# Com S 227
# Fall 2023
# Assignment 3
# 300 points

Due Date: Friday, November 3, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Nov 2)

10% penalty for submitting 1 day late (by 11:59 pm Nov 4)

No submissions accepted after Nov 4, 11:59 pm

**This assignment is to be done on your own. See the Academic Integrity policy in the syllabus, 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.

*Please start the assignment as soon as possible and get your questions answered right away. It is physically impossible for the staff to provide individual help to everyone the afternoon that the assignment is due!*

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a third) on the specchecker's functional tests and partly on the grader's assessment of the quality of your code. See the "More about grading" section.

## Contents

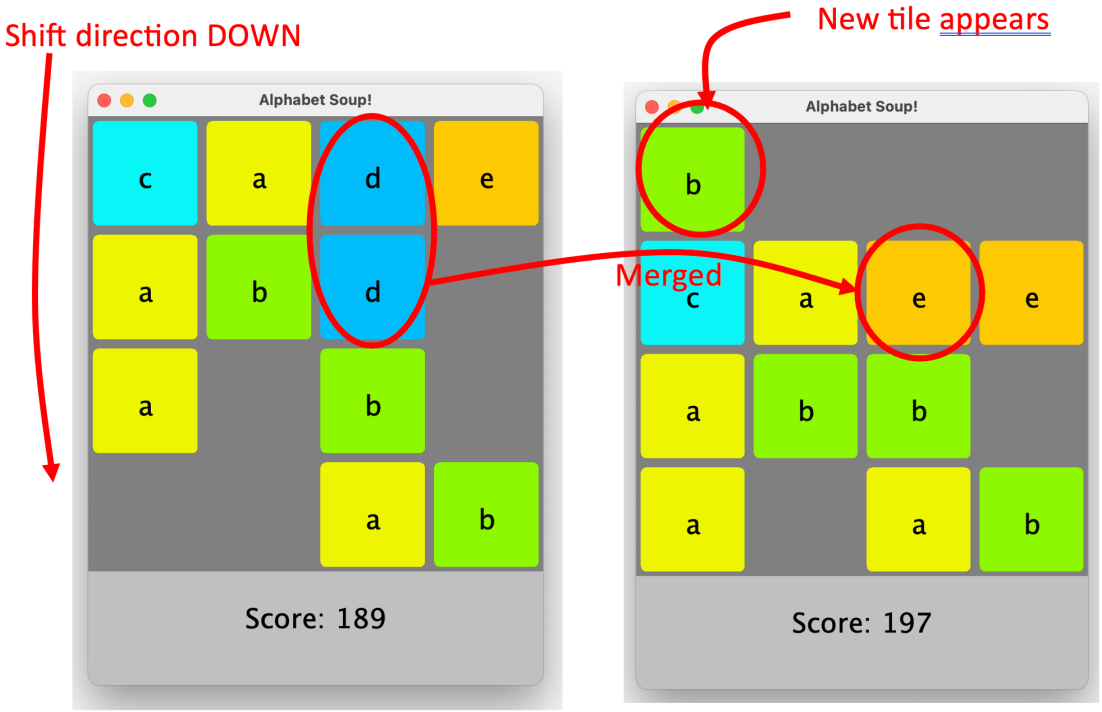
## Introduction

You will be writing the logic for a version of a video puzzle game that we call “Alphabet soup” which shares some elements with the popular game “2048.” You'll get lots of practice working with arrays and ArrayLists. Once you have your code working, you'll be able to integrate it with a GUI that we have provided to create a complete application.

Your task will specifically be to implement the two classes **`hw3.AlphabetSoup`** and **`hw3.AlphabetUtil`**.

The idea of the game is not complicated. There is an n x n grid of squares (that we call "tiles"), each labeled with a letter. There are four possible moves: to shift the grid up, down, left, or right by one square. When two adjacent tiles with the same letter are "pushed" together, they merge into a new one with the next letter of the alphabet. The illustration below shows one configuration of a 4x4 game on the left, and on the right is the result of a "shift DOWN" move. Notice that

- The two 'd' tiles are pushed together against the ones below, so they merge into an 'e'
- The two 'a' tiles at left *don't* merge, since they aren't pushed together against other tiles or against the boundary
- Everything else than *can* move shifts down one square
- A new tile appears on the opposite side of the shift direction



The object of the game is to earn the highest possible score before it becomes impossible to shift the grid in any direction. The score is just the sum, over all tiles on the grid, of a predetermined value associated with each letter (where the later letters of the alphabet are worth more points). The exact rules for merging and scoring are covered in a later section.

The two classes you implement will provide the "backend" or core logic for the game. The grid itself will be represented in your code as a 2D array of integers. A zero in an array cell is interpreted as an empty position, and each nonzero cell is interpreted as a tile with a corresponding letter of the alphabet, where 1 is 'a', 2 is 'b', and so on.

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, from which the screenshots above were produced. There is also a simple text-based user interface that is more useful for testing.

The sample code includes a partial 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, as well as the console UI. The **`api`** package contains some relatively boring types for representing data in the game.

You should not modify the code in the **`api`** package. We recommend that you don't try to modify the GUI, except for editing **`GameMain`**.

## Specification

The specification for this assignment includes this pdf, the online Javadoc, and any "official" clarifications announced on Canvas.

There are some usage examples later in this document in the "Getting started" section.

## Overview of the AlphabetUtil class

The class **`AlphabetUtil`** is a "utility" class, meaning that it is stateless (has no instance variables). It consists of some algorithms for some of the basic rules and logic of the game that can be implemented separately, without direct access to the game state. This simplifies the implementation and testing[1], and also makes it easy to later reconfigure the game with different tile values, scoring, or merging rules.

[1] In fact, in principle all the methods could have been specified to be static. We have made them non-static only because doing so makes it easier for us to test your **AlphabetSoup** using our working version of **AlphabetUtil**, if necessary.

In particular, the key algorithm for shifting and merging the tiles in an individual row or column is implemented in the **`doShift()`** method of **`AlphabetUtil`**. 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 **`AlphabetSoup`** 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 findMergedValue(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). For this game, two tiles can be merged if they are nonzero and have the same value. The merged value is the current value plus 1.

**`int getScoreForOneTile(int value)`**
Returns the score for a tile with the given value. For this game, the score is 3 raised to the power of the tile value minus 1.

**`int randomNewTileValue(Random rand)`**
Returns a randomly generated tile value according to a predefined set of probabilities, using the given **`Random`** object. For this game, the method should return either a 1 or a 2 with equal probability.

**`ArrayList<ShiftDescriptor> doShift(int[] arr)`**
Shifts the array elements to the left according to the rules of the game, possibly merging some pairs of cells.

### The doShift() 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. A zero is placed in the rightmost cell.

- 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. A zero is placed in the rightmost cell.

- Otherwise, the method does nothing.

The method **`findMergedValue()`** (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 = [1, 3, 3, 2], after doShift(arr), the array is [1, 4, 2, 0]
For arr = [1, 2, 1, 0], after doShift(arr), the array is [1, 2, 1, 0]
For arr = [7, 0, 3, 0, 2], after doShift(arr), the array is [7, 3, 0, 2, 0]
For arr = [2, 2, 2, 2], after doShift(arr), the array is [3, 2, 2, 0]
For arr = [3, 0, 2, 2], after doShift(arr), the array is [3, 2, 2, 0]
For arr = [0, 0, 5, 0, 7, 7], after doShift(arr), the array is [0, 5, 0, 7, 7, 0]

The return value of **`doShift()`** is a list of **`ShiftDescriptor`** objects. A **`ShiftDescriptor`** 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 **`ShiftDescriptor`** 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 **`ShiftDescriptor`** class is in the **`api`** package; see the source code to see what it does.

## Overview of the AlphabetSoup class

The **`AlphabetSoup`** 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 **`AlphabetUtil`**
- an instance of **`Random`** for generating new tile positions and values
- the direction of the move in progress, if any

The constructor should initialize the grid so that all but two cells are zeros. The two nonzero cells should have locations selected at random, and values determined by **`randomNewTileValue()`**. The given **`Random`** instance is used for all randomization in the game. The idea is that we can choose to construct the game using a *seeded* instance of **`Random`**, which means that the same sequence of pseudorandom values is generated every time. This makes testing easier, since anything you do is repeatable.

### Basic game play: the shift() and setNewTile() methods

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

```
public ArrayList<ShiftDescriptor> shift(Direction d),
```

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

```
public TileInfo setNewTile()
```

which (if anything in the grid was actually moved) places a new tile in the grid, at a randomly selected empty cell on the side *opposite* the direction of the most recent shift.

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 shift()**

The **`shift`** method will make use of the algorithm implemented in **`AlphabetUtil`** to shift the entire grid in the indicated direction. However, the **`doShift`** method in **`AlphabetUtil`** only shifts 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[] getRowColumn(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 **`getRowColumn(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 setRowColumn(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 **`ShiftDescriptor`** objects. The **`ShiftDescriptor`** objects themselves may be the same ones generated by the calls to the **`doShift`** method of **`AlphabetUtil`**; however, in order to be interpreted as moves in a 2D grid, each **`ShiftDescriptor`** object must have the appropriate direction and row/column index filled in by calling **`setDirection`**. This has to be done within **`shift()`**, since the **`doShift`** method in **`AlphabetUtil`** has no knowledge of the original row/column index or move direction - it only operates on a 1D array.

Note that the **`ShiftDescriptor`** objects are not generally used within the implementation of the **`AlphabetSoup`** game logic - their only purpose is to communicate information to clients, such as a GUI, about moved or merged tiles.

Thus the basic implementation of **`shift()`** would normally look like:

*for each index i up to the size*
*copy the row or column i into a temporary array*
*call doShift with the temp array, and add the ShiftDescriptor objects to the result*
*copy the temp array back into the original row or column*
*update the row/column and direction in each ShiftDescriptor object*

**The method setNewTile()**

Each time a **`shift()`** operation actually moves one or more cells, a new tile will eventually have to appear in the grid. The purpose of the **`setNewTile()`** 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 **`AlphabetUtil`** method **`randomNewTileValue()`**, and the game *must* use this method. The **`setNewTile()`** method always places the new tile in a randomly selected empty cell on the side of the grid that is *opposite* the direction of the previous move. The **`setNewTile()`** method updates the grid to contain the new tile value, and also returns a **`TileInfo`** object, which is just a simple data container for a row, column, and value. As with the **`ShiftDescriptor`** array returned by **`shift()`**, the **`TileInfo`** object is not directly used within the implementation of the **`AlphabetSoup`** game state; again, its only purpose is to

communicate information to clients. The **`TileInfo`** class is in the **`api`** package; see the source code to see what it does.

You might also notice that **`setNewTile()`** has to know the direction of the most recent move (to determine on which side of the grid to generate the new tile), so this is something you'll need to keep track of, and this information is exposed to clients in the method **`getLastDirection()`**. After **`shift()`** has been called but before a corresponding call to **`setNewTile()`**, **`getLastDirection()`** returns the direction of the shift, provided that one or more tiles were actually moved. At all other times, **`getLastDirection()`** returns null. To help prevent client errors, **`shift()`** is defined to do nothing (and return an empty list) if **`getLastDirection()`** is non-null, and **`setNewTile()`** is defined to do nothing (and return null) if **`getLastDirection()`** is null.

## The text-based UI

The **`ui`** 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 **`ShiftDescriptor`** objects returned by the **`shift()`** method, and it does not use the **`TileInfo`** object returned by the **`setNewTile()`** 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 read it.

The controls are the four arrow keys. Pressing an arrow key just invokes the game's **`shift()`** method in the corresponding direction. Releasing the key invokes the **`setNewTile()`** 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.

It's important to realize that the GUI *contains no actual logic or data for the game*. At all times, it simply displays the information returned by your **`getCell()`** method, using the values of your **`getSize()`** method for the size of the grid to display. It invokes the **`shift()`** method when the user presses an arrow key and it invokes the **`setNewTile()`** method when the user releases the key.

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 **`ShiftDescriptor`** objects returned by the **`shift()`** method, and the **`TileInfo`** object returned by **`setNewTile()`**, 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. 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, 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.

## Importing the sample code

The sample code includes a skeleton of the two classes you are writing in the **`hw3`** package along with supporting code in the **`api`** and **`ui`** packages. It is distributed as a complete Eclipse project that you can import. It should compile without errors out of the box. *However the GUI will not run correctly until you have implemented the basic functionality of the game.* Basic instructions for importing:

1. Download the zip file to a location outside your workspace. 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.

*Alternate procedure:* 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 File Explorer or OS X Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse. **Be sure to UNCHECK the box for "Create module-info"**
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.

## 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 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 the UI, we are going to test 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 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 encouraged 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.

## Suggestions for getting started

*Remember to work **incrementally** and test new features as you implement them. At this point in the course, we expect that you can study this document and read the javadoc and construct usage examples or simple test cases to guide your implementation. Here are some suggestions for how you might approach the process of testing. Note that the code snippets below can be found in the SimpleTest class in the default package of the sample code.*

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:

- **`AlphabetUtil`** does not depend on **`AlphabetSoup`** at all, so you can work on it independently
- The methods of **`AlphabetUtil`** that you have to implement are largely independent of each other, so you can work on them separately (the exception is that **`doShift()`** depends on **`findMergedValue()`**, which is easy).
- Some methods of **`AlphabetSoup`** depend on **`AlphabetUtil`**, namely, **`shift()`** depends on **`doShift()`**, and **`setNewTile()`** depends on **`randomNewtileValue()`.**
- The list of **`ShiftDescriptor`** objects returned by the **`doShift()`** method is not directly used within **`AlphabetSoup`**, so as an incremental step you can implement **`doShift()`** without constructing the move list, and just have **`shift()`** return an empty list.
- Likewise, the **`TileInfo`** object returned by **`setNewTile()`** is not used within **`AlphabetSoup`**, so you can first implement **`setNewTile()`** and just have it return null.

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

1. Some easy things to start with would be the **`AlphabetSoup`** constructor. You can deduce from the constructor specification that you'll need instance variables for the given **`AlphabetUtil`** object and **`Random`** object. You'll also need an instance variable for a 2D array of ints to represent the grid. Once you have the grid defined, it is easy to write **`getCell()`**, **`setCell()`**, and **`getSize()`**. Make sure they work. In the finished product, your constructor should select two random positions in the grid and initialize them using values from **`randomNewtileValue()`**, but for now, you could just put a 1 in the first two cells.

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

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

This should produce output something like the following:

```
Before:
---------------
   -   -   -   a   -
   b   -   -   -   -
   -   -   -   -   -
   -   -   -   -   -
   -   -   -   -   -
---------------
After:
---------------
   -   -   e   a   -
   b   -   d   -   -
   -   -   c   -   -
   -   -   b   -   -
   -   -   a   -   -
---------------
```

Here we are using a static method in **`ConsoleUI`** that just neatly prints out the grid. Note that the initial state includes some randomly positioned 'a' or 'b', but since we are providing an argument (called the "seed") to the **`Random`** constructor, the test will be reproducible. Then to test **`getRowColumn()`**, just read the same row or column back again:

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

Tip regarding the **`Direction`** constants: you can save some typing by adding the declaration:
**`import static api.Direction.*;`**

to the top of your AlphabetSoup class. Then you can just write, e.g. "**`DOWN`**" instead of "**`Direction.DOWN`**". Remember that enums are compared using **`==`**, not **`.equals()`**.

3. You could also go ahead and implement the easy methods of **`AlphabetUtil`**: **`findMergedValue()`**, **`getScoreForOneTile()`**, and **`randomNewTileValue()`** are all quite short. Try them out:

```
AlphabetUtil util = new AlphabetUtil();

// test get score for a tile
int score = util.getScoreForOneTile(4);
System.out.println(score);
System.out.println("Expected 27");

// test merge
int canMerge = util.findMergedValue(5, 5);
System.out.println(canMerge);
System.out.println("Expected 6");

canMerge = util.findMergedValue(2, 3);
System.out.println(canMerge);
System.out.println("Expected 0");
```

4. Next you'll need to work on **`doShift()`**. 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:

```
int[] test2 = {7, 0, 3, 0, 2};
util.doShift(test2);
System.out.println(Arrays.toString(test2));
System.out.println("Expected [7, 3, 0, 2, 0]");
```

Test it with 0's at the beginning, multiple 0's, and no 0's. Then think about adding the case for merging. To tell whether two values should merge (and what the result would be) use the **`findMergedValue()`** method. Can you write a loop that finds the index of the first pair that can be merged? 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?

```
int[] test = {1, 3, 3, 2};
util.doShift(test);
System.out.println(Arrays.toString(test));
System.out.println("Expected [1, 4, 2, 0]");
```

4. Once you have the pieces above, you can start on **`shift()`**. The basic logic is described on page 7. In your first attempt, you would probably want to ignore the **`ShiftDescriptor`** 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.

```
g = new AlphabetSoup(4, new AlphabetUtil(), new Random(42));
int[][] testGrid1 =
{
    { 0, 2, 3, 1 },
    { 0, 1, 3, 2 },
    { 0, 2, 3, 0 },
    { 0, 1, 2, 0 }
};
for (int row = 0; row < testGrid1.length; row += 1)
{
  for (int col = 0; col < testGrid1[0].length; col += 1)
  {
    g.setCell(row, col, testGrid1[row][col]);
  }
}
System.out.println("Before: ");
ConsoleUI.printGrid(g);
g.shift(Direction.DOWN);
System.out.println("After: ");
ConsoleUI.printGrid(g);
```

which should produce the output,

```
Before:
---------------
   -   b   c   a
   -   a   c   b
   -   b   c   -
   -   a   b   -
---------------
After:
---------------
   -   b   -   -
   -   a   c   a
   -   b   d   b
   -   a   b   -
---------------
```

This would also be a good time to implement **`getLastDirection()`**:

```
// try getLastDirection
System.out.println(g.getLastDirection());
System.out.println("Expected DOWN");
```

5. You next might want to take a whack at **`setNewTile()`**. This method is a bit tricky to get absolutely right, but one thing you could do, as an incremental step, is to set a value in *some* empty position on the correct side of the grid, not necessarily a randomly selected one. For example, you could use the lowest-indexed empty position on the correct side of the grid. To test, you'll need to set up a grid and call **`shift()`**, so that the last direction is non-null. For example, if we use the same example testGrid1 as above and shift **`DOWN`**, then your **`setNewTile()`** method should place a new value at index 0, 2, or 3 in the top row. The value should be generated by **`randomNewTileValue()`**.

Try it:

```
TileInfo info = g.setNewTile();
ConsoleUI.printGrid(g);
System.out.println();
```

Once you have decided on a row, column, and value to set in the grid, constructing a **`TileInfo`** object to return is easy. Take a look at the code to see how to construct one. Check to see that it matches your new tile:

```
System.out.println("TileInfo: " + info);
```

Note: To do the randomization, you might be tempted to write a while loop something like this:

```
i = myRandom.nextInt(getSize());
while (grid[0][index] != 0)
        i = myRandom.nextInt(size);
```

**Please don't do that!** It's ugly and inefficient and there are better ways to solve this problem. You could count the number **`n`** of empty cells in the relevant row or column, and then use **`nextInt(n)`** to get a random index **`i`**, and then put your new value in the **`i`**th empty cell. Or you could put the indices of the empty cells in an ArrayList and select one of its elements at random. (Also note that since you may be dealing with the top or bottom row, and the leftmost or rightmost column, it might be convenient to use getRowColumn to convert to a one-dimensional array first...)

6. Note that with this much done, you should be able to play the game with the **`ConsoleUI`**. It should also be possible to use the GUI, if you go into **`GameMain`** and turn animation off.

7. On the subject of randomization, at some point you need to correctly initialize the grid in your constructor. Here is one way to think about it: You have an n x n grid, so there are $n^2$ possible cells. So start by selecting two distinct numbers from 0 to $n^2$ : Suppose you use nextInt($n^2$) to get the first one, say *x*. Now you have $n^2$- 1 choices for the second one, since you have to avoid *x*. So use nextInt($n^2$- 1) but if the result is >= *x*, add 1. Finally, imagine those $n^2$ possible indices

are your rows, laid out end to end: you can divide by n to get the row index and mod by n to get the column index.

8. Finally, you'll need to work on the **`ShiftDescriptor`**s. The first step is to create a list for **`doShift()`** to return. Look at the javadoc for **`ShiftDescriptor`**; it is very simple. There are two constructors: one for simple moves and one for merges.

For example, using the array [7, 0, 3, 0, 2] from the previous example, there should be two descriptors: one for moving the 3 from position 2 to position 1, and one for moving the 2 from position 4 to position 3:

```
int[] test3 = {7, 0, 3, 0, 2};
ArrayList<ShiftDescriptor> descriptors = util.doShift(test3);
System.out.println(descriptors);
System.out.println("Expected:");
System.out.println("[Move c 2 to 1, Move b 4 to 3]");
```

(The **`ShiftDescriptor`** class has a **`toString()`** method that is automatically called when we print the elements of the ArrayList with System.out.println, which is where the above output comes from.)
Then try one with a merge:

```
int[] test4 = {1, 3, 3, 2};
descriptors = util.doShift(test4);
System.out.println(descriptors);
System.out.println("Expected:");
System.out.println("[Merge c 2 to 1, Move b 3 to 2]");
```

9. The last piece is to assemble the list of descriptors to be returned from **`shift()`**. Each time **`doShift()`** is called from **`shift()`** for a row or column, you'll get some descriptors back, and you can add these to a common list. But first, they have to be filled in with the *direction* of the shift and the *row or column index*, since that information is not known within **`doShift()`**. There is a method **`setDirection()`** in the **`ShiftDescriptor`** class for this purpose. Here is an example of what it should do:

```
g = new AlphabetSoup(4, new AlphabetUtil(), new Random(42));
int[][] testGrid2 =
{
    { 0, 2, 3, 1 },
    { 0, 1, 3, 2 },
    { 0, 2, 3, 0 },
    { 0, 1, 2, 0 }
};
```

```
    for (int row = 0; row < testGrid2.length; row += 1)
    {
      for (int col = 0; col < testGrid2[0].length; col += 1)
      {
        g.setCell(row, col, testGrid2[row][col]);
      }
    }

    System.out.println("Before: ");
    ConsoleUI.printGrid(g);
    ArrayList<ShiftDescriptor> moves = g.shift(Direction.DOWN);
    System.out.println("After: ");
    ConsoleUI.printGrid(g);

    System.out.println(moves);
```

The expected output is (slightly reformatted for readability):

```
Before:
---------------
   -   b   c   a
   -   a   c   b
   -   b   c   -
   -   a   b   -
---------------
After:
---------------
   -   b   -   -
   -   a   c   a
   -   b   d   b
   -   a   b   -
---------------

[Merge c 2 to 1 (column 2 DOWN),
Move c 3 to 2 (column 2 DOWN),
Move b 2 to 1 (column 3 DOWN),
Move a 3 to 2 (column 3 DOWN)]
```

However, remember the order of the descriptors within the list is unspecified, so yours might be differently ordered. Note also that the string output of a descriptor includes the row/column and direction *only* if they have been set.

10. Once you have the descriptor list correctly returned from **`shift()`**, you should be able to enable animation in the GUI. ***Warning: do not rely on the GUI for testing! If something in the GUI doesn't look right, it is VERY difficult to guess where a bug might be in your code. Write small, simple test cases like the ones above to check that your code is doing what it should.***

## 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 the specchecker's functional tests and partly on the TA's assessment of the quality of your code. This means you can get partial credit even if you have errors, and it also means that even if you pass all the specchecker tests you can still lose points. 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 unnecessary instance variables
    - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.

- You should not have a lot of redundant code. If a set of actions is repeated in multiple places, create a helper method.

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 in addition to those noted above.

- **Each class, method, constructor and instance variable, whether public or private, must have a meaningful javadoc comment**. The javadoc for the class itself can be very brief, but must include the **`@author`** tag. The javadoc for methods must include **`@param`** and **`@return`** tags as appropriate.
    - Try to briefly state what each method does in your own words. However, there is no rule against copying the descriptions from the online documentation.
    - 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.

- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. (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 an internal comment explaining how it works.)
    - Internal comments always *precede* the code they describe and are indented to the same level.
- 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 **`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. (In the Piazza editor, use the button labeled "pre" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "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 instructors on Canvas 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 Canvas. (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, **`AlphabetSoup.java`** and **`AlphabetUtil.java`**. Please **LOOK** at the file you upload and make sure it is the right one!

| Your submission does not include the **`api`** or **`ui`** packages. |
|---|

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", link #11 on our Canvas front 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 **AlphabetSoup.java** and **AlphabetUtil.java**. You can accomplish this by zipping up the **src** directory of your project as described in the submission HOWTO document. **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.