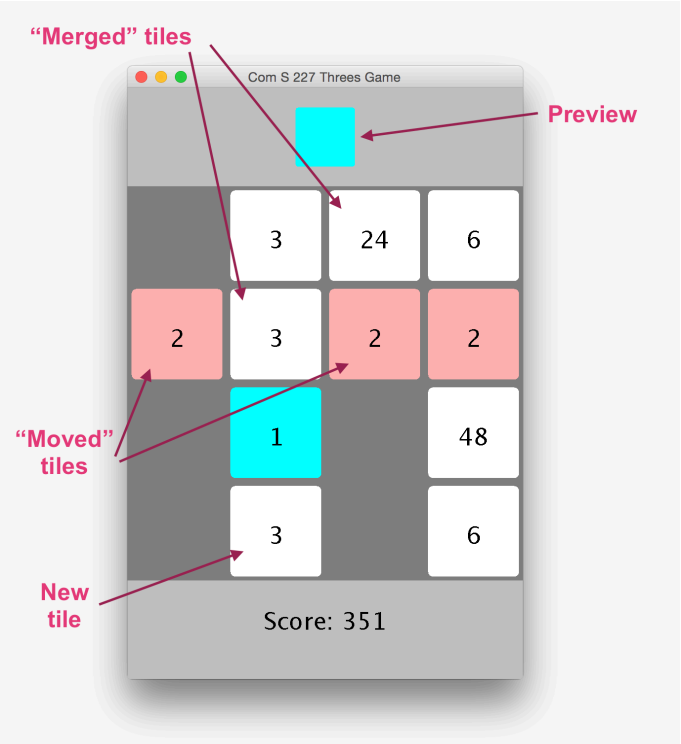
Introduction

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: `Game` and `GameUtil`. 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 the video game "Threes." If you are not familiar with the game, do not worry, it is not complicated.





The game consists of a grid of moveable *tiles*. Each tile contains a number 1, 2, 3, or a value that results from starting with a 3 and doubling a bunch of times. 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 certain combinations of tiles may be merged if "pushed" together against one of the boundaries. The exact rules for merging are discussed in a later section.

The screenshot on the right shows the result of shifting in the "up" direction. The two 12's are "pushed" against the top and merge to make 24. In addition, the 1 and 2 in the second column are merged to make a 3. All other tiles that can move (the two 2's in this case) are shifted up as well.

Whenever the grid is shifted in some direction, a new tile appears on the opposite side. The game ends when the grid can't be shifted in any direction because there are no empty cells and no merges are possible. The object of the game is to get the highest possible score. The score is the sum, for all tiles on the grid, of a predetermined number associated with each individual tile value.

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://play.threesgame.com/ *Please note that this and other versions you find online 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. There are more details below.

The sample code includes a documented skeleton for the two classes you are to create in the package **`hw3`**. A few of the methods of the **`GameUtil`** class are already implemented and you should not modify them. 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.

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

Specification

The specification for this asssignment includes

- this pdf,
- any "official" clarifications posted on Piazza, and
- the online javadoc

Overview of the GameUtil class

The class **`GameUtil`** is a "utility" class, meaning that it is stateless (has no instance variables). It consists of a collection of algorithms that implement some of the basic rules and logic of the game.

In particular, the key algorithm for shifting and merging the tiles in an individual row or column is implemented in the **`shiftArray()`** method of **`GameUtil`**. This method only operates on a one-dimensional array and only shifts to the left. We will later see how this simpler operation can be used easily by the **`Game`** class to shift in any direction. Isolating this special case (a one-dimensional form that only shifts left) makes the algorithm easier to write and test independently.

Here is a summary of the methods of this class. For full details see the javadoc:

`int mergeValues(int a, int b)`
Determines whether the two tile values can be merged and returns the merged value, returning zero if no merge is possible. The basic rules are that 1 can be merged with 2, and values larger than 2 can be merged if they are equal. In either case the merged value is the sum.
This method is already implemented.

`int getScoreForValue(int value)`
Returns the score for a tile with the given value.

`int calculateTotalScore(int[][] grid)`
Returns the total score for the given grid.
This method is already implemented.

`int[][] initializeNewGrid(int size, Random rand)`
Initializes a new grid, using the given Random object to position two initial tiles.
This method is already implemented.

`int[][] copyGrid(int[] grid)`
Makes a copy of the given 2D array.
This method is already implemented.

`int generateRandomTileValue(Random rand)`
Returns a randomly generated tile value according to a predefined set of probabilities, using the given Random object.

`TilePosition generateRandomTilePosition(int[] grid, Random rand, Direction lastMove)`
Returns a randomly selected tile position on the side of the grid that is opposite the direction of the most recent shift.

`ArrayList<Move> shiftArray(int[] arr)`
Shifts the array elements to the left according to the rules of the game, possibly merging two of the cells. See below for details.

The shiftArray() method

The basic rules for shifting are as follows. Remember that we interpret an array cell containing zero to be "empty".

- If there is an adjacent pair that can be merged, and has no empty cell to its left, then the leftmost such pair is merged (the merged value goes into the left one of the two cells) and all elements to the right are shifted one cell to the left.

- Otherwise, if there is an empty cell in the array, then all elements to the right of the leftmost empty cell are shifted one cell to the left.

- Otherwise, the method does nothing.

The method **`mergeValues()`** (see overview above) determines which pairs of values can be merged, and if so what the resulting value is. *Note that at most one pair in in the array is merged*. Here are some examples:

> For arr = [3, 1, 2, 1], after shiftArray(arr), the array is [3, 3, 1, 0]
> For arr = [1, 2, 1, 2], after shiftArray(arr), the array is [3, 1, 2, 0]
> For arr = [6, 3, 3, 3, 3], after shiftArray(arr), the array is [6, 6, 3, 3, 0]
> For arr = [3, 6, 6, 0], after shiftArray(arr), the array is [3, 12, 0, 0]
> For arr = [3, 0, 1, 2], after shiftArray(arr), the array is [3, 1, 2, 0]

The return value of **`shiftArray()`** is a list of **`Move`** objects. A **`Move`** object is a simple data container that encapsulates the information about a move of one cell or a merge of a pair of cells. The **`Move`** objects do not directly affect the game state, but can be used by a client (such as a GUI) to animate the motion of tiles. The **`Move`** class is in the **`api`** package; see the source code to see what it does.

Overview of the Game class

The **`Game`** class encapsulates the state of the game. The basic ingredients are

- an n x n grid (2D array of integers) representing the tiles
- a reference to an instance of **`GameUtil`**
- an instance of **`Random`** for generating new tile positions and values
- the direction of the most recent move
- an integer storing the value for the next tile to be generated
- a second 2D array for storing the previous state of the grid (to support the **`undo()`** operation

Basic game play: the shiftGrid() method

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

```
public ArrayList<Move> shiftGrid(Direction d),
```

which shifts each row or column in the indicated direction. This is normally followed by calling the method

```
public TilePosition newTile()
```

which (if anything in the grid was actually moved) puts a new tile in the grid.

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. You compare them to each other (or check whether equal to null) using the **`==`** operator.

Implementing shiftGrid()

The **`shiftArray`** method will make use of the algorithm implemented in **`GameUtil`** to shift the entire grid in the indicated direction. However, the **`shiftArray`** method in **`GameUtil`** 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 to do so is:

```
public int[] copyRowOrColumn(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 **`copyRowOrColumn(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 updateRowOrColumn(int[] arr, int rowOrColumn, Direction dir)
```

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

The method returns a list of **`Move`** objects. The **`Move`** objects themselves may be the same ones generated by the calls to the **`shiftArray`** method of **`GameUtil`**; however, in order to be interpreted as moves in a 2D grid, each **`Move`** object must have the appropriate direction and row/column index set by calling **`setDirection`**. Thus the basic implementation of **`shiftGrid()`** would normally look like:

for each index i up to the size
copy the row or column i into a temp array
call shiftArray with the temp array, and add the Move objects to the result
copy the temp array back into the row or column
update the row/column and direction for each Move object

The method newTile()

Each time a **`shiftGrid()`** operation actually moves one or more cells, a new tile will eventually have to appear in the grid. The purpose of the **`newTile()`** method is to select a new position and value for the tile using the game's instance of **`Random`**. The rules for selecting tile values are specified by the **`GridUtil`** method **`generateRandomTileValue()`**, and the game must use this method. Likewise, the rules for selecting new tile positions are specified in the **`GridUtil`** method **`generateRandomTilePosition()`**. The new tile is always placed in a randomly selected empty cell on the side of the grid that is *opposite* the move direction. The **`newTile()`** method updates the grid to contain the new tile value, and also returns a **`TilePosition`** object, which is just a simple data container for a row, column, and value.

The **`newTile()`** method should also recalculate the score. The score should include the value of the new tile that was just placed.

One thing that might seem odd at first is that the new tile *value* and the new tile *position* are not generated at the same time. A nice feature in playing the game is that the player gets a partial preview of the value that will appear on the next tile to be generated, and can base the decision about which direction to shift based on this preview. This means that your game must generate

each new value *one move before it is actually used*. Clients can observe this value by calling the `Game` method `getNextTileValue()`. (This method is an accessor and should not actually generate the value. Generating the value should happen once when the `Game` is constructed, and again each time `newTile()` puts a new tile on the grid.) You might also notice that `getNextTilePosition()` has an argument of type `Direction` in order to know the direction of the most recent move (to determine on which side of the grid to generate the new tile). That implies that you'll need an instance variable to keep track of this.

The undo() operation

Although the basic game play from the client point of view consists of calls to `shiftGrid()` alternated with calls to `newTile()`, there is one more feature the game supports that may cause this sequence to vary. Before the call to `newTile()`, the player can *undo* the shift operation and restore the grid to its previous state by calling the `undo()` method. This is actually easy to implement: define an instance variable of type `int[][]`, and use it to save a copy of the current grid at the beginning of the `shiftGrid()` method. If the shift operation causes any change to the grid, then the updated grid is considered "pending" until `newTile()` is actually called, after which the operation cannot be undone. If `undo()` is called before `newTile()`, the original grid is restored and the operation is no longer considered "pending". If `undo()` is called when no shift operation is pending, it has no effect; likewise, if `shiftGrid()` or `newTile()` is called when an operation is already pending, nothing should happen. If `shiftGrid()` does not actually cause any modification of the grid, that is not considered a pending operation.

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 very 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 list of `Move` objects returned by the `shiftGrid()` method, and it does not use the `TilePosition` object returned by the `newTile()` method, so you can try it out before you have those parts 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.

The controls are the four arrow keys and the shift key. Pressing an arrow key just invokes the game's **`shiftGrid()`** method in the corresponding direction. Releasing the key normally invokes the **`newTile()`** method; however, if the shift key is down while the arrow key is released, the UI instead invokes the **`undo()`** method.

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.

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 list of **`Move`** objects returned by the **`shiftGrid()`** method and the **`TilePosition`** object returned by **`newTile()`**, be completely valid. You can still try out the UI with animation off.

If you are curious to explore how the UI works, you are welcome to do so. In particular it is sometimes helpful to look at how the UI is calling the methods of the classes you are writing. 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 and a class **`PreviewPanel`** that contains the preview of the next tile to be generated. 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 about GUI development with Swing, 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. *In particular, when we grade your work we are NOT going to run the UI, we are going to verify that each method works according to its specification.*

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 specfications, 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.

> See the document "SpecChecker HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links" if don't remember how to import and run a SpecChecker.

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* **`Game`**.

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 (below 8) 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. Unzip the zip file containing the sample code.
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 Package Explorer, navigate to the src folder of the new project.
5. Drag the **`hw3`**, **`ui`**, and **`api`** folders from Explorer/Finder into the **`src`** folder in Eclipse.

More about grading

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a third) on functional tests that we run and partly on the grader's assessment of the quality of your code. Are you doing things in a simple and direct way that makes sense? Are you defining redundant instance variables? Some specific criteria that are important for this assignment are:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
    - You will lose points for having lots of unnecessary instance variables
    - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.
- Avoid code duplication. For example, the algorithm for shifting a row or column should be implemented ONLY in **`GameUtil`** method **`shiftArray()`**. The actual method **`shiftGrid()`** must not duplicate that logic.
- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. Use internal comments where appropriate to explain how your code works. (A good rule of thumb is: if you had to think for a few minutes to figure out how something works, you should probably include a comment explaining how it works.)
    - Here is a good example: take a look at the code for **`initializeNewGrid()`** in **`GameUtil`**, which is already implemented for you. Internal comments are used to explain a simple but possibly non-obvious strategy for selecting the new cells.
    - Internal comments always *precede* the code they describe and are indented to the same level.

See the "Style and documentation" section below for additional guidelines.

Style and documentation

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines:

- Each class, method, constructor and instance variable, whether public or private, must have a meaningful and complete Javadoc comment. Class javadoc must include the **`@author`** tag, and method javadoc must include **`@param`** and **`@return`** tags as appropriate.
    - Try to state what each method does in your own words, but there is no rule against copying and pasting the descriptions from this document.

    - Run the javadoc tool and see what your documentation looks like! You do not have to turn in the generated html, but at least it provides some satisfaction :)

- All variable names must be meaningful (i.e., named for the value they store).
- Your code should not be producing console output. You may add **`println`** statements when debugging, but you need to remove them before submitting the code.
- Try not to embed numeric literals in your code. Use the defined constants wherever appropriate.
- Use a consistent style for indentation and formatting.
    - Note that you can set up Eclipse with the formatting style you prefer and then use Ctrl-Shift-F to format your code. To play with the formatting preferences, go to Window->Preferences->Java->Code Style->Formatter and click the New button to create your own "profile" for formatting.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`assignment3`**. 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 **`assignment3`**. *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 posts from the instructors on Piazza 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.)

Getting started

At this point we expect that you know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification. *In particular, for this assignment we are not providing a specchecker that will perform any functional tests of your code.* It is up to you to test your own code (though you are welcome to share test cases on Piazza). There are some examples below to help you get started.

First, here are some basic observations from reading the spec:

- **`GameUtil`** does not depend on **`Game`** at all, so you can work on it independently
- The methods of **`GameUtil`** that you have to implement are independent of each other, so you can work on them separately
- Some methods of **`Game`** depend on **`GameUtil`**, namely **`shiftGrid()`** depends on **`shiftArray()`**, **`newTile()`** depends on **`generateRandomTilePosition()`**, **`generateRandomTileValue()`** and **`calculateTotalScore()`**. However there are still quite a few things you can implement in **`Game`** before you have **`GameUtil`** all done.
- The list of **`Move`** objects returned by the **`shiftArray()`** method is not directly used within **`Game`**, so as an incremental step you can implement **`shiftArray()`** without constructing the move list
- You can implement **`shiftGrid()`** and **`newTile()`** at first without worrying about the **`undo()`** operation

So, you certainly don't have to do the steps below in exactly the order given.

1. You could try working on **`shiftArray()`.** Start with some test cases based on the examples on page 5. Initially, don't worry about the list to return, just return null. Start with just the problem of shifting without a merge, for example:

```
GameUtil util = new GameUtil();
int[] test = {3, 0, 1, 2};
util.shiftArray(test);
System.out.println(Arrays.toString(test)); // expected [3, 1, 2, 0]
```

Test it with 0's at the beginning, multiple 0's, no 0's. Then think about adding the case for merge. To tell whether two values should merge (and what the result would be) use the **`mergeValues()`** method, which is already implemented. Can you write a loop that finds the index of a pair to merge? Can you write a loop to find the index of the first 0 *or* the first merge-able pair, (whichever comes first)? Can you arrange for the left cell of the merged pair to update?

2. You could think about the basics of **`Game`**. You can deduce from the constructor specification that you'll need instance variables for the given **`GameUtil`** object and **`Random`** object. You'll also need an instance variable for a 2D array of ints to represent the grid. To initialize it, your constructor just needs to call the **`GameUtil`** method **`initializeNewGrid()`**.
Once you have the grid defined, it is easy to write **`getCell()`**, **`setCell()`**, and **`getSize()`**. Make sure they work. There are also accessors **`getScore()`** and **`getNextTileValue()`**, suggesting you might need two more instance variables. The score should clearly be initialized to zero, but the "next tile value" is not so obvious. But if you carefully read the spec for **`newTile()`** you'll see that this should be initialized with a call to the **`GameUtil`** method **`generateRandomTileValue()`**. The idea is to generate the value for a new tile *one step before* it's actually going to be used on a tile, so the client can get a preview of what the number on the next tile will be. (To test the initialization, modify the **`generateRandomTileValue()`** method so it returns something recognizable, like 42.)

3. Once you have the game grid defined along with **`getCell()`** and **`setCell()`**, you can implement the methods **`copyRowOrColumn()`** and **`updateRowOrColumn()`**. Start with a simple test to visualize what they do:

```
Game g = new Game(5, new GameUtil(), new Random(42));
int[] arr = {1, 2, 3, 4, 5};
System.out.println("Before:");
ConsoleUI.printGrid(g);
g.updateRowOrColumn(arr, 2, Direction.DOWN);
System.out.println("After:");
ConsoleUI.printGrid(g);
```

This should produce output something like the following:

```
Before:
---------------
   0   0   0   2   0
   1   0   0   0   0
   0   0   0   0   0
   0   0   0   0   0
   0   0   0   0   0
---------------
After:
---------------
   0   0   5   2   0
   1   0   4   0   0
   0   0   3   0   0
   0   0   2   0   0
   0   0   1   0   0
---------------
```

Here we are using a static method in **`ConsoleUI`** that just neatly prints out the grid. Note that the initial state includes a randomly positioned 1 and 2, but since we are providing an argument

(called the "seed") to the `Random` constructor, the test will be reproducible. Then to test `copyRowOrColumn()`, just read the same row or column back again:

```
int[] result = g.copyRowOrColumn(2, Direction.DOWN);
System.out.println(Arrays.toString(result)); // expected [1, 2, 3, 4, 5]
```

4. Once you have `copyRowOrColumn()` and `updateRowOrColumn()`, you can start on `shiftGrid()`. The basic logic is described on page 7. In your first attempt, you would probably want to ignore the `Move` list and worry about that later. First, just see whether the method has the right effect on the grid. To test it, you can use the `setCell()` method to set up the grid however you want for testing. Here is an example.

```
Game g = new Game(4, new GameUtil(), new Random(42));
int[][] test =
  {
    {0, 2, 3, 1},
    {0, 1, 3, 2},
    {0, 2, 3, 0},
    {0, 1, 2, 0}
  };
for (int row = 0; row < test.length; row += 1)
{
  for (int col = 0; col < test[0].length; col += 1)
  {
    g.setCell(row, col, test[row][col]);
  }
}
System.out.println("Before: ");
ConsoleUI.printGrid(g);
g.shiftGrid(Direction.DOWN);
System.out.println("After: ");
ConsoleUI.printGrid(g);
```

which should produce the output,

```
Before:
---------------
   0   2   3   1
   0   1   3   2
   0   2   3   0
   0   1   2   0
---------------
After:
---------------
   0   0   0   0
   0   2   3   1
   0   1   6   2
   0   3   2   0
---------------
```

5. You next might want to take a whack at **`newTile()`**. However, you'll need to be sure that **`generateRandomTilePosition()`** does something sensible first. This method is a bit tricky to get absolutely right, but one thing you could do, as an incremental step, is to return *some* empty position on the correct side of the grid, not necessarily a randomly selected one. For example, you could return the lowest-indexed empty position on the correct side of the grid. Try it:

```
int[][] test =
  {
    {0, 2, 3, 1},
    {0, 1, 3, 2},
    {0, 2, 3, 0},
    {0, 1, 2, 0}
  };
GameUtil util = new GameUtil();
TilePosition tp =
    util.generateRandomTilePosition(test, new Random(), Direction.LEFT);
System.out.println(tp); // expected Position (2, 3) value 0
```

6. Then write **`newTile()`**. Its main job is to call **`generateRandomTilePosition()`** and place the tile on the grid at that position, using the tile value you have saved from a previous call to **`generateRandomTileValue()`**. Then, of course, you should update the "next tile value" and recalculate the score. The change in score won't be apparent until you implement **`getScoreForValue()`** but you could temporarily have it return 1 or some nonzero value for testing **`newTile()`**. Note that **`generateRandomTilePosition()`** requires an argument indicating the direction of the previous move. You'll need an instance variable to keep track of this.

There is something else you have to think about here, which is the requirement that **`nextTile()`** does nothing if there wasn't anything actually anything moved by a previous call to **`shiftGrid()`**. You'll have to keep track of this information, suggesting another instance variable. In order to set it correctly in **`shiftGrid()`**, you'll need to be able to determine whether anything about the grid actually changed. One possible solution is to use the **`Move`** lists that are returned from **`shiftArray()`** - if you end up with no moves, well, then nothing moved. Another possibility is to make a copy of grid before doing any shifting, and then check to see whether the grid is the same. Test the method using a sample grid as you did for **`shiftGrid()`**. Notice that **`TilePosition`** has a **`setValue()`** method that you can use to set the value in the object that is returned by **`newTile()`**.

7. Note that with this much done, you should be able to play the game with the **`ConsoleUI`**, though the "undo" part won't work. It should also be possible to use the GUI, though you won't see any animation of the tiles until you implement the **`Move`** lists.

8. You can implement **`generateRandomTileValue()`** anytime. It's not hard. For example, to get value X to randomly occur 40% of the time, just randomly generate one of ten possible values, and for four of them, return X.

9. You can also work on the **`GameUtil`** method **`getScoreForValue()`** anytime. It's easy to test:

```
GameUtil util = new GameUtil();
System.out.println(util.getScoreForValue(1)); // expected 0
System.out.println(util.getScoreForValue(2)); // expected 0
System.out.println(util.getScoreForValue(3)); // expected 3
System.out.println(util.getScoreForValue(6)); // expected 9
System.out.println(util.getScoreForValue(48));// expected 243
```

10. At some point, you'll want to think about the list of **`Move`** objects that is returned from **`shiftArray()`**. First read the javadoc for **`Move`**, so you can see what it does. It is a simple data container. A **`Move`** basically describes a move of a value from one position to another, within the same row or column. From the point of view of **`shiftArray()`**, the direction of the move and the row/column are irrelevant. Use the three-argument constructor for a move of a value, and use the five-argument constructor for a merge of two values. (You can ignore the six-argument constructor, that is only needed for potential generalizations of the Threes game and is not needed here.) As always, start with some simple test cases. Here's one that has just one move:

```
GameUtil util = new GameUtil();
int[] test = {6, 0, 12};
ArrayList<Move> result = util.shiftArray(test);
System.out.println(Arrays.toString(test));
System.out.println(result);
```

This should produce the output,

```
[6, 12, 0]
[Move 2 to 1]
```

Here's one with two moves:

```
GameUtil util = new GameUtil();
int[] test = {0, 6, 12};
ArrayList<Move> result = util.shiftArray(test);
System.out.println(Arrays.toString(test)); // expected [6, 12, 0]
System.out.println(result);
```

Which produces output,

```
[6, 12, 0]
[Move 1 to 0, Move 2 to 1]
```

(Note that the **`Move`** type has a built-in method **`toString()`** that is used by **`println`** when printing the ArrayList. Note also that the order of the moves within the list is unspecified, so you add them in any order that is convenient for you.)

Try one with a merge too:

```
GameUtil util = new GameUtil();
int[] test = {3, 1, 2, 0, 4};
ArrayList<Move> result = util.shiftArray(test);
System.out.println(Arrays.toString(test)); // expected [6, 12, 0]
System.out.println(result);
```

which results in,

```
[3, 3, 0, 4, 0]
[Merge 2 to 1, Move 4 to 3]
```

11. Once you have the right list of moves returned from **`shiftArray()`** in **`GameUtil`**, you can assemble the list to return from **`shiftGrid()`**. Basically you just have to add all moves you get from **`shiftArray()`** into a common ArrayList to return. The catch is that as you do so, you have to fill in two more pieces of information, namely, the row or column for the move, and the direction for the move. These things are unknown in the **`shiftArray()`** method, but are known within **`shiftGrid()`**.Take the example from step 4 and print out the moves:

```
Game g = new Game(4, new GameUtil(), new Random(42));
int[][] test =
  {
    {0, 2, 3, 1},
    {0, 1, 3, 2},
    {0, 2, 3, 0},
    {0, 1, 2, 0}
  };
for (int row = 0; row < test.length; row += 1)
{
  for (int col = 0; col < test[0].length; col += 1)
  {
    g.setCell(row, col, test[row][col]);
  }
}

ArrayList<Move> moves = g.shiftGrid(Direction.DOWN);
System.out.println(moves);
```

This should result in the following output (reformatted on multiple lines for clarity):

```
[Merge 1 to 0 (column 1 DOWN),
Move 2 to 1 (column 1 DOWN),
Move 3 to 2 (column 1 DOWN),
Merge 2 to 1 (column 2 DOWN),
Move 3 to 2 (column 2 DOWN),
Move 2 to 1 (column 3 DOWN),
Move 3 to 2 (column 3 DOWN)]
```

Remember the order of the moves within the list is unspecified. Note also that the string output of a **`Move`** includes the row/column and direction only if they have been set.

12. The "undo" mechanism is not hard. Whenever **`shiftGrid()`** is called, before you do anything else, save a copy of the grid. If **`newTile()`** is called next, you can ignore the copy, but if **`undo()`** is called, you can use it to restore the grid. As discussed in step 6, you should already be keeping track of whether something was moved by a previous call to **`shiftGrid()`**. Call this a "pending" move, in the sense that it can be undone by a call to **`undo()`** or confirmed by a call to **`newTile()`**. So calling **`undo()`** should do nothing if there is no pending move, calling **`newTile()`** should do nothing if there is no pending move, and **`shiftGrid()`** should do nothing if there is *already* a pending move. If there is a pending move, then calling **`undo()`** or **`newTile()`** should "clear" it somehow so that you can't, for example, call **`newTile()`** twice in a row and get two new tiles.

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, **`Game.java`** and **`GameUtil.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 in the Piazza pinned messages under “Syllabus, office hours, useful links.”

> 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 two files **Game.java** and **GameUtil.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.