WORDT
ELGNA

Final Programming Assignment
WordTangle

1. Due Dates and Submission Details
See below for all due dates and submission details. Note there are multiple submission parts.

2. Honor Code
This assignment should be completed individually to maximize learning. It is important that you adhere to all Course Policies.

3. Learning Objectives
This programming assignment is designed to help you learn about (and assess whether you have learned about):

- Creating classes of immutable objects.
- Creating classes of mutable objects.
- Creating and using objects (and classes).
- The difference between (and appropriate use of) static and non-static methods and attributes.
- The difference between the owning object of a method and the objects that are passed as parameters to a method (e.g., the difference between the `String` object `s` and the `String` object `t` in the expression `s.equals(t)`).
- A basic understanding of the reference `null`.

It is also intended to reinforce what you have already learned about conditions and decisions, loops, arrays, unit testing, and various programming patterns.

4. Background

WordTangle is a word game in which the player tries to find a specific word in a rectangular grid of tiles. The player is given a hint and the starting tile. The player must build the word by selecting letters that are to the north, south, east, or west of the previous letter in the word. For example, in the following grid the player was given the 'T' as the starting tile and the hint “Twist together into a confused mass”.

Q	**W**	**E**	**R**	**P**
Y	**U**	**I**	**O**	**P**
S	**T**	**A**	**D**	**F**
K	**X**	**N**	**H**	**G**
E	**L**	**G**	**M**	**K**

The player found the correct answer, “TANGLE” by selecting the tile to the east of the 'T', to the south of the 'A', to the south of the 'N', to the west of the 'G', and to the west of the 'L'.

4.1 Game Play

A game of WordTangle consists of multiple rounds. At the start of the game, the player is presented with a rectangular grid of tiles, each of which contains a letter. The size of the rectangle can vary, the same letter may appear on multiple tiles, and some letters in the alphabet may be omitted.

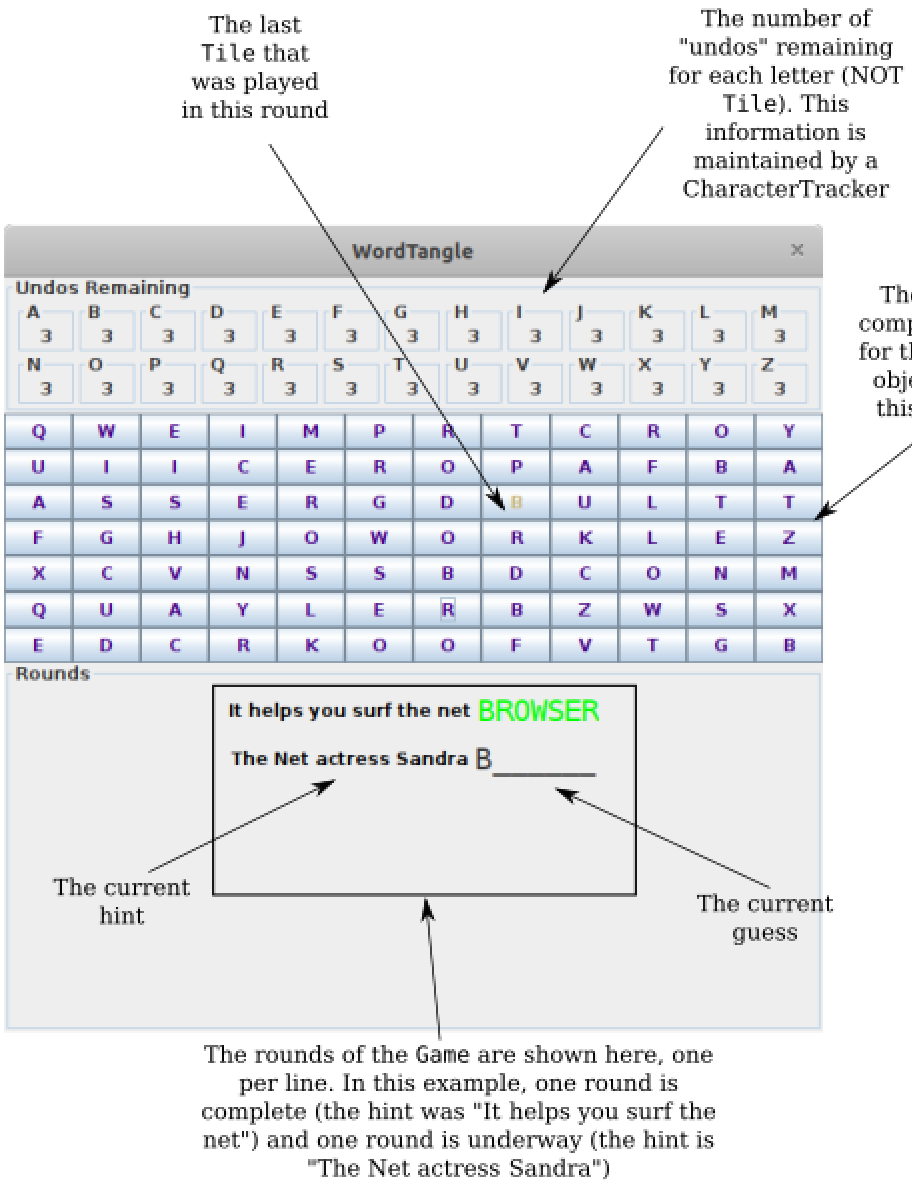
At the start of each round, the player is given a hint, the first tile in the word, and the number of letters/tiles in the word. The player must then select the next tile in the word, which must be to the north, south, east, or west of the first tile. The player must continue in this fashion, moving to the north, south, east or west of the last selected tile, until all of the letters in the word have been selected.

The player is not given any feedback after selecting a tile. Feedback is only given after all of the tiles are selected.

At any point, the player can “undo” the last selected tile. However, there may be limits placed on the number of times each **letter** (not tile) can be “undone”. If there is such a limit, it applies to the entire game (not individual rounds).

4.2 Graphical User Interface

The graphical user interface (GUI) for this implementation of WordTangle has already been created and looks something like the following:


The last Tile that was played in this round
The number of "undos" remaining for each letter (NOT Tile). This information is maintained by a CharacterTracker
WordTangle
Undos Remaining
Rounds
The GUI components for the Tile objects in this Game
The current hint
The current guess
The rounds of the Game are shown here, one per line. In this example, one round is complete (the hint was "It helps you surf the net") and one round is underway (the hint is "The Net actress Sandra")

It helps you surf the net BROWSER

The Net actress Sandra B______

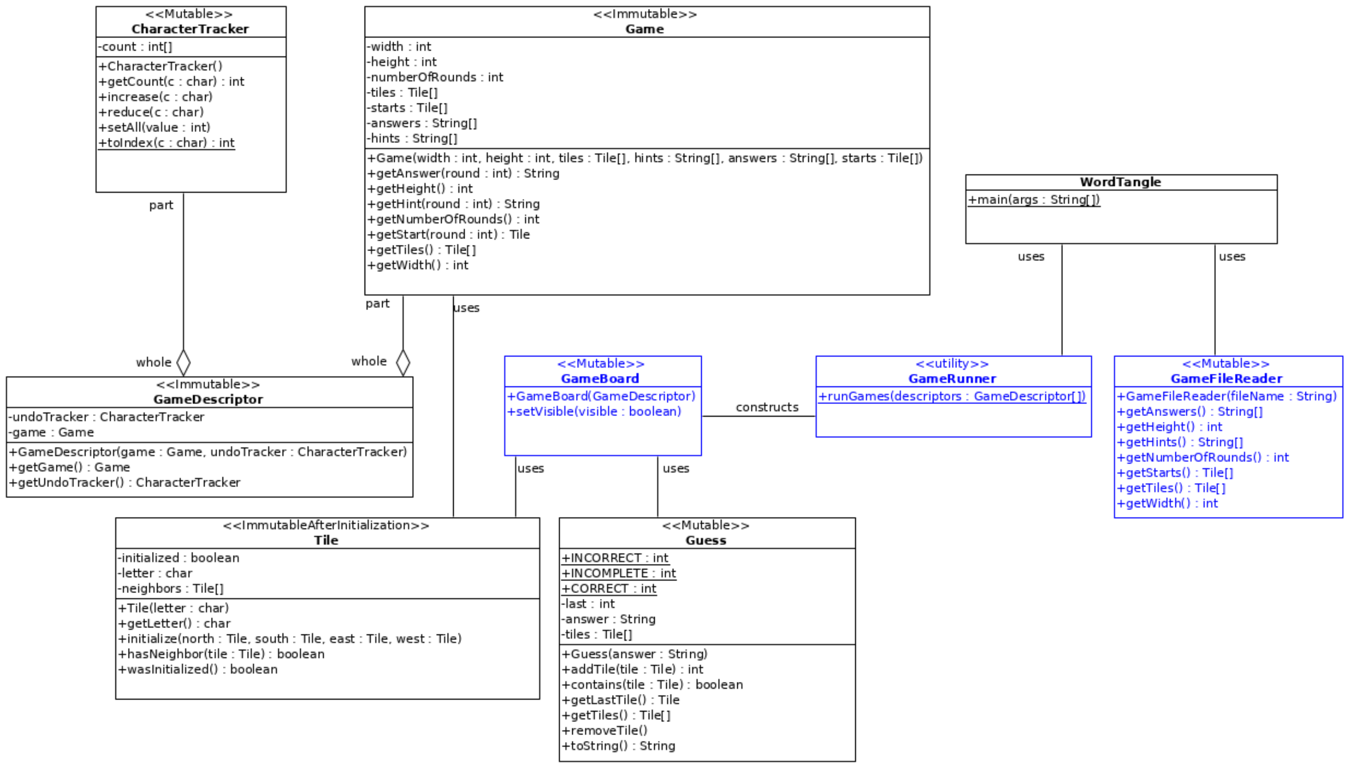
The user selects a tile by clicking on it, which: (1) changes its color from purple to gold, and (2) adds the letter to the current guess (which is displayed to the right of the hint).

The user can “undo” the last tile selected by clicking on it. This will change its color back to purple and remove it from the word after the hint. It will also reduce the number of remaining “undos” for the letter on that tile (unless the player has infinite/unlimited “undos”).

When the player’s guess is as long as the answer, the word after the hint will turn either green (if it is correct) or red (if it is not). If the guess is correct, the next round will be started. If it isn’t the player must “undo” tiles and try again. If the player runs out of “undos” for any letter, the game ends.

5. System Design

The overall design of the system is illustrated in the following UML class diagram (a larger version of which is available for download. In this diagram, the classes that are drawn in blue have already been implemented and the classes that are drawn in black must be implemented by you.



6. Description of Existing Classes
This section contains information over and above the information that is contained in the UML class diagram.

6.1 The `GameFileReader` Class
As the name implies, a `GameFileReader` object can be used to read a single file containing all of the information that describes an individual `Game`. The information will be read when the `GameFileReader` object is constructed. The various accessors can then be used to obtain all of the attributes of the `Game`.

The following attributes are the same for all rounds of the `Game`:

- The `Tile` array (containing all of the `Tile` objects in the rectangular grid);
- The width of the rectangular grid; and
- The height of the rectangular grid.

The following attributes are different for each round:

- The correct answer;
- The hint; and
- The starting `Tile`.

These attributes are stored in *conformal arrays* in which the index corresponds to the round number. So, the size of these arrays corresponds to the number of rounds.

6.2 The `GameRunner` Class
The `GameRunner` class is a utility class that contains a single method, `runGames()`. It is passed an array of `GameDescriptor` objects (discussed below) that is required to play a particular `Game`.

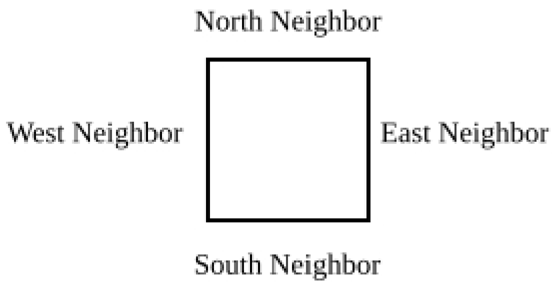
6.2 The `GameBoard` Class
The `GameBoard` class is an encapsulation of the graphical user interface and the game-play logic. You will not need to use this class at all, it is used by the `GameRunner` class.

7. Description of Classes that Need to be Written
You must write and test several classes for this assignment. If a specification states that a class must have a particular method then that method must be public. Classes may contain methods and attributes that are not included in the specifications, but they should be private. All attributes must be private except for the "class constants" in the `Guess` class.

In this section, the classes are presented in an order that makes sense conceptually. The order in which you should probably implement them is discussed later.

Note that, though you must implement and test all of the methods in all of these classes, you may not need to use all of them in the other classes that you write. Some of them may be used by the existing classes.

7.1 The `Tile` Class
The `Tile` class is an encapsulation of a single tile in the rectangular grid of tiles. The attributes of a `Tile` object include its letter and its north, south, east and west neighbors in the rectangular grid, each of which is, itself, a (reference to) a `Tile` object. This can be visualized for a single `Tile` object as follows:



When a `Tile` is constructed, the constructor will only be passed its letter. Its neighbors can only be initialized (by calling its `initialize()` method) after all of the `Tile` objects in the grid have been constructed. So, objects in the `Tile` class are described as "immutable after initialization". This is formalized in the following specification:

S7.1.1 The first time the `initialize()` method is called, the elements of the `neighbors` array must be initialized appropriately and the value of the `initialized` attribute must be set appropriately. Subsequent calls to the `initialize()` method must not change these attributes.

In addition, to this specification and the other specifications contained in the UML class diagram, your implementation must comply with the following specification:

S7.1.2 The `hasNeighbor()` method must return `true` if the parameter named `tile` is either the north, south, east or west neighbor of the owning `Tile`, and must return `false` otherwise.

S7.1.3 The `wasInitialized()` method must return `true` if the method named `initialize()` has been invoked and must return `false` otherwise.

7.2 The `Game` Class

The `Game` class is an encapsulation of the attributes of a single game. `Game` objects must be immutable. That is, the attributes of a `Game` object must not change after the object is constructed.

Note that all of the attributes needed to construct a `Game` object can be obtained from a `GameFileReader`.

In addition to the specifications that are contained in the UML class diagram, your implementation must comply with the following specifications.

S7.2.1 The `getAnswer()` method must return `null` if the parameter named `round` is out of range.

S7.2.2 The `getHint()` method must return `null` if the parameter named `round` is out of range.

S7.2.3 The `getStart()` method must return `null` if the parameter named `round` is out of range.

S7.2.4 The `getStart()` method must return the (reference to) the appropriate element of the `starts` array that is passed to the constructor, not a copy or clone.

S7.2.5 The `getTiles()` method must return the (reference to) the `tiles` array that is passed to the constructor, not a copy or clone.

Internally, rounds are 0-based.

7.3 The `Guess` Class

The `Guess` class is an encapsulation of all of the `Tile` objects in the player's guess. When a `Guess` object is constructed, it is passed the correct answer. `Guess` objects are mutable.

As the player adds a `Tile` to the `Guess`, the `addTile()` method is called. This method assigns the (reference to the) `Tile` object to the first `null` element in the array named `tiles` (if there is one) and increases the attribute named `last`. When the player "undoes" a `Tile`, the `removeTile()` method is called. This method assigns `null` to the the last non-`null` element of the array named `tiles` (if there is one) and decreases the attribute named `last`.

In addition to the specifications that are contained in the UML class diagram, your implementation must comply with the following specifications:

S7.3.1 The `tiles` array must have exactly as many elements as the number of characters in the `answer` attribute, and all "empty" elements must be `null`.

S7.3.2 The `addTile()` method must not add a `Tile` that is already an element of the `tiles` array.

S7.3.3 The `addTile()` method must not add a `Tile` if the `tiles` array is full.

S7.3.4 The `addTile()` method must not add a `Tile` if the `tile` to be added is not a neighbor of the last non-empty element of the `tiles` array.

S7.3.5 The `addTile()` method must return `Guess.INCOMPLETE` if the number of non-null elements of the `tiles` array is less than the length of the `answer` attribute. Otherwise, it must return either `Guess.CORRECT` or `Guess.INCORRECT` as appropriate. This must be the case regardless of whether the `Tile` was added or not.

S7.3.6 The `contains()` method must return `true` if the tiles array contains the parameter named `tile` and must return `false` otherwise.

S7.3.7 The `getLastTile()` method must return the last non-`null` `Tile`. It must return `null` if there are no `Tile` objects in the `Guess`.

S7.3.8 The `getTiles()` method must return all of the `Tile` objects (both `null` and non-`null`), in the correct order. The array it returns must not contain copies or clones of the `Tile` objects that were added.

S7.3.9 The `toString()` method must return a `String` that has the same length as the `answer` attribute. It must contain the characters associated with the non-`null` elements of the `tiles` array (in the appropriate order) followed by as many underscore characters (i.e., `'_'` characters) as there are `null` elements in the `tiles` array. (See the "Hints" section below for a hint on how to implement this method.)

7.4 The `CharacterTracker` Class

The `CharacterTracker` class is not specific to WordTangle; a `CharacterTracker` object can be used to keep count of the number of times a particular (upper-case) character has been "used" in a variety of different applications. `CharacterTracker` objects are mutable.

Because there are 26 different (upper-case) characters in the English alphabet, an *accumulator array* is used (rather than 26 different attributes). This array is named `count`. Each element of count contains the number of times a particular (upper-case) character has been "used". Note that this number can be negative if the `decrease()` method is called more times that the `increase()` method.

In addition to the specifications that are contained in the UML class diagram, your implementation must comply with the following specifications:

S7.4.1 The `count` array must be 0-based and have exactly 26 elements (one for each upper-case letter in the English alphabet).

S7.4.2 The elements in the `count` array must be in alphabetical order (i.e., `'A'` in element `0`, `'B'` in element `1`, …, `'Z'` in element 25.

S7.4.3 The `getCount()` method must return the element of the `count` array that corresponds to the character `c`.

S7.4.4 The `increase()` method must increase the element of the `count` array that corresponds to the character `c` by 1.

S7.4.5 The `reduce()` method must reduce the element of the `count` array that corresponds to the character `c` by 1.

S7.4.6 The `setAll()` method must set all elements of the `count` to the given `value`.

S7.4.7 The static `toIndex()` method must return the index that corresponds to the character `c`. The value returned must be in the interval [0, 25]. (See the "Hints" section below for a hint on how to implement this method.)

In the context of WordTangle, a `CharacterTracker` object is used to keep track of the number of "undos" remaining for each letter.

7.5 The `GameDescriptor` Class

A `GameDescriptor` object contains a `Game` object and a `CharacterTracker` object. In other words, it contains all of the information needed by the `GameRunner` class to start a game.

The `Game` and `CharacterTracker` are kept separate so that the same `Game` can be played with different "undo" rules.

`GameDescriptor` objects are immutable.

7.6 The `WordTangle` Class

The `WordTangle` class is the main class for the application. That is, it contains the entry-point for the application.

The `main()` method will be passed a *segmented* array in which each record contains two fields, the name of a file containing information about a game and the number of “undos” (which is assumed to be the same for all letters). So, for example, the application might be executed as follows:

```
java WordTangle net.txt 3 cs.txt -1
```

It must then use this information to construct a `GameDescriptor` array that has two elements. For element 0 of the array:

- The `Game` must be constructed using a `GameFileReader` that reads the file `net.txt`; and
- The `CharacterTracker` must have all of its elements initialized to `3`.

For element 1 of the array:

- The `Game` must be constructed using a `GameFileReader` that reads the file `cs.txt`; and
- The `CharacterTracker` must have all of its elements initialized to `-1` (which will be used to indicate that the number of “undos” is infinite).

Obviously, after the `GameFileReader` is constructed, you will need to call its various accessors to get the information you need to construct the `Game` object.

In addition to the specifications that are implicit in the discussion above and the UML class diagram, this class must comply with the following specifications:

S7.6.1 The `main()` method must work with any number of records (i.e., with any even number of command line arguments). It need not validate the command line arguments.

S7.6.2 After constructing the array of `GameDesciptor` objects, the `main()` method must call the `runGames()` method in the `GameRunner` class.

8. A Recommended Process

This assignment is complicated enough that you are unlikely to complete it if you do not use a good process. What follows is one such process. It includes an effort estimate (shown in brackets) for each of the major tasks. This is an estimate of how long it *should* take a CS149 student to complete the task (assuming you already understand the material being covered), not how long it *will take you* or how much time you should budget. You must determine how well you understand the material and be able to roughly predict how long it will take you to complete a particular task.

1. Read and understand the product description and specifications (and complete the submission of part A). [1.00 hour]
2. Create a directory/folder for this assignment. [0.05 hours]
3. Download `ExistingClasses.zip` and unzip it in the directory/folder for this assignment. [0.05 hours]
4. Download `DataFiles.zip` and unzip it in the directory/folder for this assignment. [0.05 hours]
5. Download `Stubs.zip` and unzip it into the directory/folder for this assignment. This file contains stubbed-out versions of all of the classes (i.e., initial encapsulations of all of the classes that contain the public “constants” and public methods with empty bodies, except for return statements where needed).

[0.05 hours]

6. Complete the `GameDescriptor` class. [0.25 hours]
   - 6.1. Write the constructor, `getGame()` and `getUndoTracker()` methods.
   - 6.2. Test (and debug, if necessary) these methods.
7. Complete the `Tile` class. [0.75 hours]
   - 7.1. Write the constructor and `getLetter()` method.
   - 7.2. Test (and debug, if necesssary) these methods.
   - 7.3. Write the `initialize()` and `wasInitialized()` methods.
   - 7.4. Test (and debug, if necesssary) these methods.
8. Complete the `Game` class. [1.00 hour]
   - 8.1. Write the constructor, `getHeight()`, `getWidth()`, `getNumberOfRounds()` and `getTiles()` methods.
   - 8.2. Test (and debug, if necessary) these methods.
   - 8.3. Implement the `getAnswer()` method.
   - 8.4. Test (and debug, if necessary) this method.
   - 8.5. Implement the `getHint()` and `getStart()` methods.
   - 8.6. Test (and debug, if necessary) these methods.
9. Complete the `Guess` class. [4.00 hours]
   - 9.1. Write the constructor and the `getTiles()` method.
   - 9.2. Test (and debug, if necessary) these methods.
   - 9.3. Write the `addTile()` and `getLastTile()` methods.
   - 9.4. Write the `removeTile()` method.
   - 9.5. Test (and debug, if necessary) this method. (See the “Hints” section below for hints on how to test the methods in this class.)
   - 9.6. Write the `contains()` method.
   - 9.7. Test (and debug, if necessary) this method.
   - 9.8. Write the `toString()` method. (See the “Hints” section below for a hint on how to implement this method.)
   - 9.9. Test (and debug, if necessary) this method.

To make it easier for you to work on this assignment in pieces, the `GameBoard` class has been written in such a way that you do not need to have a working implementation of the `CharacterTracker` class to use it. So, at this point you should probably continue as follows:

10. Complete v1 of the `WordTangle` class. For this version, use `null` for the `CharacterTracker` in

each `GameDescriptor` that it constructs. [1.50 hours]

11. Perform integration testing by running the `WordTangle` application using the supplied game files. [0.50 hours]

12. Complete the `CharacterTracker` class. [2.00 hours]

    12.1. Write the constructor and `getCount()` methods.

    12.2. Test (and debug, if necessary) these methods.

    12.3. Write the `toIndex()` method. (See the "Hints" section below for a hint on how to implement this method.)

    12.4. Test (and debug, if necessary) this method.

    12.5. Write the `setAll()` method.

    12.6. Test (and debug, if necessary) this method.

    12.7. Write the `increase()` and `reduce()` methods.

    12.8. Test (and debug, if necessary) these methods.

13. Complete v2 of the `WordTangle` class (i.e., it must now construct `GameDescriptor` objects that contain actual `CharacterTracker` objects). [0.50 hours]

14. Perform system testing by running the `WordTangle` application using the supplied game files. [1.00 hours]

15. Submit a file named `project.zip` containing just your implementations of `CharacterTracker.java`, `Game.java`, `GameDescriptor.java`, `Guess.java`, `Tile.java` and `WordTangle.java` using Autolab (i.e., complete the submission for B). [0.05 hours]

9. Hints

9.1 Implementing the `toString()` method in the `Guess` Class

There are many ways to implement the `toString()` method in the `Guess` class, but it is probably easiest to use a `String` accumulator (i.e., loop over the elements in the `Tile[]` and concatenate either a letter or an underscore character to the `String` accumulator).

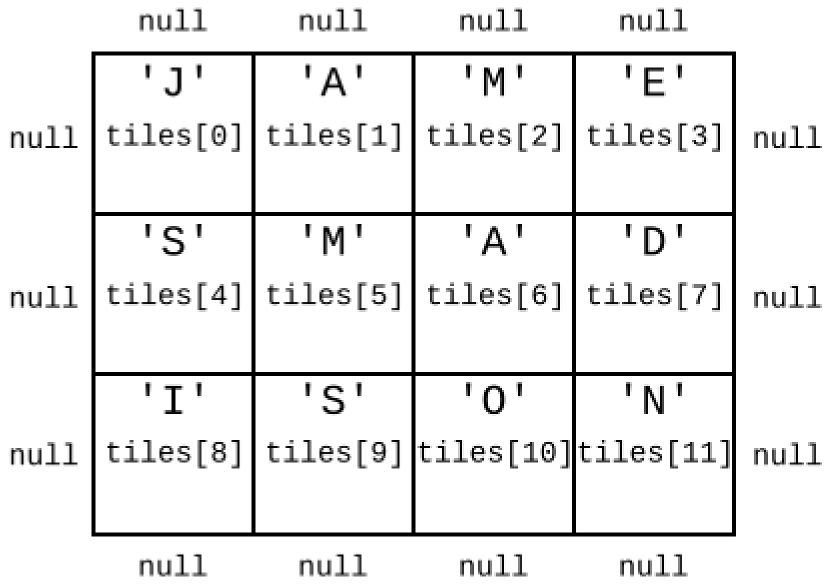
9.2 Testing the `Guess` Class

The `Guess` class is difficult to test because it requires a `Tile[]` in which each `Tile` object is properly initialized. Fortunately, a `GameReader` object can do this for you. For example, if you execute the following code:

```
GameFileReader reader;
Tile[] tiles;
```

```
reader = new GameFileReader("jmu.txt");
tiles = reader.getTiles();
```

then the array named `tiles` will contain `Tile` objects that can be visualized as follows:



You could then test the `addTile()` method in your `Guess` class as follows:

```
Guess guess;
int status;
guess = new Guess("MAD");

status = guess.addTile(tiles[5]);
// Check to make sure status is Guess.INCOMPLETE

status = guess.addTile(tiles[6]);
// Check to make sure status is Guess.INCOMPLETE

status = guess.addTile(tiles[7]);
// Check to make sure status is Guess.CORRECT
```

Of course, this is not a complete test suite. You should do the same kind of thing with:

- A `Guess` that is ultimately incorrect,
- An attempt to add a `Tile` that is already in the `Guess`,
- An attempt to add a `Tile` that is not a neighbor of the last tile that was added,
- A removal of a `Tile`,
- An addition of a `Tile` after a removal of a `Tile`,
- etc.

The example above should get you started.

9.3 Implementing the `toIndex()` Method in the `CharacterTracker` Class

The `count` array in the `CharacterTracker` class is a both a lookup array and an accumulator array. The easiest way to do the lookup (i.e., to create a correspondence between `char` values and indexes) is to use the ASCII value of the `char`. The upper-case letters A-Z have consecutive ASCII values, and the ASCII value of upper-case A is `65`. In Java, you can convert a `char` to an `int` by typecasting it. For example, the following fragment:

```
char c;
int i;
c = 'A';
i = (int)c;
```

will assign the value `65` to `i`.

10. Submission

The submission for this assignment is divided into two parts that should be completed in order.

Part A: Understanding the Problem: due 12/2 11:00 pm

To complete Part A you should first carefully read the product description and specifications. Once you feel confident that you have a good grasp of the assignment, log into Canvas and complete Part A. **YOU MUST ANSWER ALL QUESTIONS CORRECTLY (earn 100%) TO GET ANY CREDIT FOR PART A**. You may take the quiz as many times as necessary but if you do not complete it on time, you will receive zero credit.

Part B: Java Implementation: due 12/11 11:00 pm

-15% before 12/12 11:00 pm
-30% before 12/13 11:00 pm
Not accepted after 12/13 11:00 pm

Part B is an Autolab submission. You must submit a `.zip` file containing only the following files: `CharacterTracker.java`, `Game.java`, `GameDescriptor.java`, `Guess.java`, `Tile.java`, and `WordTangle.java`. Your `.zip` file must not contain any source files you created for testing purposes, any `.class` files, and/or any data files.

Note, since this is a difficult assignment, **there is no limit on the number of Autolab submissions**. However, that does not mean that you shouldn't try to test your code “locally” before submitting it.

11. Grading

Part A accounts for 10 points of your grade on this assignment and Part B accounts for 90 points. Part A will be graded by Canvas and part B will be graded by Autolab and the Professor.

11.1 Autolab Grading

Your code must compile (in Autolab, this will be indicated in the section on “Does your code compile?”), and all class names and method signatures comply with the specifications (in Autolab, this will be indicated in the section on “Do your class names, method signatures, etc. comply with the specifications?”) for you to receive any points on this assignment.

Checkstyle	10 points (No Partial Credit)
Correctness of the `GameDescriptor` class	8 points (Partial Credit Possible)
Correctness of the `Tile` class	12 points (Partial Credit Possible)
Correctness of the `Game` class	12 points (Partial Credit Possible)
Correctness of the `Guess` class	20 points (Partial Credit Possible)
Correctness of the `CharacterTracker` class	12 points (Partial Credit Possible)
Correctness of the complete product	16 points (Partial Credit Possible)

Note that this point allocation is not based solely on the difficulty of the class or the effort estimate for that class. For example, completing the `Guess` class will probably require 16 times more effort than completing the `GameDescriptor` class but it is only allocated 2.5 times as many points. This is so that you have an opportunity to earn points on this assignment even if you are unable to fully complete the difficult tasks.

Note also that, though you can earn partial credit, your code will not compile on Autolab if you don't submit all of the classes (with all of the required methods). This is why you were provided with stubs of all of the classes.

11.2 Manual Grading

After the due date, the Professor will manually review your code. At this time, points may be deducted for inelegant code, inappropriate variable names, bad comments, etc.