CSE 8B Spring 2022

Assignment 3

Single-dimension arrays, multidimensional arrays, and objects and class

Due Date: Wednesday, April 20, 11:59 PM

Welcome back! Like in most classes, the difficulty of assignments will be gradually increasing as the class progresses. Be sure to start this assignment as EARLY as possible! You got this!

Learning goals:

- Implement methods for a class that manipulates **1D** arrays.
- Implement methods for a class that manipulates **2D** arrays.
- Write unit tests using objects and instance methods.

NOTE: This programming assignment must be done individually. Paired programming is NOT allowed for this assignment.

Coding Style (10 points)

For this programming assignment, we will be enforcing the CSE 8B Coding Style Guidelines. These guidelines can also be found on Canvas. Please ensure to have COMPLETE file headers, class headers, and method headers, to use descriptive variable names and proper indentation, and to avoid using magic numbers.

Part 0: Getting started with the starter code (0 points)

1. Make sure there is no problem with your Java coding environment. If there is any, then review Assignment 1, or come to the office/lab hours before you start Assignment 3.
2. Once again, in this PA, you will start with some code that we have written for you.

3. If you are using UCSD Linux Cloud (recommended) for the assignment, download the starter code from Piazza → Resources → Homework by opening a web browser on the Remote Desktop. You will be downloading both Assignment3.java and Assignment3Game.java. You can open Firefox by going to Applications → Firefox.
4. If you are working on your local machine, then you can download the starter code from a local web browser on your machine.

Part 1: Implement 1D Array Methods (30 points)

A char can be a number, too

In `Assignment3.java`, you will be implementing **2 methods** for 1D arrays. First, you will be performing (unrealistic) encryption, which would change a sequence of characters (stored in an array) into an unrecognizable "secret" so that no one else can read the original message. In the second method, you will be performing the opposite action, decrypting the secret back to the recognizable sequence of characters!

Aside: From a computer security perspective, this is not a very secure way of encrypting information!

In Java (and other C-family languages), it's possible to directly add two characters together. Recall that common characters, under the surface, can be represented by decimal code values (see Lecture 3, slides 34-36). This is demonstrated in the following code snippet and its output.

```
char newChar = 'a' + 'b';
System.out.println((int) newChar);
System.out.println((char) newChar);

The above code prints:
195
Ã
```

The decimal value of character literal `'a'` is 97, and the decimal value of character literal `'b'` is 98, so adding these two characters together into `newChar` gives us a char with the value 195, or char `'Ã'`. In the code above, you can see that we altered how `newChar` is printed by typecasting, but really, `newChar` is just 195, like how `'a'` is really just 97.

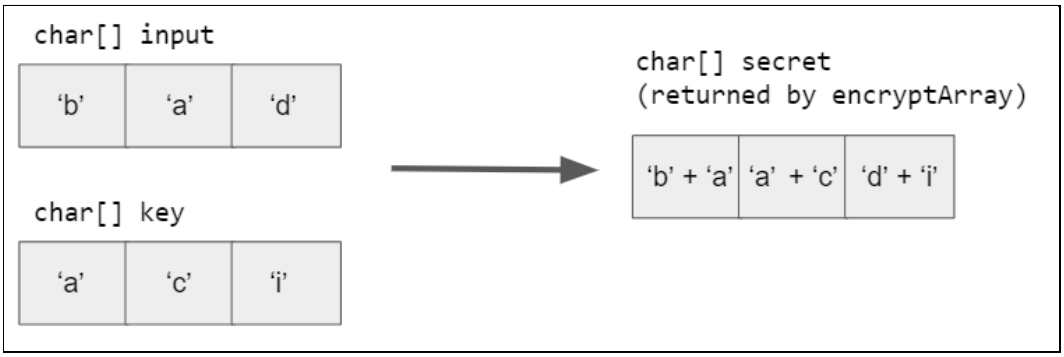
You will be implementing the two methods `encryptArray` and `decryptArray` in `Assignment3.java`, where you will be changing the Unicode values of characters in a similar fashion.

Method 1 - encryptArray (15/30)

```
public static char[] encryptArray(char[] input, char[] key)
```

This method takes in two `char` arrays, `input` and `key`, as parameters. To form the secret output that will be returned by this method, each individual character in `input` should be incremented by the respective character in the same position in the array `key`.

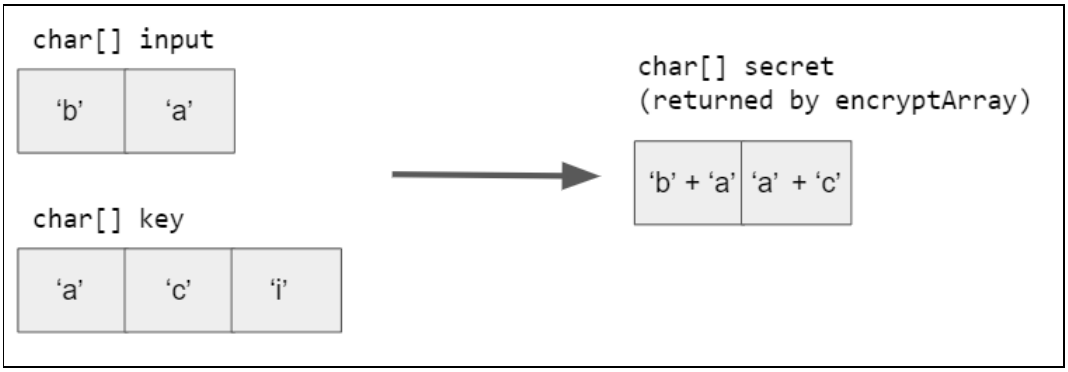
See the example below:

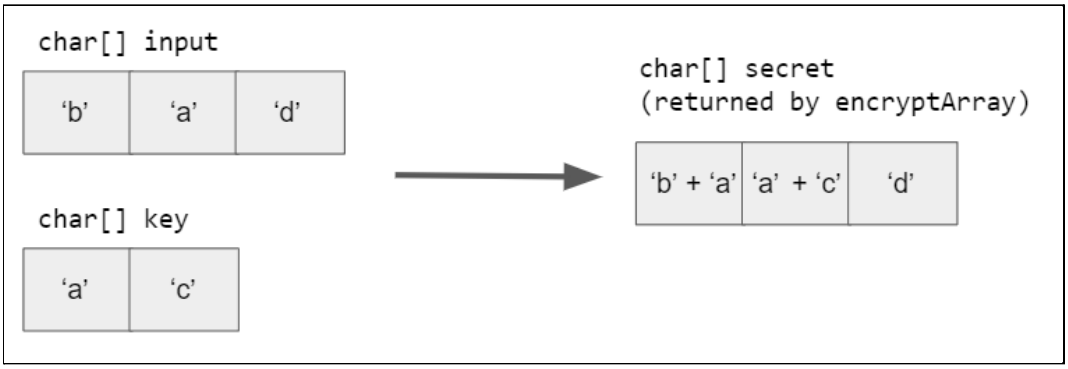


The character at `input[0]` is added to the character at `key[0]`, and so the character at `secret[0]` is the result of `'b' + 'a'` (which is character literal `'Ã'`, aka decimal value 195). And the second character in `input` was added to the second character in `key` to form the second character in `secret`, and so on.

Implement `encryptArray` to have the above functionality, returning a `char[]` (a character array) that is the result of combining `input` and `key` in the way described above. Assume that that `input` and `key` will only have uppercase and lowercase English alphabet characters.

You also must handle the cases where `input` and `key` are <u>not</u> the same length. Notice in the below examples that `secret` is always constructed to be the same length as `input`. Your code will give runtime errors (and won't pass our tests!) if you do not handle differences in length. Also, make sure that you <u>do not</u> change the values of the elements in `input` and `key`!





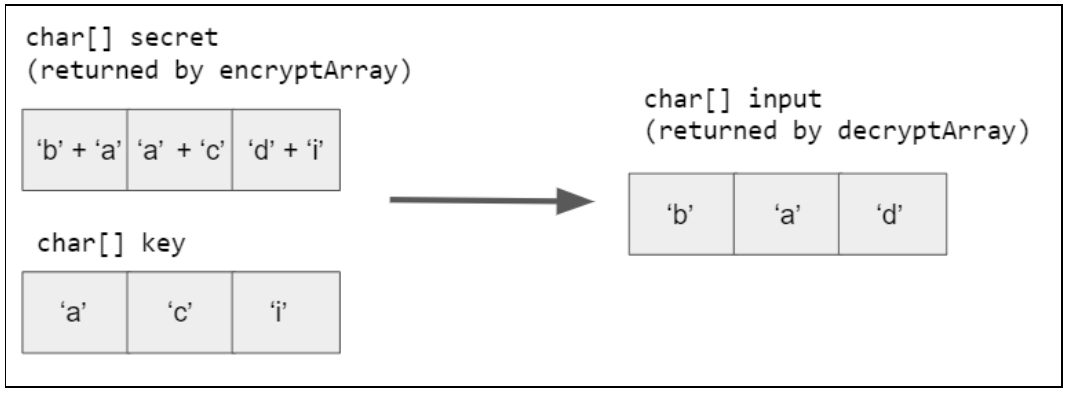
A hint for later: The compiler won't like it when you try to add two characters together and then save that value to a `char` array. Try to manually cast the type of the sum of the characters...

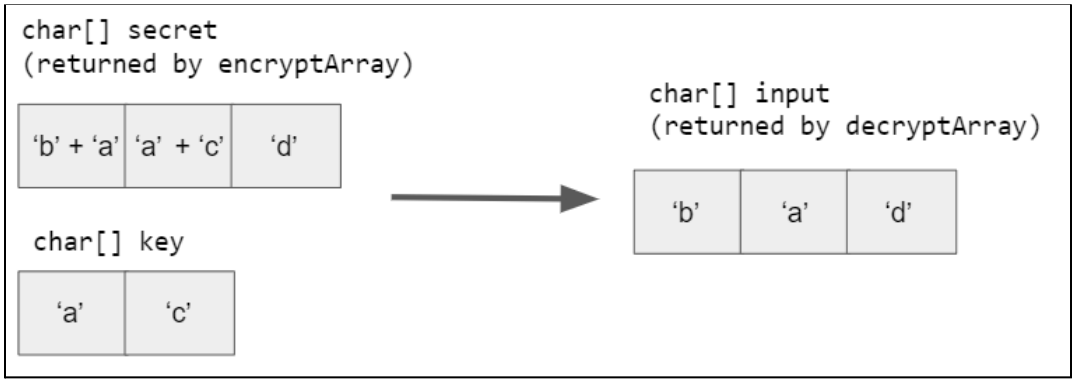
Note: Before moving onto writing code for `decryptArray`, it is strongly recommended that you temporarily skip over to Part 2 to unit test just `encryptArray`. Ensuring that `encryptArray` is working first will make your life easier, since `decryptArray` is very similar! Come back to `decryptArray` afterwards.

Method 2 - decryptArray (15/30)

```
public static char[] decryptArray(char[] secret, char[] key)
```

In `decryptArray`, you will be performing the inverse operation of `encryptArray` to reconstruct the original `input` array. This method takes in two `char` arrays, `secret` and `key`, as parameters - make sure that you do not change the values of the elements in `secret` and `key`! See the below examples:





Since you already wrote `encryptArray`, and since `decryptArray` just does the inverse of `encryptArray`, this should be enough information for you to complete this method!

Part 2: Compile, Run and UnitTest Your Code (10 points)

Just like in Assignment2, **in this part of the assignment, you need to implement your own test cases in the method called `unitTests`.**

In the starter code, some test cases are already implemented for you. You can regard it as an example to implement other cases. Recall, the general approach is to come up with different inputs and manually give the expected output, then call the method with that input and compare the result with expected output.

You are encouraged to create as many test cases as you think to be necessary to cover all the edge cases. The `unitTests` method should `return true` only when all the test cases are passed. Otherwise, it should `return false`. **To get full credit for this section, you should create at least five test cases that cover different situations (including the ones we have**

provided) for these two methods. In other words, you will need to create at least **three** more tests that test `encryptArray` and `decryptArray`.

To compare arrays by the equality of contents, you must use **Arrays.equals()**. See the given unit tests for examples.

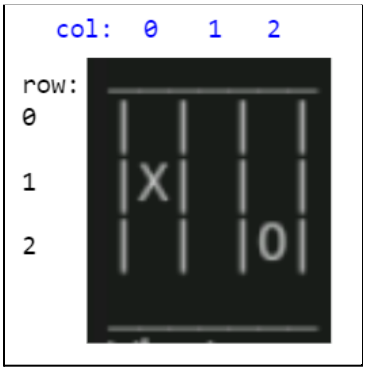
To help you create more test cases and find out the expected output of those cases, you can optionally use this char to ASCII value converter and/or this ASCII value to char converter. We can use these converters since ASCII is a subset of Unicode.

If a test is not passing, try temporarily printing the result of your method(s) and comparing them to the expected output.

Part 3: Implement 2D Array Methods (50 points)

TicTacToeGame - Introduction

In Part 3, you will be implementing TicTacToe on the command line! In this game, the command line will alternate between asking player X and player O to make their play - with player X always going first. Each play consists of entering a row index number and a column index number, where either an 'X' or an 'O' (depending on the player) will be placed on the 3x3 TicTacToe grid. If the game detects an invalid play, it will ask the player to make another play. The game ends when a player has won (filling up consecutive squares in a row, column, or diagonal) or when 9 turns have been made with no one winning.



Your first step is to rename `Assignment3Game.java` to `TicTacToeGame.java`. You will be writing the `TicTacToeGame` class in this file.

Unlike in Part 1, the majority of `TicTacToeGame` will be **instance methods**. This means that you should be implementing `TicTacToeGame` with objects and object data in mind. We will go over the instance methods that you need to implement one by one.

TicTacToeGame - Methods (20/50)

`TicTacToeGame()`
This is the constructor of the `TicTacToeGame` class. Here, you should be initializing the instance variable `grid` to a 2D character array with 3 rows and 3 columns. Every element in the 2D array should be initialized to the space character, which we give you in the character constant called `SPACE_CHAR` at the top of the file. You can also find that we already declared `grid` above the constructor.

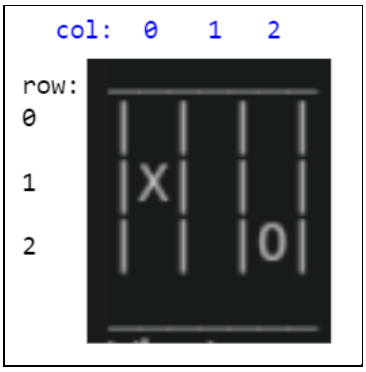
`char[][] grid`
This is the instance variable you just initialized in the constructor. As stated before, `grid` is a 2D character array, always with 3 rows and 3 columns.. Each character describes what was played at that position in the `grid`. `grid` can only contain these 3 characters:

1. `SPACE_CHAR` - no '`X`' or '`O`' has been put at this position in `grid`
2. `X_CHAR` - an '`X`' has been put at this position in `grid` by player X
3. `O_CHAR` - an '`O`' has been put at this position in `grid` by player O

For example, if `grid[row][col] == X_CHAR`, that means player X has put an '`X`' there.
You will be manipulating `grid` in the later methods so that it can remain up-to-date.

`int numTurns`
`numTurns` is an instance variable (not a method) that is supposed to keep track of the total number of turns made (from 0 to 9). You will have to manipulate `numTurns` in the later methods so that it can remain up-to-date. You can find that we already declared `numTurns` above the constructor.

`boolean putX(int row, int col)`
This method is for when Player X wants to make a play. It takes in two integers, a row index and a column index, such that an `'X'` character should be placed at the respective position on the `grid`.



In this image, 'X' is played at row index 1 and column index 0. 'O' is placed at row 2 and column 2.

If `row` and `col` result in an invalid play, then the method returns `false`. Otherwise, the method returns `true` for valid plays.

Plays are **invalid** when:

- `row` index is out of bounds of the grid (please recall that arrays are zero-indexed)
  - For example, `row = -1` is out of bounds
- `col` index is out of bounds of the grid (please recall that arrays are zero-indexed)
  - For example, `col = 4` is out of bounds
- a play has already been made at `grid[row][col]` (an `'X'` or an `'O'` is already at the position indicated by `row` and `col`)

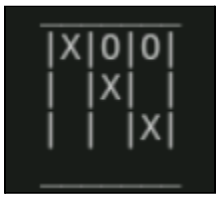
If a play is valid, remember to indicate that another turn has been made using one of the instance variables that we talked about earlier.

`boolean putO(int row, int col)`

This method is almost exactly the same as `putX()`, but for player O. The only difference is that you should be writing an `'O'` character instead of an `'X'` when `row` and `col` result in a valid play.

`static void print2DArray(char[][] arr)`

This method is less specific to the `TicTacToeGame grid`. It takes in an array of characters and prints it out in a particular way. For example, this output:



Is printed when we you pass in this array:

```
char[] arr = { {'X','O','O'}, {' ','X',' '}, {' ', ' ', 'X'} };
```

Here is what is being printed, in words:

- A line of underscores to delimit the start of the grid.
- A '|' character at the start/end of every column to delimit columns.
- The characters from `arr` in the correct row & column position, with each line in the output representing a different row. You would want to iterate through `arr` to get each character somehow.
- A line of underscores to delimit the end of the grid.

Make sure to look through the constants we defined at the top of the `TicTacToeGame.java` file to see which ones you can use to help you format the printing! We will be checking for exact output!

`int checkWinner(int row, int col)`
`checkWinner` handles all the logic for determining whether the `TicTacToeGame` has ended. The `row` and `col` parameters provided describe the position of the most recent valid play and will be helpful for checking the row win condition and column win condition.

We have provided four integer constants at the top of the file, each that signal a different state of the game. You will be returning the corresponding state.

1. `INT_NO_WINNER_YET` - the game should continue, neither player X/O have won and 9 turns have not been reached.
2. `INT_X_WINS` - player X has won, so the game should end.
3. `INT_O_WINS` - player O has won, so the game should end.
4. `INT_MAX_TURNS_REACHED` - 9 valid turns have been played, meaning no more plays can be made, so the game should end.

To determine which integer constant to return, there are **six** different general game conditions that you will need to check in `checkWinner`:

1. *Row win condition* - There are three `'X'` characters or three `'O'` characters in the current `row`. Return either `INT_X_WINS` or `INT_O_WINS` (based on if it's `'X'` or `'O'`).
2. *Col win condition* - There are three `'X'` characters or three `'O'` characters in the current `col`. Return either `INT_X_WINS` or `INT_O_WINS`.
3. *Top-left to Bottom-right diagonal win condition* - There are three `'X'` characters or three `'O'` characters in the diagonal that starts top-left and ends bottom-right. Return either `INT_X_WINS` or `INT_O_WINS`.

4. *Bottom-left to Top-right diagonal win condition* - There are three `'X'` characters or three `'O'` characters in the diagonal that starts bottom-left and ends top-right. Return either `INT_X_WINS` or `INT_O_WINS`.
5. *Max turns reached end-game condition* - **9** valid turns have been played, so all squares are filled. Return `INT_MAX_TURNS_REACHED`.
6. If none of the above cases are true, return `INT_NO_WINNER_YET`.

Before you continue to the next section, make sure to compile and run the unit tests we provided! Our unit tests are quite comprehensive, but we left a few test cases missing that should be simple for you to add (hint: check the comments in the unit test method!). We won't be checking unit tests that you write for this section, but writing them will help you confirm if `TicTacToeGame` is working!

TicTacToeGame - Main Method (30/50)

Once your unit tests pass and you are confident that the above methods are working, it's time for you to put everything together so that we can play from the command line!

Similarly to Assignment2, all of this command line functionality will be implemented in the `main` method. We have defined several `String` constants for you (at the top of `TicTacToeGame.java`) to use for printing. Assume that every `String` constant needs to be printed with a new line after it.

The following steps outline the logic of the code to be written in `main`.
Hint: You will be using all of the instance methods you wrote, and you will need to use (multiple) loops!

1. Print out the welcome message to welcome the user to the game!
2. Print out the grid (you wrote a method earlier...).
3. Ask player X what row they would like to play. Scan the integer that is entered.
4. Ask player X what column they would like to play. Scan the integer that is entered.
5. Try to put an 'X' character at the specified row and column on the TicTacToe grid. If the play is invalid, print the invalid play message, and go back to step 3 to ask player X again.
6. If the play is valid, move on by printing out the grid (which should be updated with the newly-placed 'X'). Then, check if the game ends (such as if someone wins). If so, skip to step 11.
7. Ask player O what row they would like to play. Scan the integer that is entered.

8. Ask player O what column they would like to play. Scan the integer that is entered.
9. Try to put an '0' character at the specified row and column on the TicTacToe grid.. If the play is invalid, print the invalid play message, and go back to step 7 to ask player O again.
10. If the play is valid, move on by printing out the grid (which should be updated with the newly-placed '0'). Then, check if the game ends (such as if someone wins). If so, skip to step 11. Otherwise, go back to step 3 (back to player X's turn).
11. The game has ended! Print out the respective ending message depending on if player X won, if player O won, or if 9 turns (the max) have been completed with no one winning.

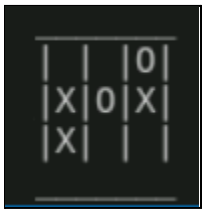
Optional (but very helpful) suggestions:

- You can use the instance variable `numTurns` and the modulus operator `%` to decide when it is player X's turn or player O's turn.
- View the **4/14** Discussion section to see the command line `TicTacToeGame` in action. Your output will be compared to the reference solution being run by the TA. Line spacing and other formatting of the output should be **exact** for full points in this section.
- Once you are done with writing the code for this section, start playing your game in the command line to test out whether everything works as expected. Try to invoke as many different cases as possible to see how your code handles them.

[OPTIONAL] TicTacToeGame - Generate Random Turns (0/50)

As an optional, ungraded extension of this assignment, you can implement unique functionality that will allow the user to generate a `TicTacToeGame` with a specified number of random turns already played. Be sure to make a copy of your `TicTacToeGame.java`, as we will be grading for the game **without** the random functionality extension. **If you aren't doing this section, please skip to Submission.**

In this program, before making any plays, the user can input that they want **5** random turns to be played, and instead of being a blank grid, the game can start with the grid looking something like this:



Implement this functionality inside of an overloaded constructor for `TicTacToeGame` with this method signature:
`public TicTacToeGame(int numRandTurns, int randSeed)`

First, initialize the `TicTacToeGame`'s `grid` instance variable like you did in the zero-parameter constructor.

Next create a **Random** object from **java.util.Random** with its seed as `randSeed`. You will be using this object and seed to pseudo-randomly generate row and column values.

In a loop,

- Generate a random row index. Generate a random col index.
- Depending on whether it's player X's turn or player O's turn, try to put an `'X'` or an `'O'` at that row and col on the grid.
- If the play was invalid, generate a new random row index and col index and try again.
- If the play was valid, call `checkWinner` on the random row and col index.

The above loop should run such that the number of valid plays ends up exactly equal to the value of `numRandTurns` OR up to when `checkWinner` signals that the game ends.

Now, we want to implement the functionality on the command line. In main, add these two steps after printing the welcome message:

1. Ask the user if they want to start a game with some random turns already played. Scan the `'y'` char or the `'n'` char that is entered.
2. If an `'n'` character was entered, create a new `TicTacToeGame` object with no random turns and print out the grid. If `'y'` was entered (meaning yes random turns):
    a. Ask the user how many random turns they want. Scan the integer that is entered.
    b. Ask the user what seed to use for the random turn generator. Scan the integer that is entered.
    c. Create a new `TicTacToeGame` object with the specified number of random turns and the seed.
    d. Print out the grid.
    e. Then, check if the game ended during the random turns (such as if someone wins). If so, skip to the step for printing the correct end-game message. Otherwise, if it didn't, skip to the step to ask player X for their play.

Example output:

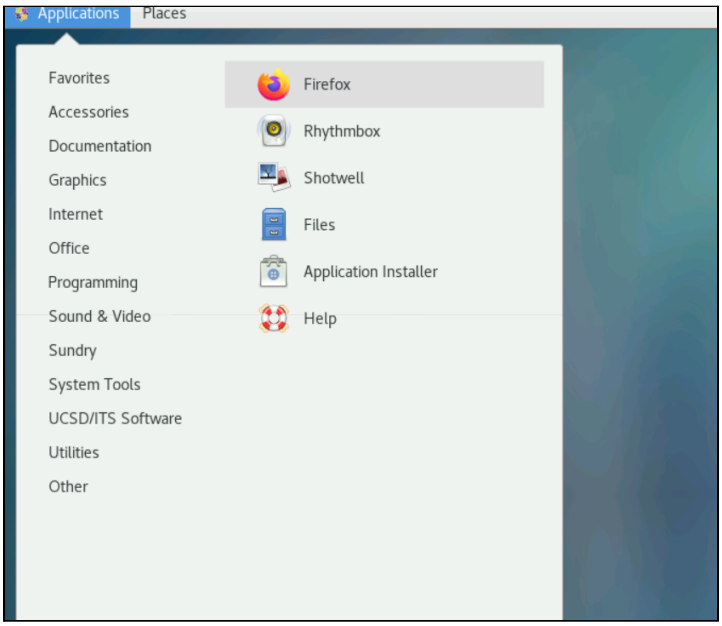
```
Welcome to TicTacToeGameRand!
Would you like to play with some random turns already played? Enter (y/n):
y
How many random turns would you like to have been played? Enter an int:
5
What seed would you like to use to generate random turns? Enter an int:
5

 _____
| |X|X|
| |O|O|
| |X| |
 _____
What row would player O like to play? Enter an int:
```

Submission

You're almost there! Please follow the instructions below carefully and use the **exact submission format**. Because we will use scripts to grade, **you may receive a zero** if you do not follow the same submission format.

1. Open a web browser on the Remote Desktop. You can open Firefox by going to Applications → Firefox.



2. Open Gradescope in Firefox and login. Then, select this course → PA3.
3. Click the DRAG & DROP section and directly select the required files **Assignment3.java** and **TicTacToeGame.java**. Drag & drop is fine. Please make sure you don't submit a zip, just the separate files in one Gradescope submission. Make sure the names of the files are correct.

4. You can resubmit unlimited times before the due date. Your score will depend on your final (most recent) submission, even if your former submissions have higher scores.
5. Your submission should look like the below screenshot. If you have any questions, feel free to post on Piazza!