ASSIGNMENT 4

COMP-202, Winter 2019

Due: Friday, March 29th, (23:59)

Please read the entire PDF before starting. You must do this assignment individually.

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment, in some cases through automated tests. While these tests will never be used to determine your entire grade, they speed up the process significantly, which allows the TAs to provide better feedback and not waste time on administrative details. Plus, if the TA is in a good mood while he or she is grading, then that increases the chance of them giving out partial marks. :)

Up to 30% can be removed for bad indentation of your code as well as omitting comments, or poor coding structure.

To get full marks, you must:

- Follow all directions below
  - In particular, make sure that all classes and method names are **spelled and capitalized exactly** as described in this document. Otherwise, you will receive a **50% penalty**.
- Make sure that your code compiles
  - **Non-compiling code will receive a 0**.
- Write your name and student ID as a comment in all .java files you hand in
- Indent your code properly
- Name your variables appropriately
  - The purpose of each variable should be obvious from the name
- Comment your work
  - A comment every line is not needed, but there should be enough comments to fully understand your program

Part 1 (0 points): Warm-up

Do **NOT** *submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that you consult the TAs during their office hours; they can help you and work with you through the warm-up questions. You are responsible for knowing all of the material in these questions.*

Warm-up Question 1 (0 points)
Write a method `longestSubArray` that takes as input an array of integer arrays (i.e. a multi-dimensional array) and returns the *length* of the longest sub-array. For example, if the input is: `int[][] arr= {{1,2,1},{8,6}};` then `longestSubArray` should return 3.

Warm-up Question 2 (0 points)
Write a method `subArraySame` that takes as input an array of integer arrays (i.e. a multi-dimensional array) and checks if all of the numbers in each 'sub-array' are the same. For example, if the input is: `int[][] arr= {{1,1,1},{6,6}};` then `subArraySame` should return true and if the input is: `int[][] arr= {{1,6,1},{6,6}};` then `subArraySame` should return false.

Warm-up Question 3 (0 points)
Write a method `largestAverage` that takes as input an array of double arrays (i.e. a multi-dimensional array) and returns the double array with the largest *average* value. For example, if the input is: `double[][] arr= {{1.5,2.3,5.7},{12.5,-50.25}};` then `largestAverage` should return the array $\{1.5, 2.3, 5.7\}$ (as the average value is 3.17).

Warm-up Question 4 (0 points)
Write a class describing a `Cat` object. A cat has the following `attributes`: a name (String), a breed (String), an age (int) and a mood (String). The mood of a cat can be one of the following: `sleepy, hungry, angry, happy, crazy`. The cat `constructor` takes as input a String and sets that value to be the breed. The `Cat` class also contains a method called `talk()`. This method takes no input and returns nothing. Depending on the mood of the cat, it prints something different. If the cat's mood is `sleepy`, it prints *meow*. If the mood is `hungry`, it prints *RAWR!*. If the cat is `angry`, it prints *hsssss*. If the cat is `happy` it prints *purrrr*. If the cat is `crazy`, it prints a String of 10 gibberish characters (e.g. raseagafqa).

The cat `attributes` are all **private**. Each one has a corresponding **public** get method (ie: `getName()`, `getMood()`, etc.) which returns the value of the `attribute`. All but the `breed` also have a **public** set method (ie: `setName()`, `setMood()`, etc.) which takes as input a value of the type of the attribute and sets the attribute to that value. Be sure that only valid mood sets are permitted. (ie, a cat's mood can only be one of five things). There is no setBreed() method because the breed of a cat is set at birth and cannot change.

Test your class in another file which contains only a main method. Test all methods to make sure they work as expected.

Warm-up Question 5 (0 points)
Using the `Cat` type defined in the previous question, create a `Cat[]` of size 5. Create 5 `Cat` objects and put them all into the array. Then use a loop to have all the `Cat` objects `meow`.

Warm-up Question 6 (0 points)

Write a class `Vector`. A `Vector` should consist of three `private` properties of type double: x, y, and z. You should add to your class a constructor which takes as input 3 doubles. These doubles should be assigned to x, y, and z. You should then write methods `getX()`, `getY()`, `getZ()`, `setX()`, `setY()`, and `setZ()` which allow you to get and set the values of the vector. Should this method be static or non-static?

Warm-up Question 7 (0 points)

Add to your `Vector` class a method `calculateMagnitude` which returns a double representing the magnitude of the vector. Should this method be static or non-static? The magnitude can be computed by taking:

$$magnitude = \sqrt{x^2 + y^2 + z^2}$$

Warm-up Question 8 (0 points)

Write a method `scalarMultiply` which takes as input a `double[]`, and a `double scale`, and returns `void`. The method should modify the input array by multiplying each value in the array by `scale`. Should this method be static or non-static?

Warm-up Question 9 (0 points)

Write a method `deleteElement` which takes as input an `int[]` and an `int target` and deletes all occurrences of `target` from the array. By "delete" we mean create a new array (of smaller size) which has the same values as the old array but without any occurrences of `target`. The method should return the new `int[]`. Question to consider: Why is it that we have to return an array and can't simply change the input parameter array like in the previous question? Should these methods be static or non-static?

Warm-up Question 10 (0 points)

Write the same method, except this time it should take as input a `String[]` and a `String`. What is different about this than the previous method? (Hint: Remember that `String` is a reference type.)

Part 2

The questions in this part of the assignment will be graded.

Question 1: Game of Life (60 points)

For this question, you will have to write a Java program that implements a simple version of Conway's Game Of Life (`https://en.wikipedia.org/wiki/Conway\%27s_Game_of_Life`)

This is a zero-player game, which means that it is completely determined by the initial state provided. For our purpose, we will represent the universe of the Game of Life as a finite rectangular two-dimensional array. Each element denotes a cell in the universe: if the cell is alive the value of the element is 1, if the cell is dead the value of the element is 0. Whether a cell is alive or dead in the next generation of the universe depends on its relation with its neighboring cells (the cells that are horizontally, vertically, or diagonally adjacent to it). The next generation of cell is determined by the following rules:

- Any live cell with fewer than two live neighbors dies, as if caused by under population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if caused by overpopulation.
- Any dead cell with exactly three live neighbors becomes a live cell, as if caused by reproduction.

In this question, you will implement a method that will allow you to display a given number of generations of a specified universe.

To complete this task, you will need to implement all the methods listed below. All the code for this question must be placed in a file named `GameOfLife.java`. Note that you are free to write more methods if they help the design or readability of your code.

Note, that you are not required to write a `main` method for this question. Use the `main` method to test all your methods while writing the program. When you hand in your assignment though, make sure that your `main` method is empty.

1a) Method to check whether a 2D array represents a valid universe

Write a method `isValidUniverse` which takes as input a two-dimensional array of integers. The method should return `true` if such array is a valid representation of a universe, `false` otherwise. For the purpose of this assignment, we consider a two-dimensional array to be a valid representation of a universe if it is rectangular (i.e. all its sub-arrays are of the same size) and all the elements of its sub-arrays are either 0s or 1s.

For example, consider the following two-dimensional arrays:

```
int[][] a = {{0,0}, {1,0}, {1,0}};
int[][] b = {{0,0,0}, {1,1}, {0}};
int[][] c = {{1,2}, {3,4}};
int[][] d = {{1,1}, {1,1}};
```

Then:

- `isValidUniverse(a)` returns `true`.
- `isValidUniverse(b)` returns `false`.
- `isValidUniverse(c)` returns `false`.
- `isValidUniverse(d)` returns `true`.

1b) Method to display the universe

Write a method `displayUniverse()` which takes a two-dimensional array of integers as input. In this method you can assume that the array is a valid representation of a universe (as described above). The method should display the universe using the star character (`'*'`) for alive cells, and the whitespace character (`' '`) for dead cells. Your method should also display a box (as shown below) surrounding the actual cells of the universe.

For example, consider the following arrays:

```
int[][] block = {{0,0,0,0}, {0,1,1,0}, {0,1,1,0}, {0,0,0,0}};
int[][] tub = {{0,0,0,0,0}, {0,0,1,0,0}, {0,1,0,1,0}, {0,0,1,0,0}, {0,0,0,0,0}};
int[][] toad = {{0,0,0,0,0,0}, {0,0,1,1,1,0}, {0,1,1,1,0,0}, {0,0,0,0,0,0}};
```

Then:

- `displayUniverse(block)` prints

```
+----+
|    |
| ** |
| ** |
|    |
+----+
```

- `displayUniverse(tub)` prints

```
+-----+
|     |
|  *  |
| * * |
|  *  |
|     |
+-----+
```

- `displayUniverse(toad)` prints

```
+------+
|      |
|  *** |
| ***  |
|      |
+------+
```

1c) Method to get the cell from the next generation

Write a method `getNextGenCell` that takes as input a two-dimensional array of integers representing a valid universe, as well as two integers `x` and `y` indicating the location of a specific cell. For example, if `x=2` and `y=1`, the cell analyzed is the one represented by the second element of the third sub-array. The method returns `1` if such cell will be alive in the next generation of the given universe, `0` otherwise. Remember that the next generation of cells is determined by the following rules:

- Any live cell with fewer than two live neighbors dies, as if caused by under population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if caused by overpopulation.
- Any dead cell with exactly three live neighbors becomes a live cell, as if caused by reproduction.

For example, consider the following arrays:

```
int[][] beehive = {{0,0,0,0,0,0}, {0,0,1,1,0,0}, {0,1,0,0,1,0}, {0,0,1,1,0,0},
                   {0,0,0,0,0,0}};
int[][] toad = {{0,0,0,0,0,0}, {0,0,1,1,1,0}, {0,1,1,1,0,0}, {0,0,0,0,0,0}};
```

Then:

- `getNextGenCell(beehive, 1, 3)` returns 1, while `getNextGenCell(beehive, 3, 1)` returns 0.
- `getNextGenCell(toad, 0, 3)` returns 1, while `getNextGenCell(toad, 2, 3)` returns 0.

Note that this method *must not* modify the input array in any way.

1d) Method to create the next generation universe

Write a method `getNextGenUniverse` that takes as input a two-dimensional array of integers representing a valid universe and returns a two-dimensional array of integers (with equal dimensions) representing the universe in its next generation. To get full marks, your method must call `getNextGenCell()`.

For example, consider the following arrays:

```
int[][] tub = {{0,0,0,0,0}, {0,0,1,0,0}, {0,1,0,1,0}, {0,0,1,0,0}, {0,0,0,0,0}};
int[][] toad = {{0,0,0,0,0,0}, {0,0,1,1,1,0}, {0,1,1,1,0,0}, {0,0,0,0,0,0}};
int[][] pentadec = {{0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0},
 {0,0,0,0,1,0,0,0,0,0}, {0,0,0,0,1,0,0,0,0,0}, {0,0,0,1,0,1,0,0,0,0}, {0,0,0,0,1,0,0,0,0,0},
 {0,0,0,0,1,0,0,0,0,0}, {0,0,0,0,1,0,0,0,0,0}, {0,0,0,0,1,0,0,0,0,0}, {0,0,0,1,0,1,0,0,0,0},
 {0,0,0,0,1,0,0,0,0,0}, {0,0,0,0,1,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0},
 {0,0,0,0,0,0,0,0,0,0}};
```

Then:

- `getNextGenUniverse(tub)` returns the following array:

  ```
  {{0,0,0,0,0}, {0,0,1,0,0}, {0,1,0,1,0}, {0,0,1,0,0}, {0,0,0,0,0}}
  ```

- `getNextGenUniverse(toad)` returns the following array:

  ```
  {{0,0,0,1,0,0}, {0,1,0,0,1,0}, {0,1,0,0,1,0}, {0,0,1,0,0,0}}
  ```

- `getNextGenUniverse(pentadec)` returns the following array:

  ```
  {{0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0},
  {0,0,0,1,1,1,0,0,0,0}, {0,0,0,1,0,1,0,0,0,0}, {0,0,0,1,1,1,0,0,0,0}, {0,0,0,1,1,1,0,0,0,0},
  {0,0,0,1,1,1,0,0,0,0}, {0,0,0,1,1,1,0,0,0,0}, {0,0,0,1,0,1,0,0,0,0}, {0,0,0,1,1,1,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0}}
  ```

Note that this method *must not* modify the input array in any way.

1e) Method to display a given number of generations

Finally, create a method `simulateNGenerations`. This method takes a two-dimensional array of integers and one integer `n` as input. The method should first check if the input array represents a valid universe. If this is not the case, the method should throw an `IllegalArgumentException` with an appropriate message. On the other hand, if the array is a valid representation of a universe, then the method should display the original seed (i.e. the universe as it has been input) as well as the next `n` generations of such universe. Note that, to get full marks, your method must use the methods described above (`isValidUniverse()`, `displayUniverse()`, and `getNextGenUniverse()`).

For example, consider the following arrays:

```
int[][] tub = {{0,0,0,0,0}, {0,0,1,0,0}, {0,1,0,1,0}, {0,0,1,0,0}, {0,0,0,0,0}};
int[][] toad = {{0,0,0,0,0,0}, {0,0,1,1,1,0}, {0,1,1,1,0,0}, {0,0,0,0,0,0}};
int[][] pentadec = {{0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0},
 {0,0,0,0,1,0,0,0,0,0}, {0,0,0,0,1,0,0,0,0,0}, {0,0,0,1,0,1,0,0,0,0}, {0,0,0,0,1,0,0,0,0,0},
 {0,0,0,0,1,0,0,0,0,0}, {0,0,0,0,1,0,0,0,0,0}, {0,0,0,0,1,0,0,0,0,0}, {0,0,0,1,0,1,0,0,0,0},
 {0,0,0,0,1,0,0,0,0,0}, {0,0,0,0,1,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0,0,0},
 {0,0,0,0,0,0,0,0,0,0}};
```

Then:

- `simulateNGenerations(tub,2)` prints

```
Original seed
+-----+
|     |
|  *  |
| * * |
|  *  |
|     |
+-----+
Generation 1
+-----+
|     |
|  *  |
| * * |
|  *  |
|     |
+-----+
Generation 2
+-----+
|     |
|  *  |
| * * |
|  *  |
|     |
+-----+
```

- `simulateNGenerations(toad, 3)` prints

```
Original seed
+------+
|      |
|  *** |
| ***  |
|      |
+------+
Generation 1
+------+
|   *  |
| *  * |
| *  * |
|  *   |
+------+
Generation 2
+------+
|      |
|  *** |
| ***  |
|      |
+------+
Generation 3
+------+
|   *  |
| *  * |
| *  * |
|  *   |
+------+
```

- `simulateNGenerations(pentadec, 9)` prints (all in one column of course)

```
Original seed
+---------+
|         |
|         |
|         |
|    *    |
|    *    |
|   * *   |
|    *    |
|    *    |
|    *    |
|    *    |
|   * *   |
|    *    |
|    *    |
|         |
|         |
|         |
+---------+

Generation 1
+---------+
|         |
|         |
|         |
|         |
|   ***   |
|   * *   |
|   ***   |
|   ***   |
|   ***   |
|   ***   |
|   * *   |
|   ***   |
|         |
|         |
|         |
|         |
+---------+

Generation 2
+---------+
|         |
|         |
|         |
|    *    |
|   * *   |
|  *   *  |
|  *   *  |
|  *   *  |
|  *   *  |
|  *   *  |
|  *   *  |
|   * *   |
|    *    |
|         |
|         |
|         |
+---------+

Generation 3
+---------+
|         |
|         |
|         |
|    *    |
|   ***   |
|  ** **  |
| *** *** |
| *** *** |
| *** *** |
| *** *** |
|  ** **  |
|   ***   |
|    *    |
|         |
|         |
|         |
+---------+

Generation 4
+---------+
|         |
|         |
|         |
|   ***   |
|  *   *  |
| *     * |
|         |
|*       *|
|*       *|
|         |
| *     * |
|  *   *  |
|   ***   |
|         |
|         |
|         |
+---------+

Generation 5
+---------+
|         |
|         |
|    *    |
|   ***   |
|  *****  |
|         |
|         |
|         |
|         |
|         |
|         |
|  *****  |
|   ***   |
|    *    |
|         |
|         |
+---------+

Generation 6
+---------+
|         |
|         |
|   ***   |
|  *   *  |
|  *   *  |
|   ***   |
|         |
|         |
|         |
|         |
|   ***   |
|  *   *  |
|  *   *  |
|   ***   |
|         |
|         |
+---------+

Generation 7
+---------+
|         |
|    *    |
|   ***   |
|  * * *  |
|  * * *  |
|   ***   |
|    *    |
|         |
|         |
|    *    |
|   ***   |
|  * * *  |
|  * * *  |
|   ***   |
|    *    |
|         |
+---------+

Generation 8
+---------+
|         |
|   ***   |
|         |
|  *   *  |
|  *   *  |
|         |
|   ***   |
|         |
|         |
|   ***   |
|         |
|  *   *  |
|  *   *  |
|         |
|   ***   |
|         |
+---------+

Generation 9
+---------+
|    *    |
|    *    |
|   ***   |
|         |
|         |
|   ***   |
|    *    |
|    *    |
|    *    |
|    *    |
|   ***   |
|         |
|         |
|   ***   |
|    *    |
|    *    |
+---------+
```

Question 2: Blackjack (40 points)

For this question, you will write two classes, which you can then use to simulate a game of Blackjack. We want to represent a standard deck of playing cards which consists of 52 Cards in each of the 4 suits: Spades, Hearts, Diamonds, and Clubs. Each suit contains 13 cards: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King. For this assignment, we will represent the value of a card using an int which has a value between 1 and 13.

Note that in addition to the required methods below, you are free to add as many other `private` methods as you want (**no** additional `public` method is allowed).

(a) (8 points) Write a class `Card`. A Card has the following `private` attributes:

- A `int` representing its value
- A `String` representing its suit

The Card class also contains the following `public` methods:

- A constructor that takes as input the value and the suit of the card and uses them to initialize the corresponding attributes. This constructor should throw an `IllegalArgumentException` if the inputs do not represent a valid card. A card is considered to be valid if it has a value between 1 and 13 (both included) and if it has one of the following suits: Spades, Hearts, Diamonds, and Clubs. Note, that your constructor should accept as valid both `"Spades"`, `"spades"`, or `"SPADES"` for instance.
- `getValue()` which takes no inputs and returns the value of the card as a number between 1 and 13.
- `getSuit()` which takes no inputs and returns the suit of the card as a String.

(b) (32 points) Write a class `Deck`. A Deck has the following `private` attributes:

- A array of `Card` representing the playing cards in the deck.
- An `int` representing the number of cards left in the deck.
- A `static Random` numberGenerator.

Initialize the `Random` numberGenerator "in place" (i.e. on the same line as its declaration) with a reference to a `Random` object created with the seed 123.

The Deck class should also have the following `public` methods:

- A constructor that takes no input. The constructor initializes the two non-static attributes in order for them to represent a standard deck or cards. Note that this means that the array of Cards should be initialized with an array of 52 elements containing all 52 possible cards belonging to a standard deck. You must use at least 1 loop to accomplish this (that is, you cannot write 52 statements to assign all possible values). *Hint: If you create a String array of size 4 with all the possible suits value in it, you can easily use two nested loops to initialize the array of cards.*
- `getNumOfCards()` which takes no inputs and returns the number of cards left in the deck.
- `getCards()` which takes no inputs and returns an array of Cards with all the cards that are left in the deck. Note that no one should be able to use `getCards()` and, for instance, made this deck a trick deck. Hint: review what we have learned about private attributes that are mutable reference types.

In addition to the above methods, we need to add `public` methods that allow us to "play" with the deck. The general idea is to never change the size of the array of cards, but instead use the attribute representing the number of cards left in the deck to keep track of which cards are still in the deck after we have played around with it. Note that to write a more efficient code, we will be thinking of the cards as listed in the reverse order as they appear in the deck from a player's point of view. This means that the last card in the deck (the one *at the bottom of the deck*) appears in the first position of our array, while the card that is *on top of the deck* will appear in the last position of the array which represents a card still in the deck (note that this in not necessarily the last position of the array). For instance, if we look at a fresh deck with 52 cards, then the card at the bottom of the deck appears in position 0, while the card on top of the deck appears in position 51. On the other hand, if we look at a deck with 26 cards left, then the card at the bottom of the deck appears in position 0, while **the card of the top of the deck appears in position 25**.

Here are the additional `public` methods to be added to class Deck:

- `showCards()` which takes no inputs and displays all the cards left in the deck (from top to bottom). For example, if the deck as 4 cards left, you might see displayed something similar to the following:

  ```
  6 of clubs, 3 of hearts, 1 of spades, 13 of hearts.
  ```

  Note that you should not print words such as "king" or "queen", you can simply display them as 13 or 12. IMPORTANT: Once again, the first card displayed should be the one *on top of the deck*, while the last one is the one *at the bottom of the deck*. See previous paragraph for a detailed explanation.

- `shuffle()` which takes no inputs and shuffles the cards *left* in the deck. You can do this by repeatedly generating 2 random indices (from 0 to the number of cards left in the deck) and swap the content of the array of cards at those two position. Repeat this operation for 1000 times.

- `deal()` which takes no inputs and returns the Card which appears *on top of the deck*. **If there are no more cards left in the deck, then the method should return `null`.** Note that if a card was dealt, then after this operation the deck will contain one less card and the card returned by the method should not appear anymore as one of the cards left in the deck.

- `pickACard()` which takes an int as input indicating the position of a card in the deck *from the player's point of view.* If the deck does not have enough cards left, then the method returns `null`. Otherwise, the method returns the Card in that position. Please note once again, that the cards in the array are stored in the reverse order compare to how they appear in the deck from a player's point of view. Note that after this operation the deck will contain one less card and the card returned by the method should not appear anymore as one of the cards left in the deck.

- `restockDeck()` which takes no inputs and resets the deck back to its original form. That is, the deck will then contain all 52 cards belonging to a standard deck.

In order to give you some examples of how your methods should word, I need to start by showing you what will be displayed if I call `showCards()` on a "freshly" made deck. This of course will depend on how the constructor initializes the attributes. On my program, if I create a deck and I call `showCards()` on it, the following will be displayed all on one line:

```
1 of Clubs, 2 of Clubs, 3 of Clubs, 4 of Clubs, 5 of Clubs, 6 of Clubs, 7 of Clubs,
8 of Clubs, 9 of Clubs, 10 of Clubs, 11 of Clubs, 12 of Clubs, 13 of Clubs,
1 of Diamonds, 2 of Diamonds, 3 of Diamonds, 4 of Diamonds, 5 of Diamonds,
6 of Diamonds, 7 of Diamonds, 8 of Diamonds, 9 of Diamonds, 10 of Diamonds,
11 of Diamonds, 12 of Diamonds, 13 of Diamonds, 1 of Hearts, 2 of Hearts,
3 of Hearts, 4 of Hearts, 5 of Hearts, 6 of Hearts, 7 of Hearts, 8 of Hearts,
9 of Hearts, 10 of Hearts, 11 of Hearts, 12 of Hearts, 13 of Hearts, 1 of Spades,
2 of Spades, 3 of Spades, 4 of Spades, 5 of Spades, 6 of Spades, 7 of Spades,
8 of Spades, 9 of Spades, 10 of Spades, 11 of Spades, 12 of Spades, 13 of Spades.
```

Make sure to initialize the attributes as I did, so that you'll be able to check that your methods work by matching the examples below.

If I then call `shuffle()` followed by `showCards()`, the following will be displayed:

```
13 of Diamonds, 2 of Clubs, 12 of Diamonds, 12 of Hearts, 6 of Clubs,
1 of Diamonds, 12 of Clubs, 10 of Hearts, 9 of Diamonds, 9 of Spades,
4 of Hearts, 2 of Diamonds, 8 of Hearts, 5 of Diamonds, 11 of Clubs,
8 of Diamonds, 7 of Diamonds, 11 of Hearts, 3 of Hearts, 1 of Spades,
6 of Diamonds, 7 of Clubs, 5 of Spades, 11 of Spades, 10 of Diamonds,
2 of Hearts, 4 of Spades, 3 of Diamonds, 6 of Hearts, 4 of Diamonds,
7 of Hearts, 11 of Diamonds, 6 of Spades, 1 of Clubs, 3 of Clubs, 3 of Spades,
4 of Clubs, 5 of Hearts, 12 of Spades, 10 of Spades, 8 of Spades, 10 of Clubs,
13 of Spades, 5 of Clubs, 1 of Hearts, 9 of Hearts, 9 of Clubs, 7 of Spades,
2 of Spades, 13 of Hearts, 8 of Clubs, 13 of Clubs.
```

If I then call `deal()` I receive the card "13 of Diamonds". If I then call `pickACard(11)` I receive the card "8 of Hearts". If I then call `getNumOfCards()` I receive `50`, and if I call `showCards()` I see the following displayed:

```
2 of Clubs, 12 of Diamonds, 12 of Hearts, 6 of Clubs, 1 of Diamonds, 12 of Clubs,
10 of Hearts, 9 of Diamonds, 9 of Spades, 4 of Hearts, 2 of Diamonds,
5 of Diamonds, 11 of Clubs, 8 of Diamonds, 7 of Diamonds, 11 of Hearts,
3 of Hearts, 1 of Spades, 6 of Diamonds, 7 of Clubs, 5 of Spades, 11 of Spades,
10 of Diamonds, 2 of Hearts, 4 of Spades, 3 of Diamonds, 6 of Hearts,
4 of Diamonds, 7 of Hearts, 11 of Diamonds, 6 of Spades, 1 of Clubs,
3 of Clubs, 3 of Spades, 4 of Clubs, 5 of Hearts, 12 of Spades, 10 of Spades,
8 of Spades, 10 of Clubs, 13 of Spades, 5 of Clubs, 1 of Hearts, 9 of Hearts,
9 of Clubs, 7 of Spades, 2 of Spades, 13 of Hearts, 8 of Clubs, 13 of Clubs.
```

If I then call `shuffle()` followed by `showCards()` I see the following displayed:

```
12 of Clubs, 5 of Diamonds, 7 of Diamonds, 13 of Spades, 4 of Diamonds,
13 of Hearts, 2 of Hearts, 4 of Spades, 12 of Spades, 10 of Diamonds,
7 of Hearts, 3 of Clubs, 5 of Clubs, 11 of Spades, 5 of Spades, 1 of Diamonds,
11 of Clubs, 9 of Spades, 1 of Hearts, 6 of Diamonds, 11 of Hearts, 4 of Clubs,
4 of Hearts, 9 of Diamonds, 10 of Clubs, 10 of Hearts, 1 of Spades, 8 of Diamonds,
3 of Spades, 13 of Clubs, 7 of Spades, 3 of Hearts, 9 of Hearts, 8 of Spades,
1 of Clubs, 12 of Diamonds, 2 of Spades, 2 of Diamonds, 5 of Hearts, 7 of Clubs,
9 of Clubs, 3 of Diamonds, 6 of Spades, 11 of Diamonds, 12 of Hearts, 6 of Hearts,
10 of Spades, 6 of Clubs, 2 of Clubs, 8 of Clubs.
```

Finally, if I then call `restockDeck()` followed by `showCards()` I see the following displayed:

```
1 of Clubs, 2 of Clubs, 3 of Clubs, 4 of Clubs, 5 of Clubs, 6 of Clubs, 7 of Clubs,
8 of Clubs, 9 of Clubs, 10 of Clubs, 11 of Clubs, 12 of Clubs, 13 of Clubs,
1 of Diamonds, 2 of Diamonds, 3 of Diamonds, 4 of Diamonds, 5 of Diamonds,
6 of Diamonds, 7 of Diamonds, 8 of Diamonds, 9 of Diamonds, 10 of Diamonds,
11 of Diamonds, 12 of Diamonds, 13 of Diamonds, 1 of Hearts, 2 of Hearts,
3 of Hearts, 4 of Hearts, 5 of Hearts, 6 of Hearts, 7 of Hearts, 8 of Hearts,
9 of Hearts, 10 of Hearts, 11 of Hearts, 12 of Hearts, 13 of Hearts, 1 of Spades,
2 of Spades, 3 of Spades, 4 of Spades, 5 of Spades, 6 of Spades, 7 of Spades,
8 of Spades, 9 of Spades, 10 of Spades, 11 of Spades, 12 of Spades, 13 of Spades.
```

(c) (Optional points) Write a class `Blackjack`. In this class you can set up a main method. You can use `Scanner` to interact with a user and simulate a game of Blackjack against the AI. For time reasons, this part of the assignment as been removed from the required part, but we suggest you try to implement the game using the two classes defined above. It will help you build a deeper understanding of how to use objects in Java. Here is a brief outline of how you can go about simulating Blackjack:

- Create a deck and shuffle it.
- Create an object of type Scanner to interact with the player (i.e. the user of your program).
- Deal two cards, one to the player, one to the dealer (the AI)
- Create a helper method that counts the points obtained in Blackjack from an arrays of Cards
- Show the player their first card and simulate the player's turn. Until the player wants a card and did not exceed 21 points do the following
  - Ask the player if the want a card
  - If so, deal a new card to the player and show it.
- If the player exceeded 21, then they lost.
- If they did not, then simulate the dealer's turn. Note that the dealer's turn is very similar to the players one with the exception that the dealer will keep drawing cards until they either get a score higher than the player or they exceed 21.

What To Submit

Please put all your files in a folder called Assignment4. Zip the folder (DO NOT RAR it) and submit it in MyCourses.

Inside your zipped folder, there must be the following files. **Do not submit any other files, especially .class files.** Any deviation from these requirements may lead to lost marks.

`GameOfLife.java`
`Card.java`
`Deck.java`
`Blackjack.java` (optional)
`Confession.txt` (optional) In this file, you can tell the TA about any issues you ran into doing this assignment.