CS-330: PROGRAMMING LANGUAGE PROJECT

Final Project - Language: Java

Java Tic-Tac-Toe

A tic-tac-toe game programmed in Java that runs in the command line with three different modes: Single Player Mode, Two Player Mode, and Spectator Mode. The game keeps track of the winning combinations within the board as well as the list of previous winners within one execution of the program.

Eliana Idalys Lopez
https://github.com/elianalopez

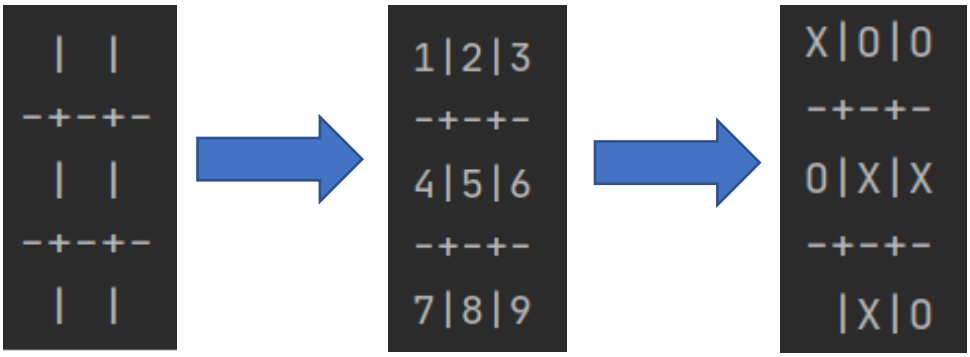
CS-330: Programming Language Project (PLP)

Final Project: Tic-Tac-Toe

Overview

For my final project utilizing the programming language Java, I have decided to work on a tic-tac-toe game that functions within the command line. The game works and functions within the same rules as a regular tic-tac-toe game, first to get three consecutive items in a row would win, where the items would be either an “X” or an “O.”

Within the command line would be a gameboard that is constructed from a 2D char array. In order to place a symbol within the command line, numbers [1-9] would add a symbol within the empty spaces of the gameboard, the images below illustrate what part of this process would look like.



Game Board

Game Board with numerical placements

The Game Board with our symbols

Within **Appendix A**, there is the method that allows the placement of pieces for each player, which works via a switch case statement.

How the program keeps track of winning positions is that in two parts, at the top of the Java code an array list is created to help keep track of the winning combinations, and winning combinations are first stated in a method known as `winningPatterns`. So how patterns are declared as winning is basically based off the numerical placements in the board seen above. If you compare the winning patterns with the numerical placements above you will notice how it simulates a combination of winning patterns in a tic-tac-toe game.

```
public String winningPatterns(){
    List topRow = Arrays.asList(1, 2, 3);
    List middleRow = Arrays.asList(4, 5, 6);
    List bottomRow = Arrays.asList(7, 8, 9);
    List leftColumn = Arrays.asList(1, 4, 7);
    List middleColumn = Arrays.asList(2, 5, 8);
    List rightColumn = Arrays.asList(3, 6, 9);
    List acrossRight = Arrays.asList(1, 5, 9);
    List acrossLeft = Arrays.asList(7, 5, 3);
```

A portion of `winningPatterns` Method [See lines **551 – 597** for full method]

Recent Changes

Overall my goal within creating these recent changes is to first clean up my Java code, and from this change came other slight changes such as modifying the start screen and then expanding this project by creating new modes followed suit.

Some of the recent changes I have added within this project is an entirely new start menu, the start menu is different not only in appearance (See **Appendix B** and **Appendix C**) but also within functionality because of how I modified it. The original start menu was built utilizing a while loop and an if-else statement where the user would have to press "O" in order to *sequentially continue* towards the game. Within the modified code, inside of the while loop is a switch case statement, where each mode of a game, I have combined into each as a separate method (**Appendix D**).

Some outstanding differences that occurred due to my change in the start screen also affected regarding how the entire program runs. For example, once I finish a game, I would return to the start menu in the newly modified code while if I finish a game in the old code, the program will end, this is due to the while loop being directly outside and connected to the scope of the start menu.

Old Start Menu Code:

```
boolean S = true;

Scanner keyboard = new
Scanner(System.in);

while(S) {
    welcome();
    String input = keyboard.nextLine();
    if (input.equals("I")) {
        instructions();
    } else if (input.equals("O")) {
        break;
    }
}
```

New Start Menu Code [36-82]:

```
while (true) {
   …
switch (choice) {
    case 1:
        Game oneGame = new Game();
        oneGame.onePlayer(gameBoard);
        break;
    case 2:
        Game twoGame = new Game();
        twoGame.twoPlayer(gameBoard);
        break;
    case 3:
        Game cpuGame = new Game();
        cpuGame.spectator(gameBoard);
        break;
    case 4:
        instructions();
        break;
    default:
        System.out.println("\nThis is not
        a valid Menu Option! Please Select
        Another!!\n"); "O"
        //if inputs do not match cases
        this will print
        break;
    }
}
```

Since the game functions within a loop another change I was required to make was create a method that would reset the game board. It was a straight forward process because all I did was replace the previous rows and elements that was utilized for our X's and O'x with spaces, which was how our game board was originally constructed within `Lines 19-25 of our code` (See **Appendix E**).

Another change within the program I made is by expanding the program by adding two different modes: single player mode and spectator mode (my favorite).

The original program I showed in my presentation was a local 1-v-1, where each player would play a game of tic-tac-toe locally, and take turns until one player wins, or until both succumb to a tie. In both single player mode and spectator mode, there is a CPU player this time who places its piece based off a picking a random number between [1-9].

The first project "Algorimon" inspired me by how I believe there is a random timer within the process of the first presentation in order to make the game seem more life-like. So I in turn also decided to add a random timer prior before a CPU makes a move, so the duration of each move a CPU takes will vary. In **Appendix F** you can see how a CPU would usually take turns, and the random timer to pause in between each turn happens within this line:

```
Thread.sleep((long)(Math.random()*2000));
```

I found this line of code being the easiest way to create some type of random pauses within the Java code itself. One thing I learned is that if you view Appendix D, after adding the line above my editor recommends adding "`throws InterruptedException`" within the Java Method Header.

Now how spectator mode works is that, two CPU players would go against one another until one is declared a winner or until both succumb to a tie. Spectator mode is my favorite because I do not have to test the game manually if it works for this mode since CPU's play it. Also that I should note that due to the additions of CPU's I had to modify some methods to include these two new participants. Methods that I had to modify include `turn` and `winningPatterns`.

Within Single Player Mode, it is a hybrid between both spectator mode and our local 1-v-1 mode that I demonstrated in class, since there is a local player along with the addition of a CPU playing against that one player.

Lastly the most recent change in overall is the Winner List I have created which keeps track of previous winners in the game, in the case there is no winner from not starting the game it will leave a positive message to go and attempt the game. In the case of a tie both players are added because everyone is a winner (and I didn't know how to fix that, but that's our little secret)! View **Appendix G** and **Appendix H** for the winners list.

Overall these are the changes I that I have established within this Java project. My favorite part out of all recent changes other than spectator mode is by creating a program that would always bring me back to the start menu like a regular tic-tac-toe game!

Hardships and Learning Scenarios

Overall there were a few hardships I had while doing this project. One of them was is sudden failure of my original laptop, which is the largest obstacle I encountered due to needing a laptop to finish this project, which has been solved rather quickly luckily. 😊

Another problem I had was just how I declared Java methods or data types such as making them static or not, since it was something that is new to be, but I managed to learn so much about Java within this process. I also learned a lot about the java.util package.

Appendices

Appendix A

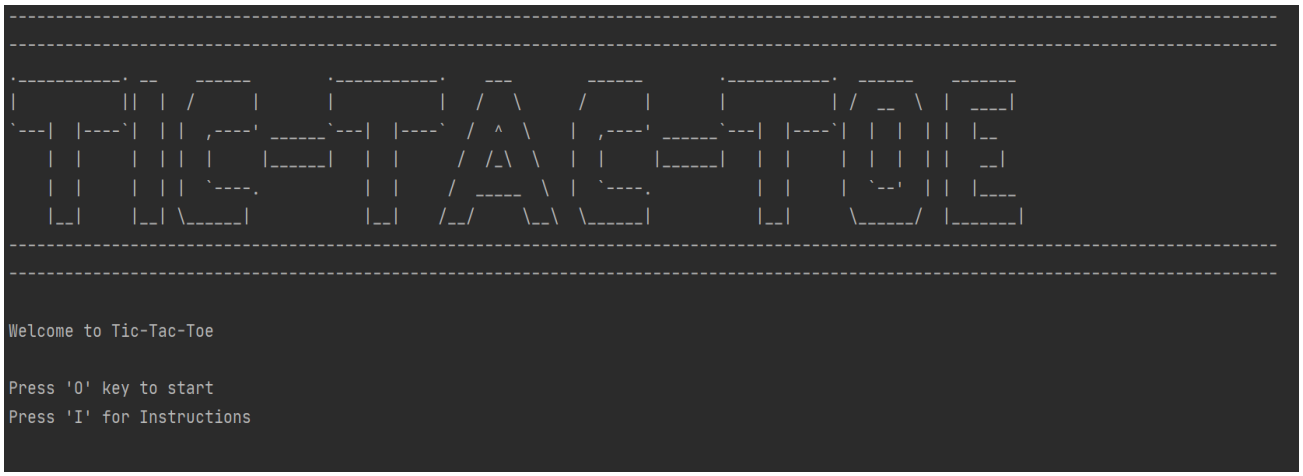
```
public void turn(char[][] gameBoard, int pos, String user) {
    char symbol = ' ';

    if (user.equals("player1")||(user.equals("CPU1"))) {
        symbol = 'X';
        playerPos1.add(pos);
        CPUPos1.add(pos);
    } else if (user.equals("player2")||(user.equals("CPU2"))){
        symbol = 'O';
        playerPos2.add(pos);
        CPUPos2.add(pos);
    }

    switch (pos) {
        case 1:
            gameBoard[0][0] = symbol;
            break;
        case 2:
            gameBoard[0][2] = symbol;
            break;
        case 3:
            gameBoard[0][4] = symbol;
            break;
```

The method on how placing pieces works, cut off but expands to case 9 [See lines **455-499**]

Appendix B

The New Start Screen
New features are Single Player Mode, Spectator Mode, and a Winner List

Appendix C



The Old Start Screen

Appendix D

```
public void onePlayer(char[][] gameBoard) throws InterruptedException {...}

public void spectator(char[][] gameBoard) throws InterruptedException {...}

public void twoPlayer(char[][] gameBoard) {...}
```

Methods Created of each mode [See lines **218-286**, **289-353**, **356-420**]

Appendix E

```
public static void resetGameBoard(char[][] gameBoard) {
    gameBoard[0][0] = ' ';
    gameBoard[0][2] = ' ';
    gameBoard[0][4] = ' ';
    gameBoard[2][0] = ' ';
    gameBoard[2][2] = ' ';
    gameBoard[2][4] = ' ';
    gameBoard[4][0] = ' ';
    gameBoard[4][2] = ' ';
    gameBoard[4][4] = ' ';
}
```

The `restGameBoard` Method would replace the following selected [See lines **534-544**]

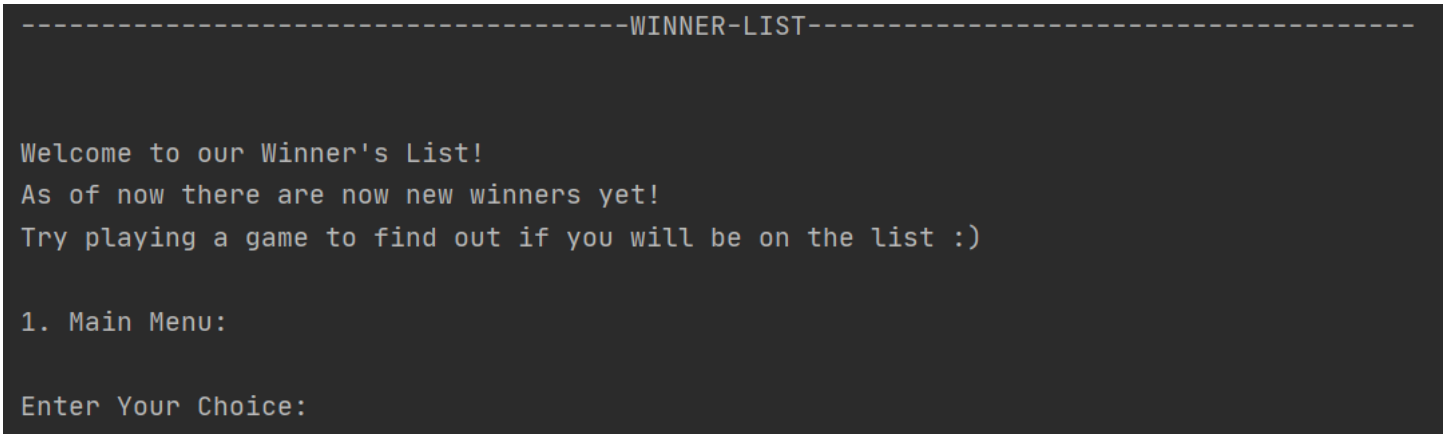
rows and elements of our gameboard with spaces

Appendix F

```
while(true){
    //CPU1's turn
    Random rand = new Random();
    System.out.println(CPU1 + "'s turn, please placement (1-9): ");
    Thread.sleep((long)(Math.random()*2000));
    int X = rand.nextInt( bound: 9) + 1;
    while(CPUPos1.contains(X) || CPUPos2.contains(X)){
        Thread.sleep((long)(Math.random()*2700));
        X = rand.nextInt( bound: 9) + 1;
    }

    turn(gameBoard, X,  user: "CPU1");
```

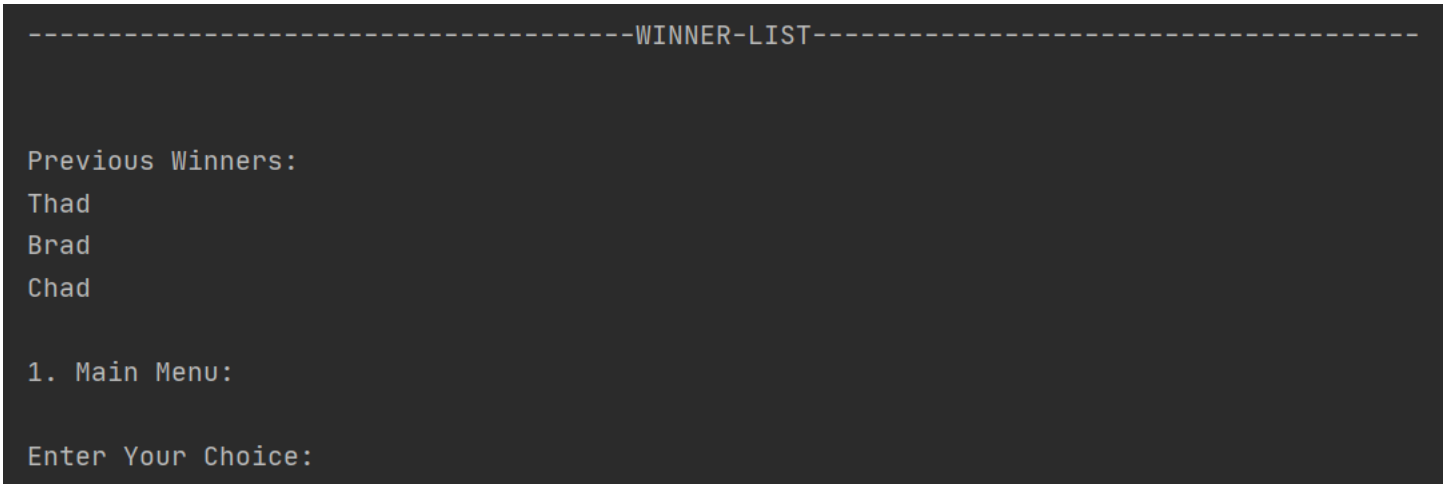
How turns happen for CPU's [See lines **308-318**]

Appendix G



Winner List before any wins
Utilized the built-in `isEmpty()` method of the ArrayList to check for previous winners [See lines **505-530**]

Appendix H



Winner List with previous winners [See lines **505-530**]

Note: Citations Needed are within the comments of the code directly above the targeted selection