Overview

In this second CS1002 practical, I was asked to code a game of Connect Four played by two human players. Just like the previous assessment, the output had to follow a certain format, especially for the representation of the board, the pieces, and the input/output messages. I also had to make sure that my program passed the stacscheck tests, and I also had to test my code manually.

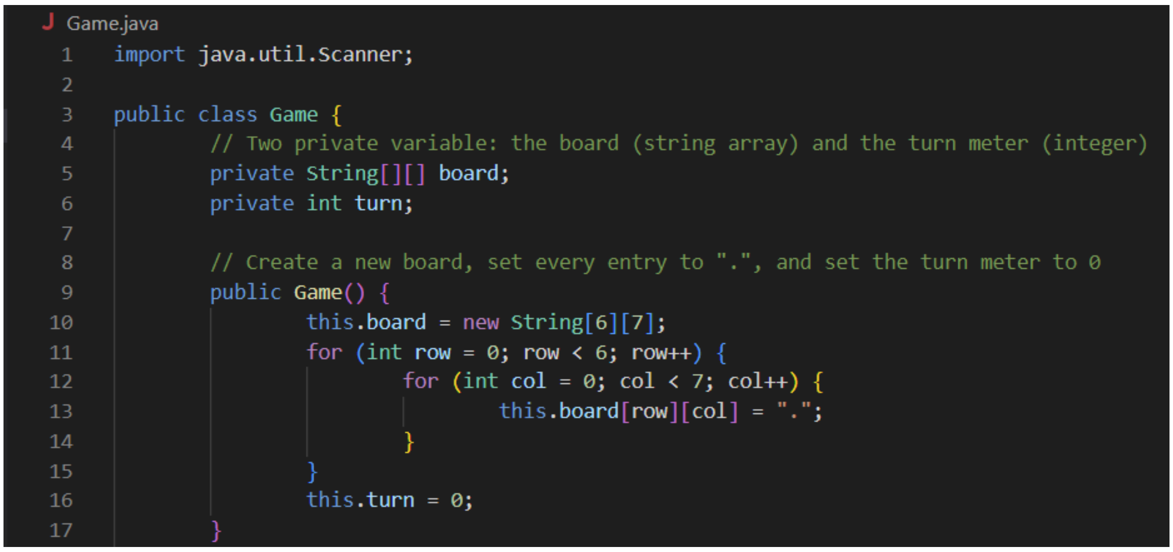
All in all, I've achieved the previously established goals:
- The game of Connect Four is playable by two human players
- The output is in accordance with the practical's specifications
- All tests are successful (see **Testing** for more details)

Design

The W09 Practical is divided in two Java files:
- On one hand, Game.java is where all the code related to the game is stored: this is the most important file, as it contains the core methods of the Connect Four game.
- On the other hand, W09Practical.java creates a new Game object (defined in the Game.java file) and runs the game.

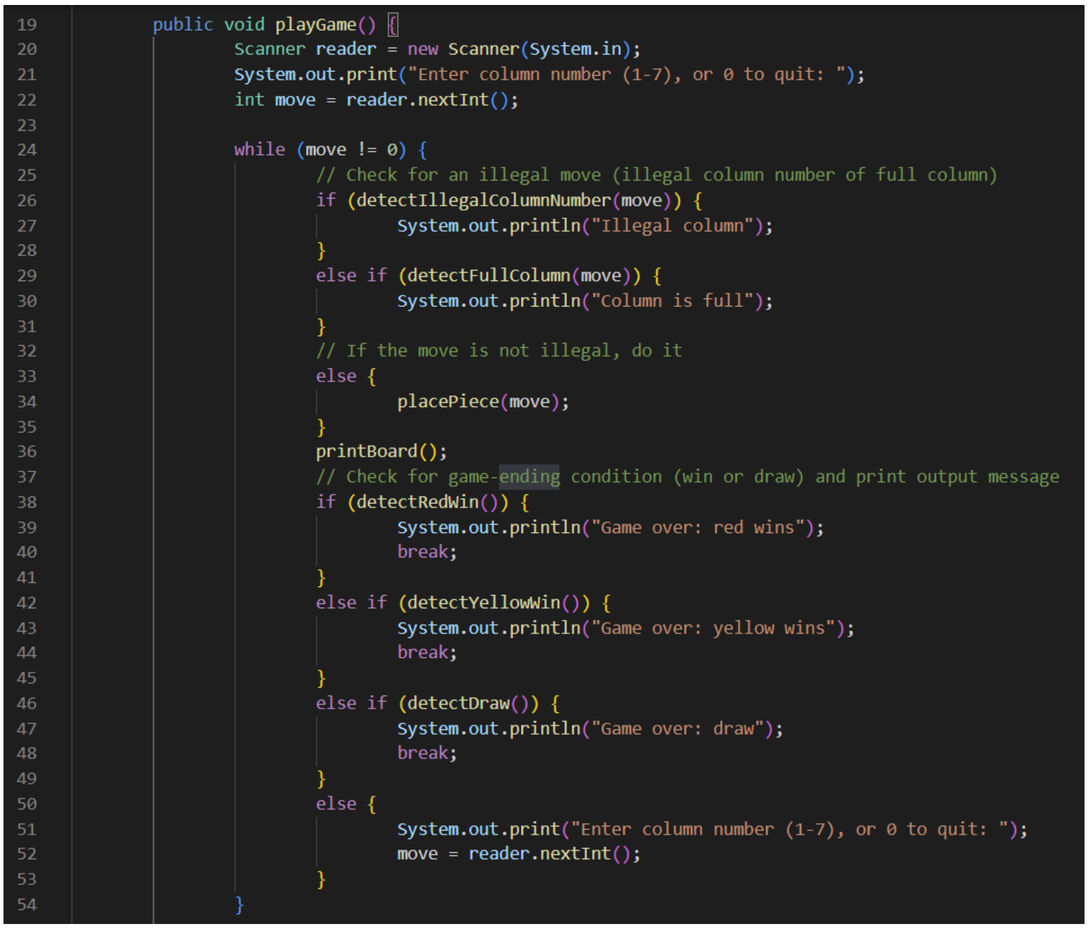
I will describe Game.java in more details, as I did not modify W09Practical.java.

The *Game* class in Game.java only uses two private variables: *board*, a String array representing the board, and *turn*, an integer for the turn meter. The syntax I've used is *private String[][] board;* and *private int turn;*. These variables are initialised in the constructor *Game*: the dimensions of *board* are set to 6x7 (6 rows and 7 columns), every entry of *board* is set to "." (as every entry of the board is empty at the beginning of the game), and *turn* is set to 0. To do that, I've used two *for* loops: one goes through every row, and the other one (nested in the first) goes through every column per row.


Game.java
```java
import java.util.Scanner;

public class Game {
        // Two private variable: the board (string array) and the turn meter (integer)
        private String[][] board;
        private int turn;

        // Create a new board, set every entry to ".", and set the turn meter to 0
        public Game() {
                this.board = new String[6][7];
                for (int row = 0; row < 6; row++) {
                        for (int col = 0; col < 7; col++) {
                                this.board[row][col] = ".";
                        }
                }
                this.turn = 0;
        }
```
Figure 1: The two *private* variables and the constructor

The next method in the *Game* is *playGame*. This method is the "core" of the program: it runs the game of Connect Four until either one player wins, one player exits the game or there's a draw. This method uses other methods that are defined afterwards, but what they do is relatively explicit thanks to their names.

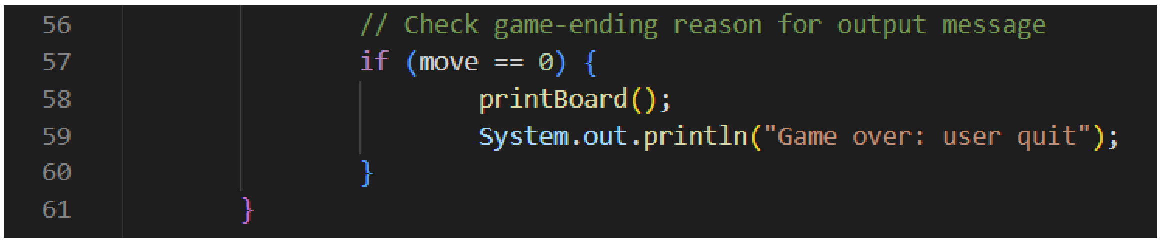
playGame begins by creating a new *Scanner* object from the *java.util.Scanner* class. This object, called *reader*, is used to input the user what move he/she wants to do, using *System.out.print(""Enter column number (1-7), or 0 to quit: ");*, and store the inputted move in an integer called *move* (*int move = reader.nextInt();*).
I've then used a *while* loop to run the game. The Boolean condition for the loop to stop is if *move* is equal to 0: if the user enters 0 when inputted, then he/she quits the game, and the loop stops. If the user does not enter 0, then we must check whether the number inputted is an illegal move, and (if not) if the column entered is full. If the move is "legal", then place a piece in the user inputted column. To do so, I've used an *if… else if… else…* statement (see Figure 2.1). After checking for an illegal move, I was required to print out the board, using the *printBoard()* method, and check for a game-ending condition. If either the yellow or the red player wins, or the game is a draw, we break out the loop using the *break;* keyword and print out the corresponding output message. Otherwise, we input the user again what move he/she wants to do and store it in *move*.
The only ways to break out this *while* loop is either the game ends (win or draw) or one user quits.



```java
	public void playGame() {
		Scanner reader = new Scanner(System.in);
		System.out.print("Enter column number (1-7), or 0 to quit: ");
		int move = reader.nextInt();

		while (move != 0) {
			// Check for an illegal move (illegal column number of full column)
			if (detectIllegalColumnNumber(move)) {
				System.out.println("Illegal column");
			}
			else if (detectFullColumn(move)) {
				System.out.println("Column is full");
			}
			// If the move is not illegal, do it
			else {
				placePiece(move);
			}
			printBoard();
			// Check for game-ending condition (win or draw) and print output message
			if (detectRedWin()) {
				System.out.println("Game over: red wins");
				break;
			}
			else if (detectYellowWin()) {
				System.out.println("Game over: yellow wins");
				break;
			}
			else if (detectDraw()) {
				System.out.println("Game over: draw");
				break;
			}
			else {
				System.out.print("Enter column number (1-7), or 0 to quit: ");
				move = reader.nextInt();
			}
		}
```

Figure 2.1: The *while* loop in the *playGame* method

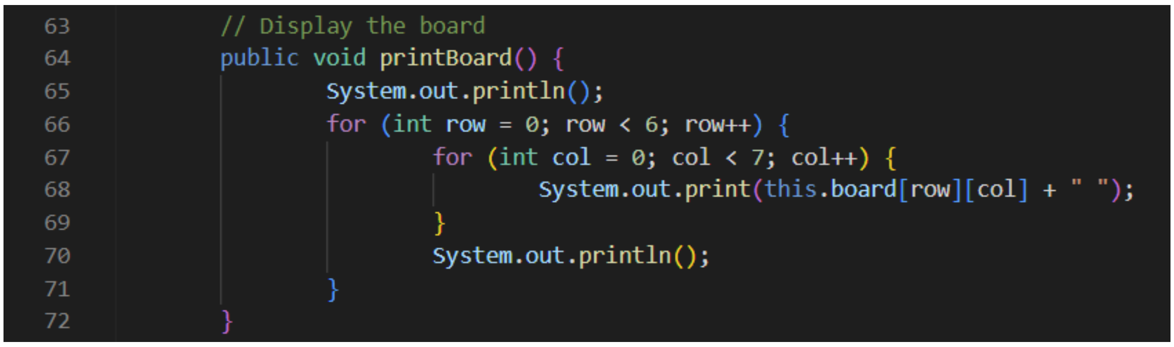
After breaking out from the *while* loop, to print out the correct output message if a user quits, I had to check if the reason for breaking out the loop is if *move* is equal to 0. If so, the system prints out the board one last time and the output message *System.out.println("Game over: user quit")*.



```
            // Check game-ending reason for output message
            if (move == 0) {
                printBoard();
                System.out.println("Game over: user quit");
            }
        }
```

Figure 2.2: The end of the *playGame* method

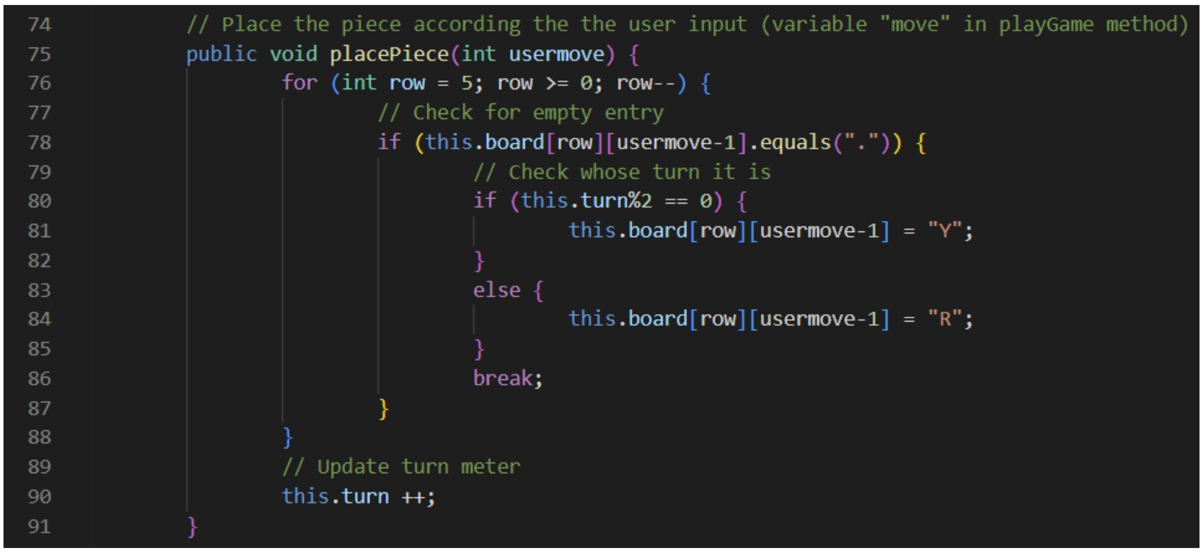
I shall now describe the different methods used in the *playGame* method, starting with *printBoard*. The first thing I've spotted is that every time I was asked to print out the board, I had to skip a line after the previous message. Thus, the method starts with *System.out.println();* (prints out an empty line). I then used two *for* loops, one nested in the other, to get the entries one by one. For each row, the system prints out the entries of the *board* array followed by a space, and a new line after all entries are printed out.



```
    // Display the board
    public void printBoard() {
        System.out.println();
        for (int row = 0; row < 6; row++) {
            for (int col = 0; col < 7; col++) {
                System.out.print(this.board[row][col] + " ");
            }
            System.out.println();
        }
    }
```

Figure 3: The *printBoard* method

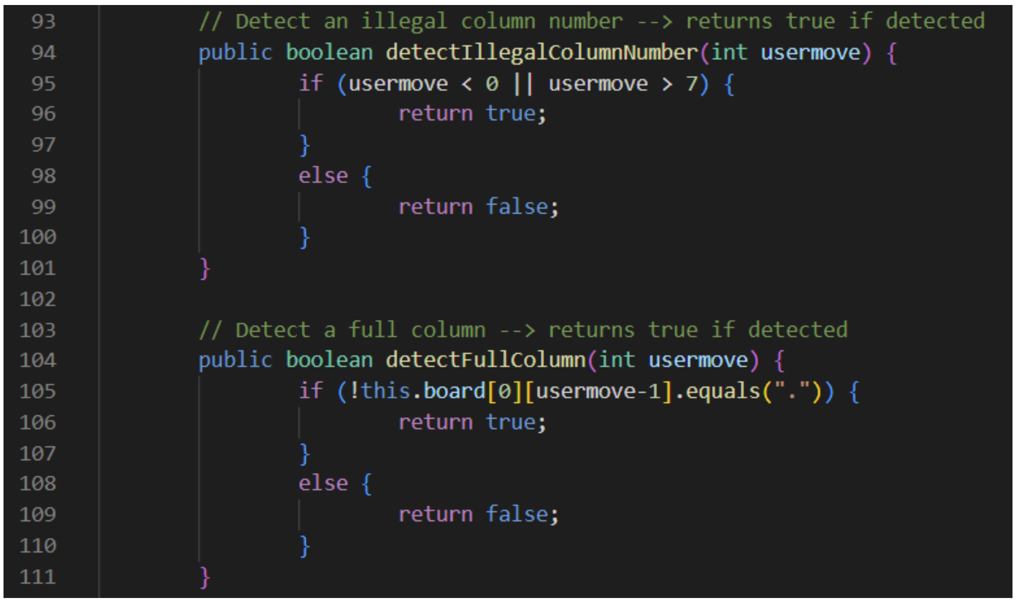
The next method is *placePiece*. It takes an integer *usermove* as a parameter and updates the board according to the *usermove*. In the *playGame* method, *usermove* is replaced by the variable *move*, the integer from 1 to 7 corresponding to the index of the column in which the user wants to put a pawn. In a real game of Connect Four, the pawns "fall" in the board to the "lowest" empty space in the column. That is why I used a *for* loop to cycle through the rows from the bottom row (with index 5) to the top row (with index 0). For each row, I checked if the entry is empty, with an *if* statement. Note that in Figure 4, the entry is accessed with *this.board[row][usermove-1]*: *usermove* being an integer from 1 to 7, and the indexes of the columns of the *board* array ranging from 0 to 6, we have to subtract 1 from the integer inputted by the user to get the column number. After finding the "lowest" empty entry in the board, I checked whose turn it is: as the yellow player plays first, it is his/her turn when the turn meter is even (the first move of the game happens when the turn meter is equal to 0, i.e., an even number), and it's the red player's turn when the turn meter is odd. Knowing who's turn it is, and where to update the board, I modified the empty entry, setting its value to "Y" or "R" accordingly. After updating the board, I used the *break;* keyword to exit the *for* loop, and after the *for* loop, I increased the turn meter by one.



```java
    // Place the piece according the the user input (variable "move" in playGame method)
    public void placePiece(int usermove) {
        for (int row = 5; row >= 0; row--) {
            // Check for empty entry
            if (this.board[row][usermove-1].equals(".")) {
                // Check whose turn it is
                if (this.turn%2 == 0) {
                    this.board[row][usermove-1] = "Y";
                }
                else {
                    this.board[row][usermove-1] = "R";
                }
                break;
            }
        }
        // Update turn meter
        this.turn ++;
    }
```

Figure 4: The *placePiece* method

The next methods are *detectIllegalColumnNumber* and *detectFullColumn*. Just like the previous method, they take an integer *usermove* as a paramterer. They both are Booleans, as they return true if the user inputs an illegal column number or if the chosen column is full.
The *detectIllegalColumnNumber* is made of one *if… else…* conditional statement: *if (usermove < 0 || usermove > 7) {*. Indeed, the user can input an integer from 0 (included) to 7 (included): 0 to quit, 1 to 7 to place a piece in a column. Therefore, if the user input is strictly less than 0, or (the double pipe syntax *||*) strictly greater than 7, the method detects an illegal column number and returns true. Otherwise, it returns false.
The *detectFullColumn* is also made of one *if… else…* statement: we just have to check if the inputted column is full. The column is full if the "upmost" entry is occupied. Therefore, if this entry is not equal to "." (the full-stop representing an empty entry), then the method returns true; otherwise, it returns false.
The fact that these methods return true when something wrong happens (it might seem counter-intuitive) is used in the *playGame* method (see l. 26 and l. 29 in Figure 2.1), as to "enter" the *if* statement, the Boolean condition must be true.

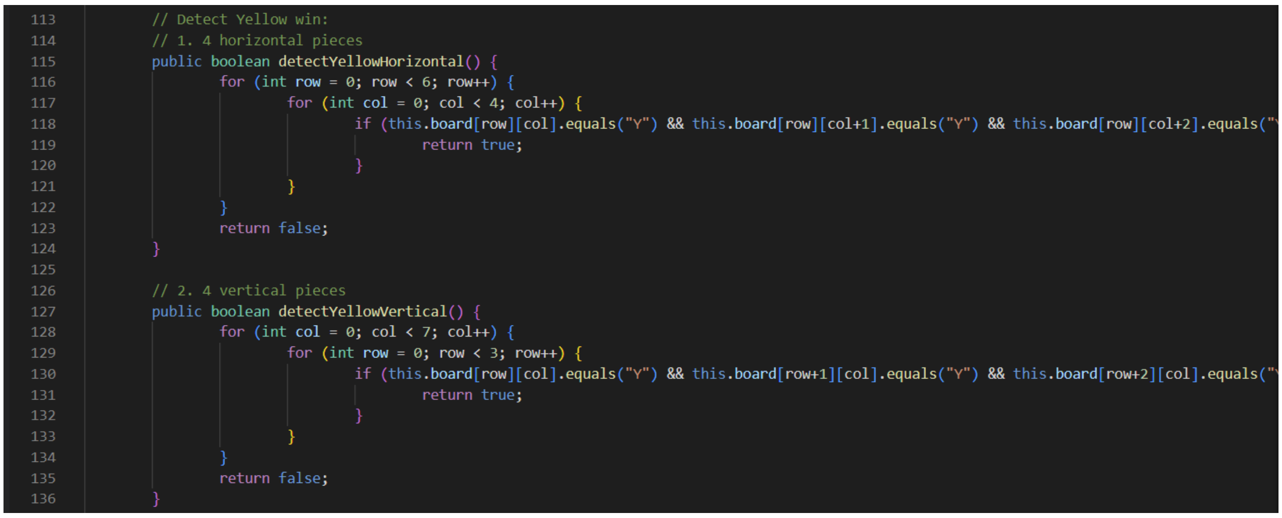


```java
    // Detect an illegal column number --> returns true if detected
    public boolean detectIllegalColumnNumber(int usermove) {
        if (usermove < 0 || usermove > 7) {
            return true;
        }
        else {
            return false;
        }
    }

    // Detect a full column --> returns true if detected
    public boolean detectFullColumn(int usermove) {
        if (!this.board[0][usermove-1].equals(".")) {
            return true;
        }
        else {
            return false;
        }
    }
```

Figure 5: The *detectIllegalColumnNumber* and *detectFullColumn* methods

Note that in *detectFullColumn*, I did not check if the *usermove* is legal, i.e., $0 \leq usermove \leq 7$, and in *placePiece*, I did not check if the inputted column number is legal or if the column is full. The reason is I don't have to, thanks to the order of the *if… else if… else…* statement in the *playGame* method (Figure 2.1, l. 26, l. 29 and l. 33). Indeed, the first thing the program checks is if the user inputted an illegal column number, using the *detectIllegalColumnNumber* method. If the method does not return true, then we can safely assume that $0 \leq usermove \leq 7$ before checking if the column is full, using the *detectFullColumn*. If this method also returns false, then we're sure that the column is not full. Thus, checking it again would be redundant.

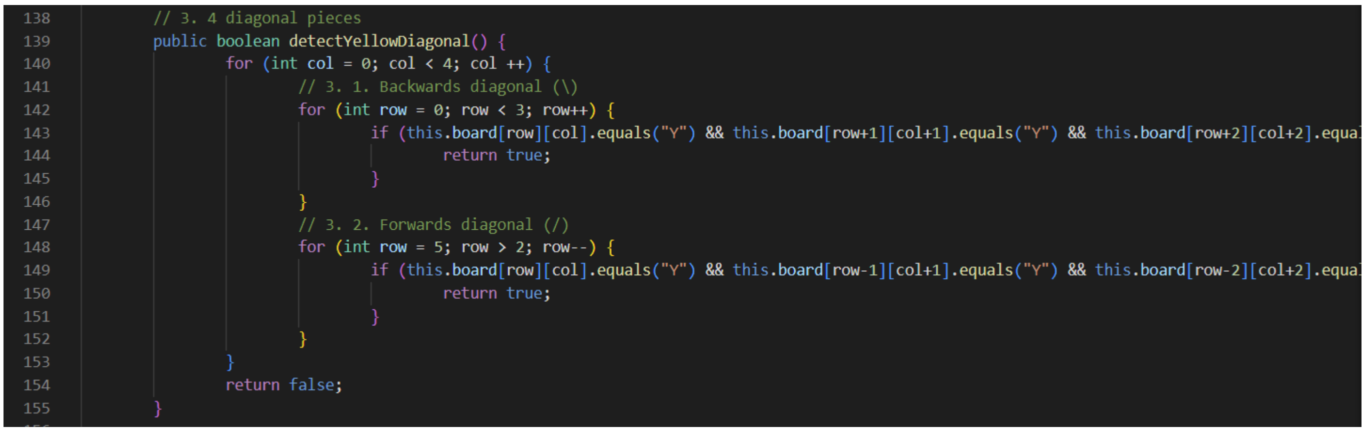
The next three methods are to detect a horizontal, vertical, or diagonal winning move for the yellow player: *detectYellowHorizontal*, *detectYellowVertical*, and *detectYellowDiagonal*. The corresponding methods for the red player, *detectRedHorizontal*, *detectRedVertical*, and *detectRedDiagonal*, are nearly identical, so I won't be describing them here. Just like the previous *detect* methods, these are Booleans, they don't take any arguments, and they return true if they detect a yellow (or red) vertical, horizontal, or diagonal winning move.
detectYellowHorizontal uses two *for* loops, one nested in another, to cycle through the board: the first one cycles through every row, and the second one cycles only through the first four columns. For every column, we check if that column and the next three columns at the right are all equal to "Y": if that's the case, we return true. If we did not find such a case after the two *for* loops, the method returns false.
Similarly, *detectYellowVertical* uses two *for* loops to cycle through every column, and through the first three rows. For every entry cycled through, the algorithm checks if this entry and the three entries "above" it are all equal to "Y": if so, the method returns true. If not, it returns false.



```java
    // Detect Yellow win:
    // 1. 4 horizontal pieces
    public boolean detectYellowHorizontal() {
        for (int row = 0; row < 6; row++) {
            for (int col = 0; col < 4; col++) {
                if (this.board[row][col].equals("Y") && this.board[row][col+1].equals("Y") && this.board[row][col+2].equals("
                    return true;
                }
            }
        }
        return false;
    }

    // 2. 4 vertical pieces
    public boolean detectYellowVertical() {
        for (int col = 0; col < 7; col++) {
            for (int row = 0; row < 3; row++) {
                if (this.board[row][col].equals("Y") && this.board[row+1][col].equals("Y") && this.board[row+2][col].equals("
                    return true;
                }
            }
        }
        return false;
    }
```

Figure 6.1: The *detectYellowHorizontal* and *detectYellowVertical* methods

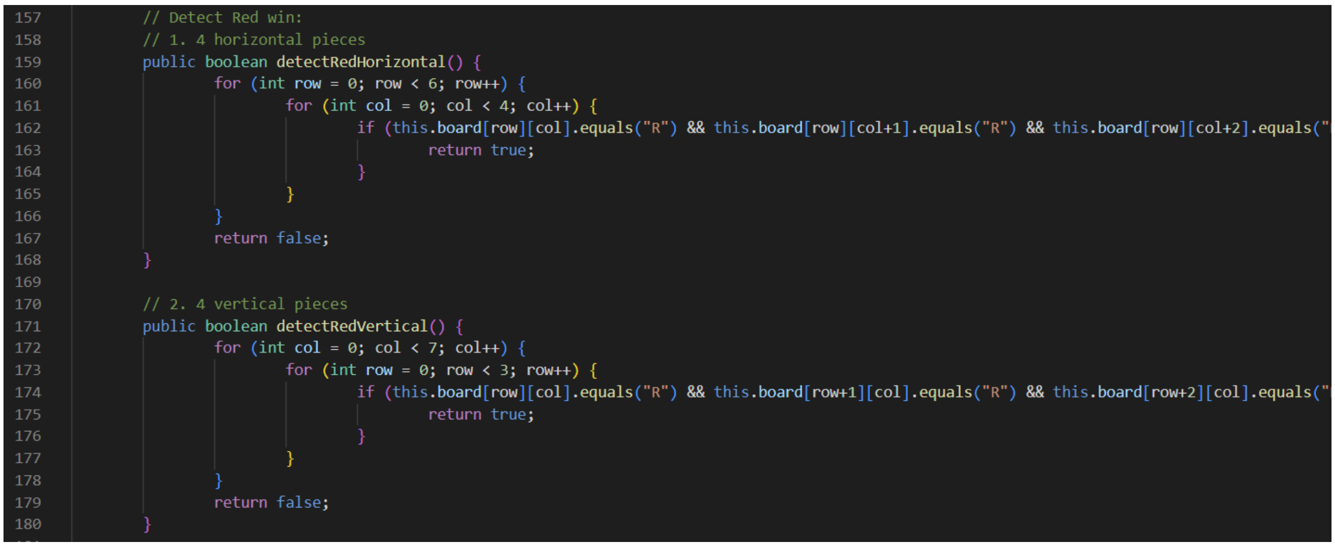
detectYellowDiagonal is made of two parts: one part to check "backwards diagonals" and the other to check "forwards diagonals". Both parts are in the same first *for* loop that cycles through the first 4 columns. For each column, the first part uses a second *for* loop to cycle through the top 3 rows, whereas the second part uses the second *for* loop to cycle through the bottom 3 rows.



```
    // 3. 4 diagonal pieces
    public boolean detectYellowDiagonal() {
        for (int col = 0; col < 4; col ++) {
            // 3. 1. Backwards diagonal (\)
            for (int row = 0; row < 3; row++) {
                if (this.board[row][col].equals("Y") && this.board[row+1][col+1].equals("Y") && this.board[row+2][col+2].equa
                    return true;
                }
            }
            // 3. 2. Forwards diagonal (/)
            for (int row = 5; row > 2; row--) {
                if (this.board[row][col].equals("Y") && this.board[row-1][col+1].equals("Y") && this.board[row-2][col+2].equa
                    return true;
                }
            }
        }
        return false;
    }
```

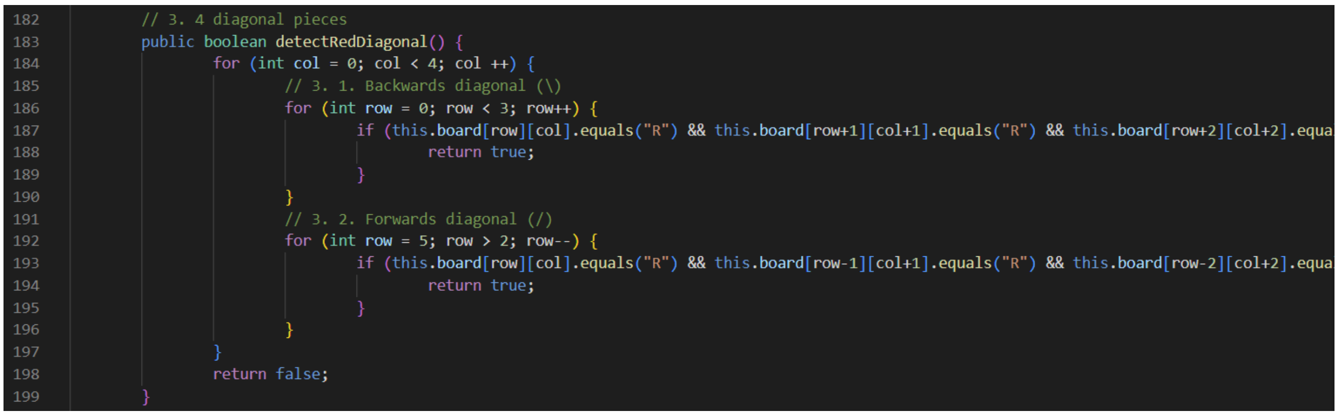
Figure 6.2: The *detectYellowDiagonal* method

The *detectRedHorizontal*, *detectRedVertical*, and *detectRedDiagonal* are extremely similar to the previously explained methods, so I'll just show the methods.



```
    // Detect Red win:
    // 1. 4 horizontal pieces
    public boolean detectRedHorizontal() {
        for (int row = 0; row < 6; row++) {
            for (int col = 0; col < 4; col++) {
                if (this.board[row][col].equals("R") && this.board[row][col+1].equals("R") && this.board[row][col+2].equals("
                    return true;
                }
            }
        }
        return false;
    }

    // 2. 4 vertical pieces
    public boolean detectRedVertical() {
        for (int col = 0; col < 7; col++) {
            for (int row = 0; row < 3; row++) {
                if (this.board[row][col].equals("R") && this.board[row+1][col].equals("R") && this.board[row+2][col].equals("
                    return true;
                }
            }
        }
        return false;
    }
```

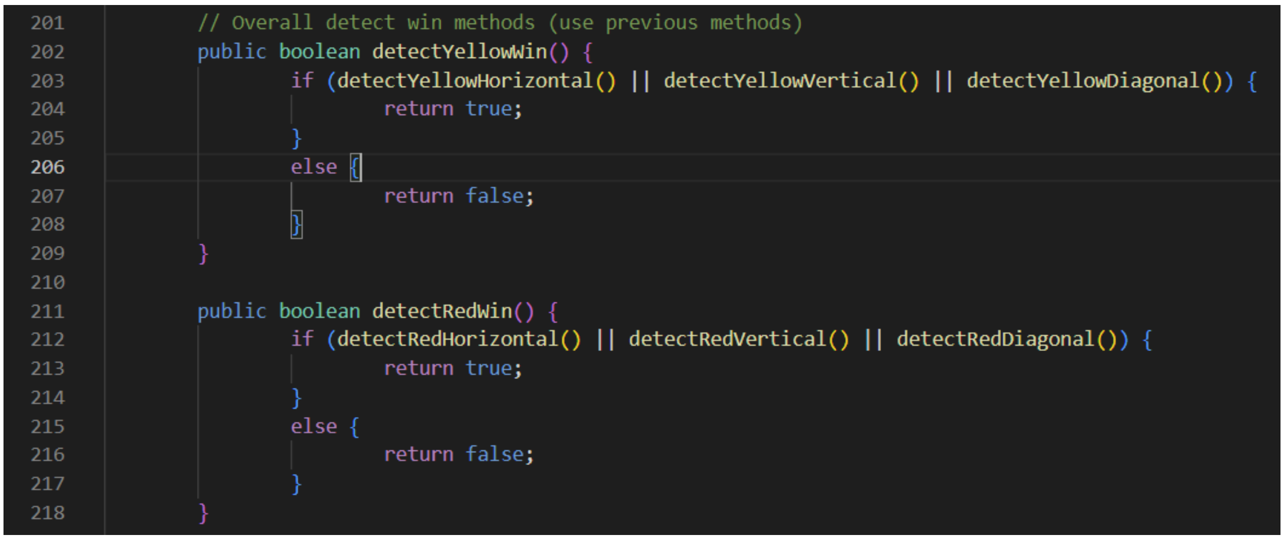
Figure 7.1: The *detectRedVertical* and *detectRedHorizontal* methods



```
    // 3. 4 diagonal pieces
    public boolean detectRedDiagonal() {
        for (int col = 0; col < 4; col ++) {
            // 3. 1. Backwards diagonal (\)
            for (int row = 0; row < 3; row++) {
                if (this.board[row][col].equals("R") && this.board[row+1][col+1].equals("R") && this.board[row+2][col+2].equa
                    return true;
                }
            }
            // 3. 2. Forwards diagonal (/)
            for (int row = 5; row > 2; row--) {
                if (this.board[row][col].equals("R") && this.board[row-1][col+1].equals("R") && this.board[row-2][col+2].equa
                    return true;
                }
            }
        }
        return false;
    }
```

Figure 7.2: The *detectRedDiagonal* method

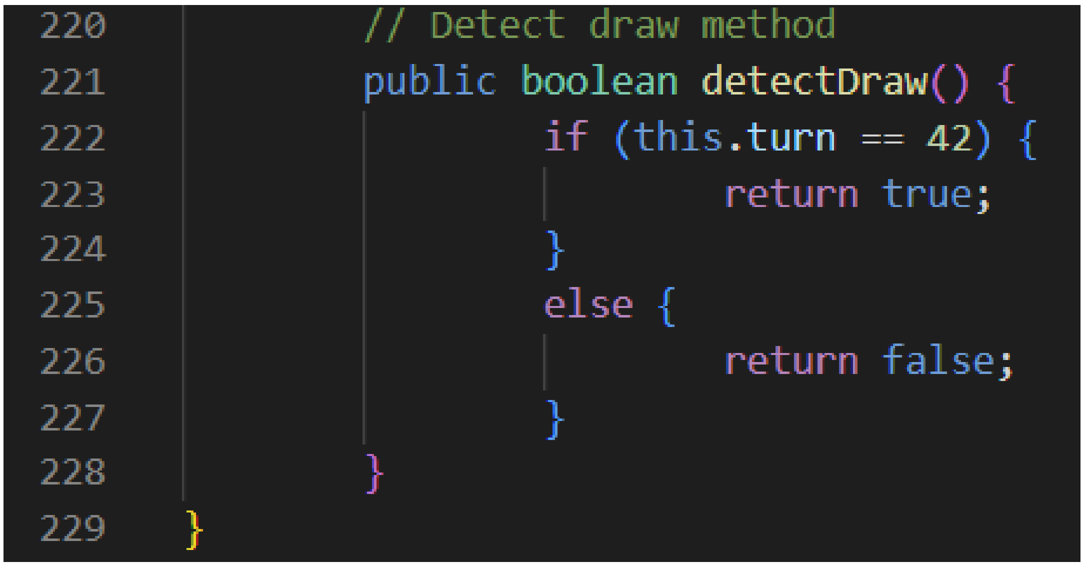
The next two methods, *detectYellowWin* and *detectRedWin*, are used to make the code nicer: they are purely decorative, as they use the previously established methods. They are made of one *if… else…* statement: if either the *detectYellowHorizontal*, *detectYellowVertical* or *detectYellowDiagonal* return true, then *detectYellowWin* returns true. Otherwise, it returns false.



```java
        // Overall detect win methods (use previous methods)
        public boolean detectYellowWin() {
                if (detectYellowHorizontal() || detectYellowVertical() || detectYellowDiagonal()) {
                        return true;
                }
                else {
                        return false;
                }
        }

        public boolean detectRedWin() {
                if (detectRedHorizontal() || detectRedVertical() || detectRedDiagonal()) {
                        return true;
                }
                else {
                        return false;
                }
        }
```

Figure 8: The *detectYellowWin* and *detectRedWin* methods

Finally, the last (and important) method of the Game.java file is *detectDraw*. As its name indicates, it returns true if the game ends with a draw, and false if not. The game ends in a draw if the board is full (and, of course, no one won), i.e., at the 42nd turn. This is why, in the *detectDraw* method, I used an *if... else...* statement to check if the turn meter is equal to 42.

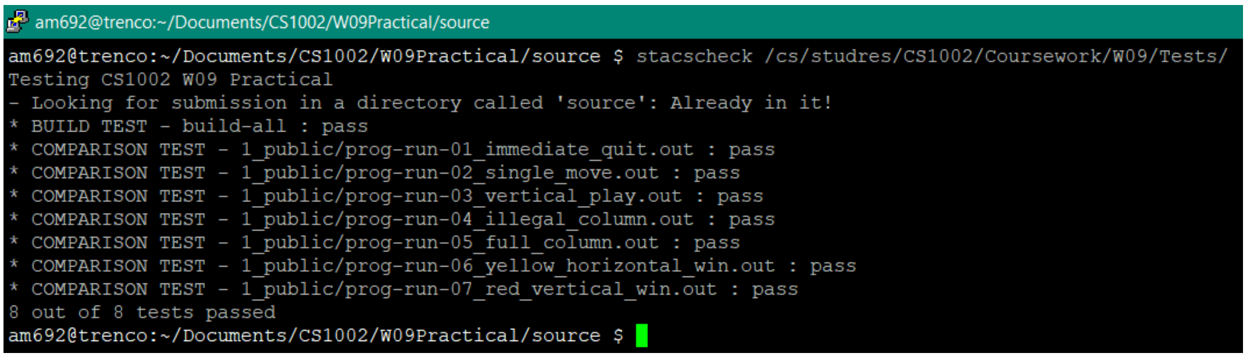


```java
        // Detect draw method
        public boolean detectDraw() {
                if (this.turn == 42) {
                        return true;
                }
                else {
                        return false;
                }
        }
}
```

Figure 9: The *detectDraw* method, and the end of the *Game* file

Testing

As briefly explained in the **Overview**, I've performed two types of tests to make sure that my programs worked. The first type of checks was mandatory: those are the *stacscheck* tests. To run these tests, I had to type in the *stacscheck /cs/studres/CS1002/Coursework/W09/Tests/* command in the Linux terminal. As you can see in the Figure 10, my programs have passed all 8 *stacscheck* tests:

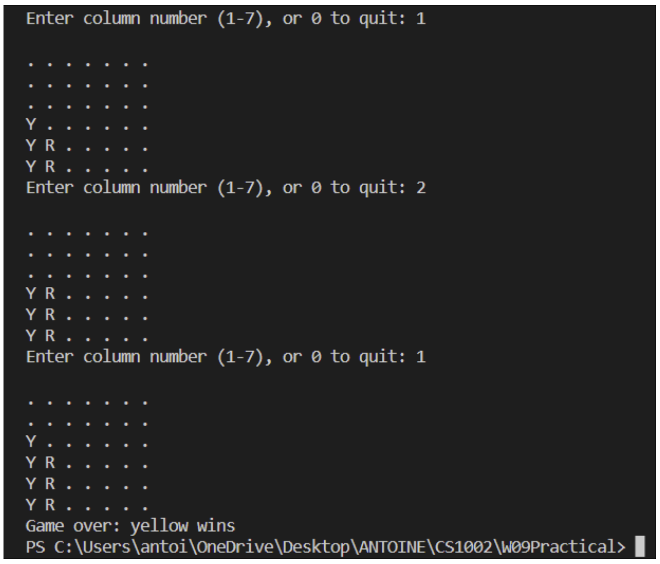


```
am692@trenco:~/Documents/CS1002/W09Practical/source
am692@trenco:~/Documents/CS1002/W09Practical/source $ stacscheck /cs/studres/CS1002/Coursework/W09/Tests/
Testing CS1002 W09 Practical
- Looking for submission in a directory called 'source': Already in it!
* BUILD TEST - build-all : pass
* COMPARISON TEST - 1_public/prog-run-01_immediate_quit.out : pass
* COMPARISON TEST - 1_public/prog-run-02_single_move.out : pass
* COMPARISON TEST - 1_public/prog-run-03_vertical_play.out : pass
* COMPARISON TEST - 1_public/prog-run-04_illegal_column.out : pass
* COMPARISON TEST - 1_public/prog-run-05_full_column.out : pass
* COMPARISON TEST - 1_public/prog-run-06_yellow_horizontal_win.out : pass
* COMPARISON TEST - 1_public/prog-run-07_red_vertical_win.out : pass
8 out of 8 tests passed
am692@trenco:~/Documents/CS1002/W09Practical/source $
```

Figure 10: The *stacscheck* tests

However, these tests weren't enough, as they did not cover every single possibility. Therefore, I've manually tested my code to make sure it worked:

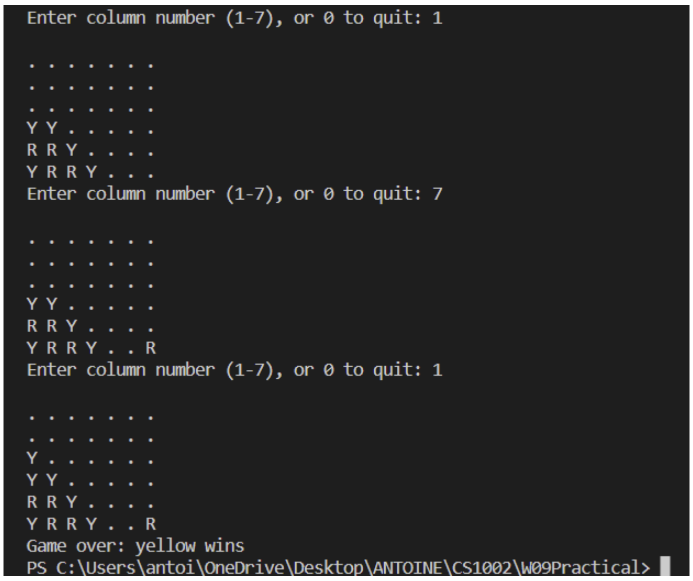
- Testing for a yellow vertical win:



```
Enter column number (1-7), or 0 to quit: 1

. . . . . . .
. . . . . . .
. . . . . . .
Y . . . . . .
Y R . . . . .
Y R . . . . .
Enter column number (1-7), or 0 to quit: 2

. . . . . . .
. . . . . . .
. . . . . . .
Y R . . . . .
Y R . . . . .
Y R . . . . .
Enter column number (1-7), or 0 to quit: 1

. . . . . . .
. . . . . . .
Y . . . . . .
Y R . . . . .
Y R . . . . .
Y R . . . . .
Game over: yellow wins
PS C:\Users\antoi\OneDrive\Desktop\ANTOINE\CS1002\W09Practical>
```

Figure 11.1: Testing for a yellow vertical win

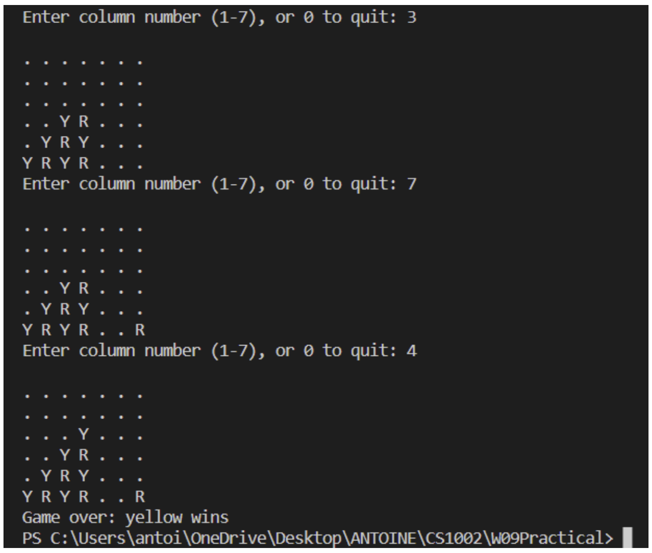
- Testing for a yellow backwards diagonal win:



```
Enter column number (1-7), or 0 to quit: 1

. . . . . . .
. . . . . . .
. . . . . . .
Y Y . . . . .
R R Y . . . .
Y R R Y . . .
Enter column number (1-7), or 0 to quit: 7

. . . . . . .
. . . . . . .
. . . . . . .
Y Y . . . . .
R R Y . . . .
Y R R Y . . R
Enter column number (1-7), or 0 to quit: 1

. . . . . . .
. . . . . . .
Y . . . . . .
Y Y . . . . .
R R Y . . . .
Y R R Y . . R
Game over: yellow wins
PS C:\Users\antoi\OneDrive\Desktop\ANTOINE\CS1002\W09Practical>
```

Figure 11.2: Testing for a yellow backwards diagonal win

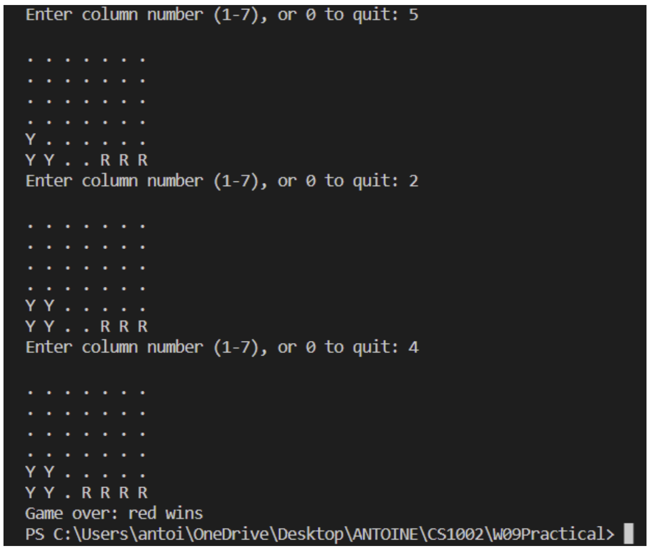
- Testing for a yellow forwards diagonal win:



```
Enter column number (1-7), or 0 to quit: 3

. . . . . . .
. . . . . . .
. . . . . . .
. . Y R . . .
. Y R Y . . .
Y R Y R . . .
Enter column number (1-7), or 0 to quit: 7

. . . . . . .
. . . . . . .
. . . . . . .
. . Y R . . .
. Y R Y . . .
Y R Y R . . R
Enter column number (1-7), or 0 to quit: 4

. . . . . . .
. . . . . . .
. . . Y . . .
. . Y R . . .
. Y R Y . . .
Y R Y R . . R
Game over: yellow wins
PS C:\Users\antoi\OneDrive\Desktop\ANTOINE\CS1002\W09Practical>
```

Figure 11.3: Testing for a yellow forwards diagonal win

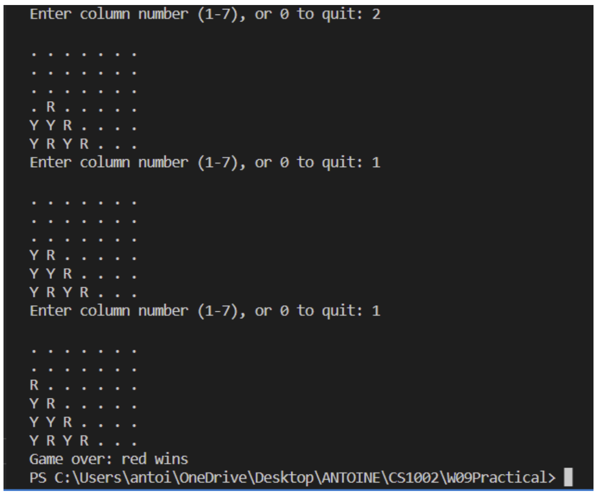
- Testing for a red horizontal win:



```
Enter column number (1-7), or 0 to quit: 5

. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
Y . . . . . .
Y Y . . R R R
Enter column number (1-7), or 0 to quit: 2

. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
Y Y . . . . .
Y Y . . R R R
Enter column number (1-7), or 0 to quit: 4

. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
Y Y . . . . .
Y Y . R R R R
Game over: red wins
PS C:\Users\antoi\OneDrive\Desktop\ANTOINE\CS1002\W09Practical>
```

Figure 11.4: Testing for a red horizontal win

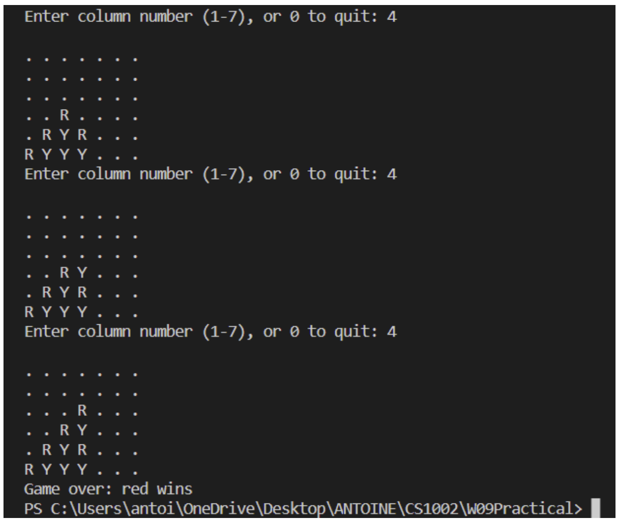
- Testing for a red backwards diagonal win:



```
Enter column number (1-7), or 0 to quit: 2

. . . . . . .
. . . . . . .
. . . . . . .
. R . . . . .
Y Y R . . . .
Y R Y R . . .
Enter column number (1-7), or 0 to quit: 1

. . . . . . .
. . . . . . .
. . . . . . .
Y R . . . . .
Y Y R . . . .
Y R Y R . . .
Enter column number (1-7), or 0 to quit: 1

. . . . . . .
. . . . . . .
R . . . . . .
Y R . . . . .
Y Y R . . . .
Y R Y R . . .
Game over: red wins
PS C:\Users\antoi\OneDrive\Desktop\ANTOINE\CS1002\W09Practical>
```

Figure 11.5: Testing for a red backwards diagonal win

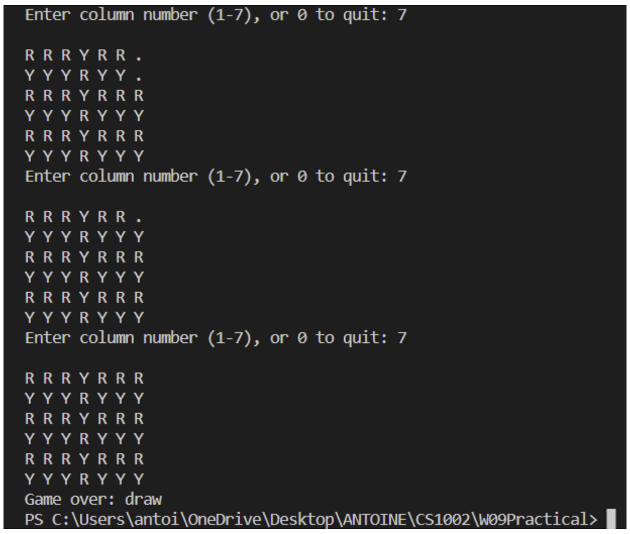
- Testing for a red forwards diagonal win:



```
Enter column number (1-7), or 0 to quit: 4

. . . . . . .
. . . . . . .
. . . . . . .
. . R . . . .
. R Y R . . .
R Y Y Y . . .
Enter column number (1-7), or 0 to quit: 4

. . . . . . .
. . . . . . .
. . . . . . .
. . R Y . . .
. R Y R . . .
R Y Y Y . . .
Enter column number (1-7), or 0 to quit: 4

. . . . . . .
. . . . . . .
. . . R . . .
. . R Y . . .
. R Y R . . .
R Y Y Y . . .
Game over: red wins
PS C:\Users\antoi\OneDrive\Desktop\ANTOINE\CS1002\W09Practical>
```

Figure 11.6: Testing for a red forwards diagonal win

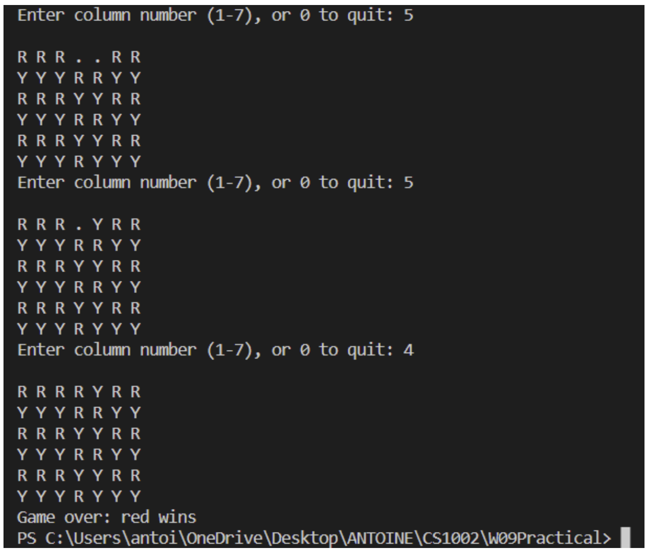
- Testing for a draw:



```
Enter column number (1-7), or 0 to quit: 7

R R R Y R R .
Y Y Y R Y Y .
R R R Y R R R
Y Y Y R Y Y Y
R R R Y R R R
Y Y Y R Y Y Y
Enter column number (1-7), or 0 to quit: 7

R R R Y R R .
Y Y Y R Y Y Y
R R R Y R R R
Y Y Y R Y Y Y
R R R Y R R R
Y Y Y R Y Y Y
Enter column number (1-7), or 0 to quit: 7

R R R Y R R R
Y Y Y R Y Y Y
R R R Y R R R
Y Y Y R Y Y Y
R R R Y R R R
Y Y Y R Y Y Y
Game over: draw
PS C:\Users\antoi\OneDrive\Desktop\ANTOINE\CS1002\W09Practical>
```

Figure 11.7: Testing for a draw

- Testing for a win at the 42nd move:



```
Enter column number (1-7), or 0 to quit: 5

R R R . . R R
Y Y Y R R Y Y
R R R Y Y R R
Y Y Y R R Y Y
R R R Y Y R R
Y Y Y R Y Y Y
Enter column number (1-7), or 0 to quit: 5

R R R . Y R R
Y Y Y R R Y Y
R R R Y Y R R
Y Y Y R R Y Y
R R R Y Y R R
Y Y Y R Y Y Y
Enter column number (1-7), or 0 to quit: 4

R R R R Y R R
Y Y Y R R Y Y
R R R Y Y R R
Y Y Y R R Y Y
R R R Y Y R R
Y Y Y R Y Y Y
Game over: red wins
PS C:\Users\antoi\OneDrive\Desktop\ANTOINE\CS1002\W09Practical>
```

Figure 11.8: Testing for a win at the 42nd move

Evaluation

I am proud to say that my program works: it runs a game of Connect Four, played by two human players. To make sure that my code is functional, I used a variety of tests, some compulsory, some optional.

Conclusion

All in all, I found this W09Practical not as challenging as the first one. Yes, there were more methods to code, and we had less help to make sure our program works. But I found this assignment more “straightforward” than the previous one, as the output and the format of the code was less of a challenge than the W05Practical. This also being the second Practical of the CS1002 module, I was already accustomed in some way to what was expected.

If I had more time, I would love to make the program input the user more parameters, such as his/her name, what symbol he/she wants to use as pawns, etc. I would also want to input the players, after the game ends, if they want to play again, without running W09Practical.java again. Finally, I would want to add my manual tests to Stacscheck so that it would test my code automatically every time I change it.