CPSC 2150 Project Report

Skylar Hubbarth

Requirements Analysis

Functional Requirements:

1. As a player, I can enter the amount of rows on the board, between 3 and 20, so that the size of the board can be customized.
2. As a player, I can enter the amount of columns on the board, between 3 and 20, so that the size of the board can be customized.
3. As a player, I can enter the number of tokens in a row to win, between 3 and 20, so that I can customize the amount to win.
4. As a player, I can select the number of players from a drop down, between 2 and 10, so that I can choose how many people are playing.
5. As a player, I can select a button of a column so that I can place my token in the spot I desire.
6. As a player, I can play the game again after a round is over so that I can continue playing the game with a new board.
7. As a player, I can exit the game so that I can stop the program and don't have to play any longer.
8. As a player, I can play the game with my friend(s) on one computer so that we can play a game together and see who wins.
9. As a player, I can play the game by myself so that I can test out how the game works.
10. As a player, I can switch player turns, so we can each have a turn.
11. As a player, I can fill up the entire board so that the game will end in a tie.
12. As a player, I can align my tokens in a horizontal line [number to win] tokens long so that I can win the game.
13. As a player, I can align my tokens in a vertical line [number to win] tokens high so that I can win the game.
14. As a player, I can align my tokens in a diagonal line [number to win] tokens long so that I can win the game.
15. As a player, I can place my tokens on top of my opponent's tokens so that I can cut off their tokens and keep them from winning.
16. As a player, I can see where previous tokens were placed on the board so that I can know where I'm able to place my tokens next.

17. As a player, I can see my previous tokens that match my respective player symbol that I chose so that I can connect [number to win] of them and win the game.
18. As a player, I can see my opponent's previous tokens that match their respective player symbol that they chose so that I can see when they are about to win the game and try to stop them.
19. As a player, I can pick again if I pick an unavailable column, so I don't lose my turn.
20. As a player, I can pick again if I pick a column that does not exist, so I don't lose my turn.

Non-Functional Requirements

1. The game will be played in a Java Frame window launched by IntelliJ.
2. The program will be started in IntelliJ.
3. The user cannot click a button outside of the selected number of columns.
4. The user cannot place a token in a column that has been selected [number of rows] times in one game.
5. The user will select their column number by clicking a button on the frame.
6. The program will be written in Java.
7. The board size is of size [number of rows] x [number of columns], with a minimum size of 3x3 and a maximum size of 20x20.
8. The minimum number to win is 3, and the maximum is 20.
9. The number to win is always less than the number of columns and the number of rows.
10. The minimum number of players is 2, and the maximum is 10.
11. Player X always goes first.
12. Position 0,0 is at the bottom left of the game board.
13. The characters are assigned in the following order and are not customizable: {'X', 'O', 'Y', 'Z', 'W', 'A', 'B', 'C', 'D', 'E'}

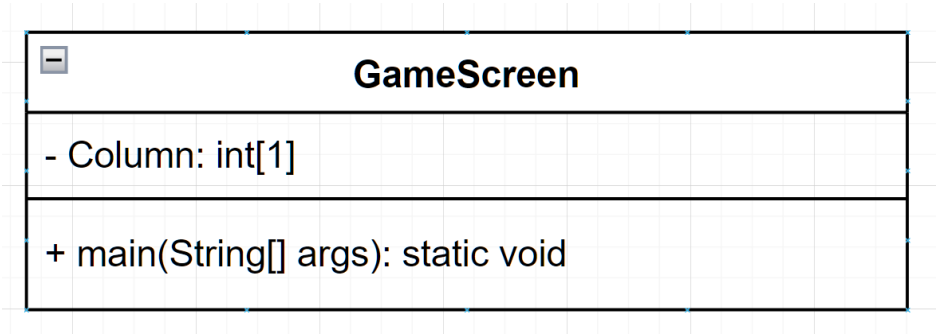
Deployment Instructions

1. Download and extract the zip file.
2. Open the project in IntelliJ Idea.
3. Create a new configuration using the main program ConnectXApp.
4. Hit the play button next to the new configuration.
5. The game setup will open in a separate window.
6. After setting up the game, the board will open in a new window.

System Design

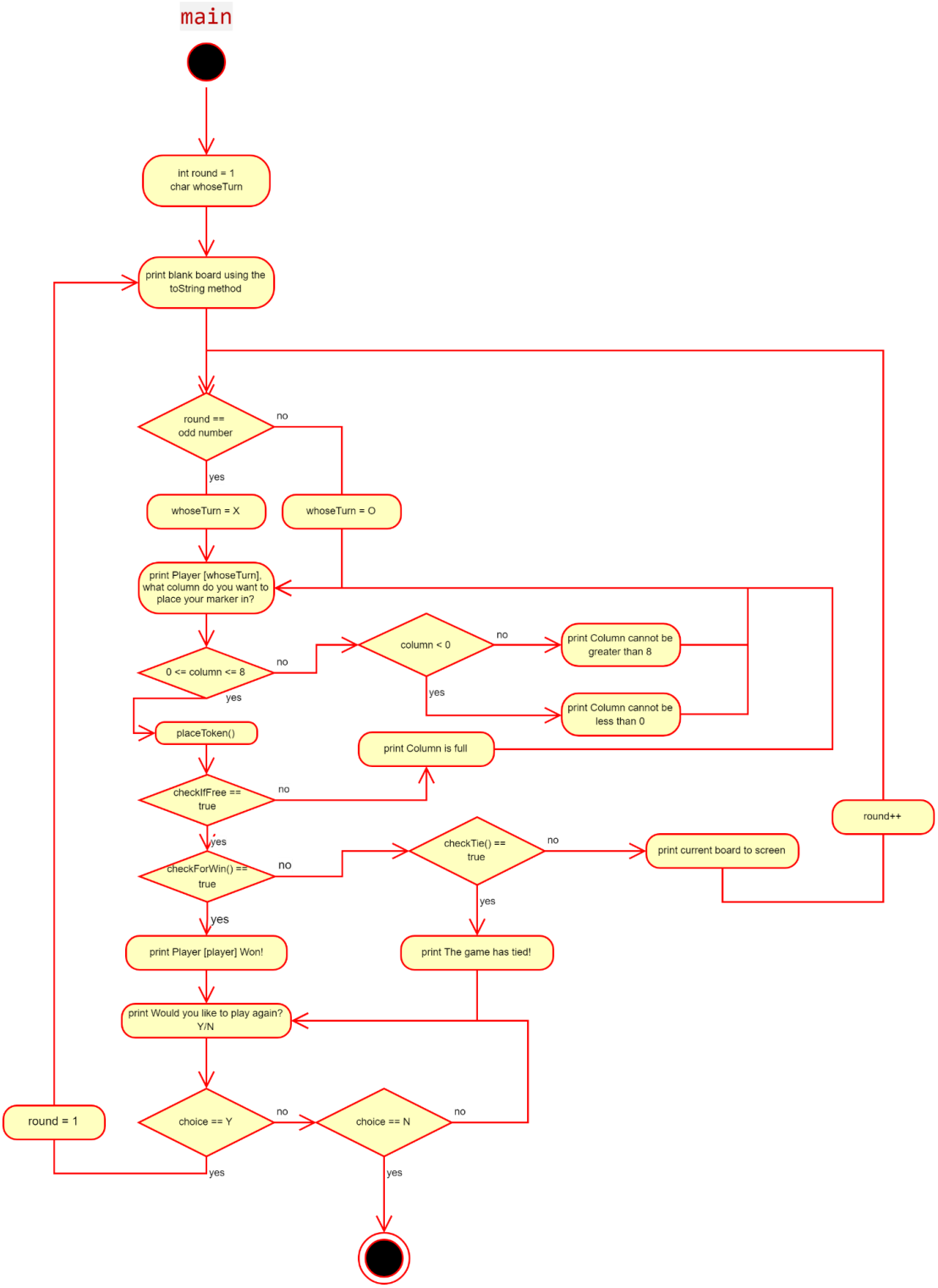
Class 1: GameScreen

Class diagram



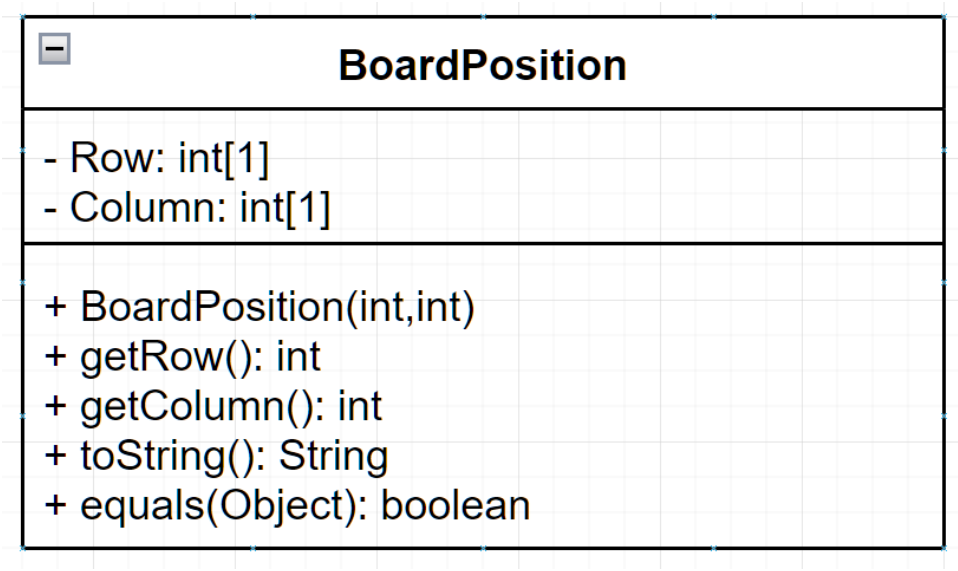
Activity diagrams

`main`



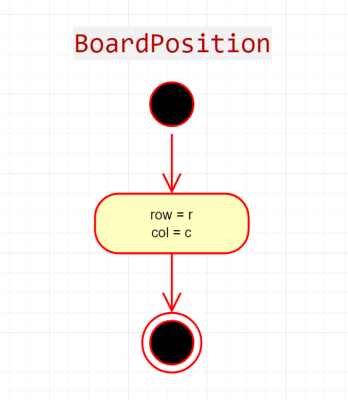
Class 2: `BoardPosition`

Class diagram

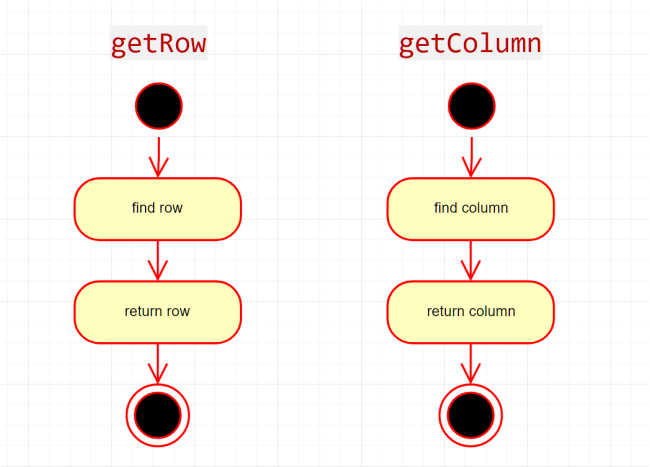


Activity diagrams

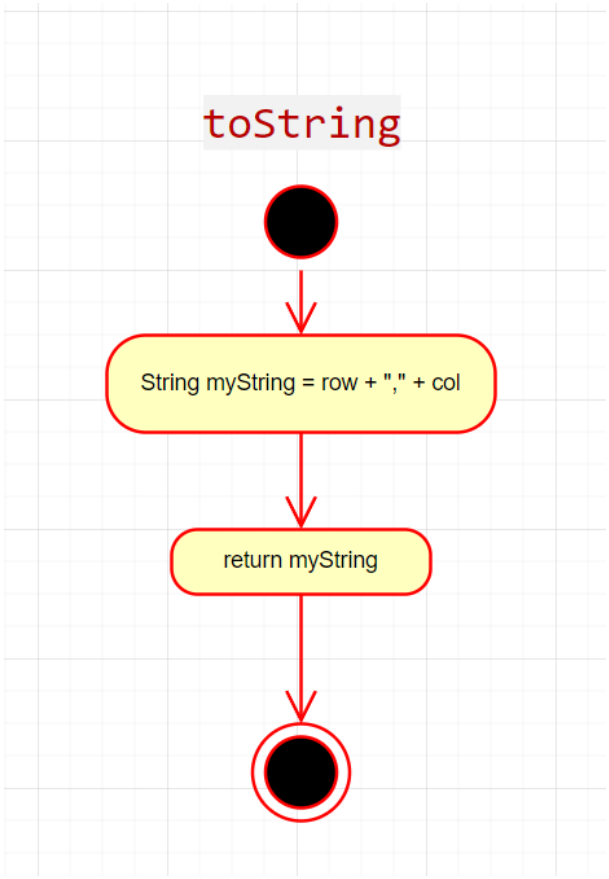
`BoardPosition`



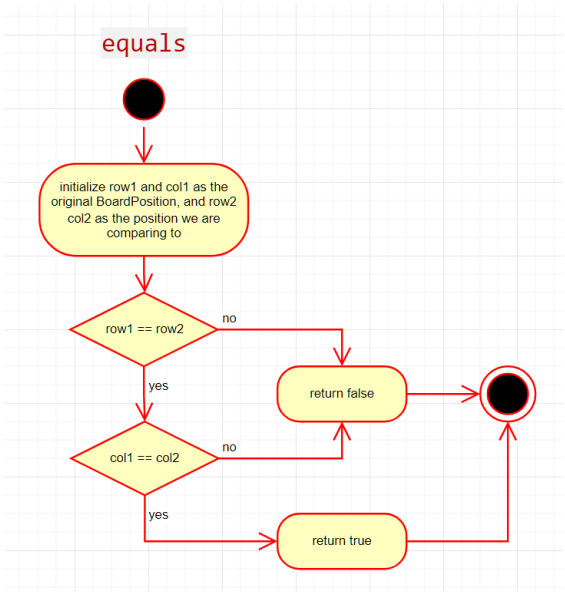
`getRow` `getColumn`



toString

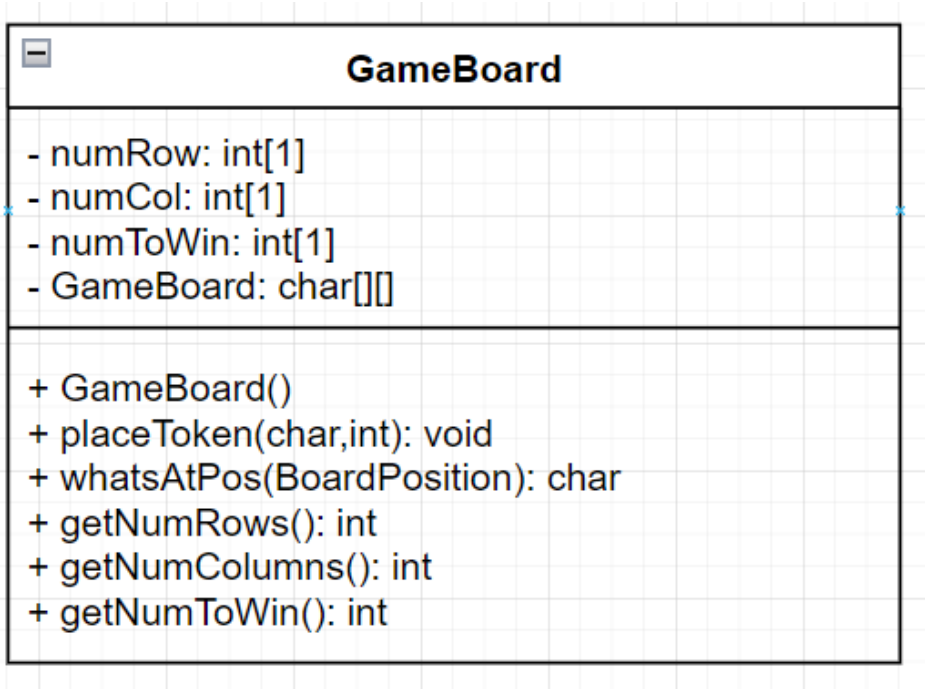


equals



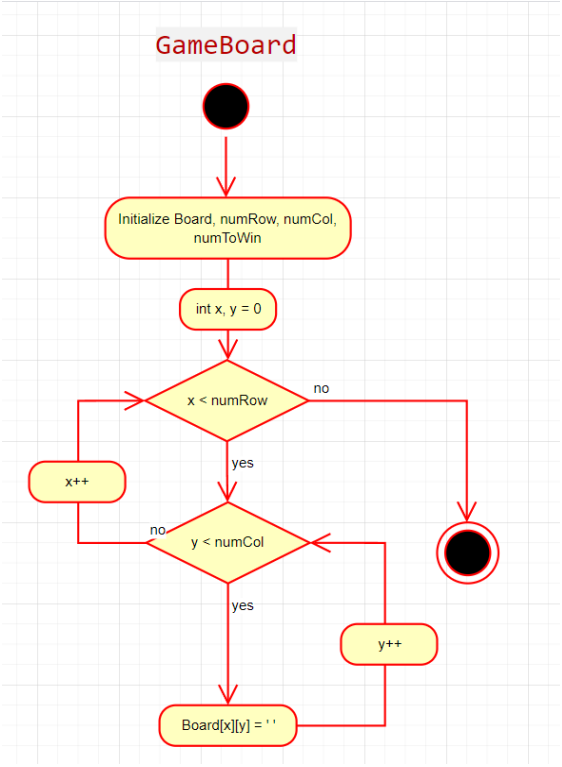
Class 3.0: `GameBoard`

Class diagram

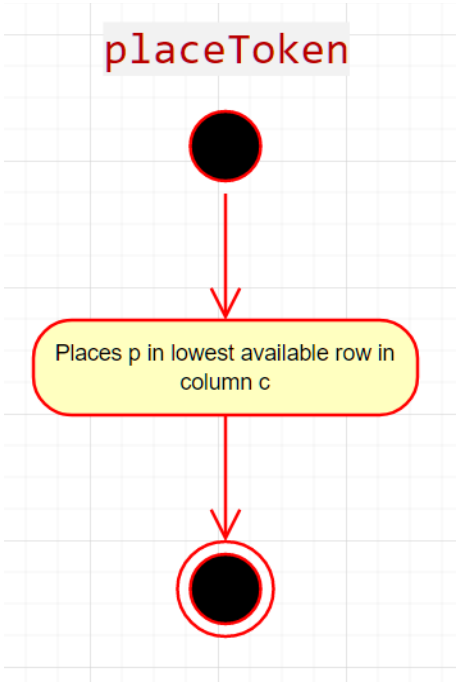


Activity diagrams

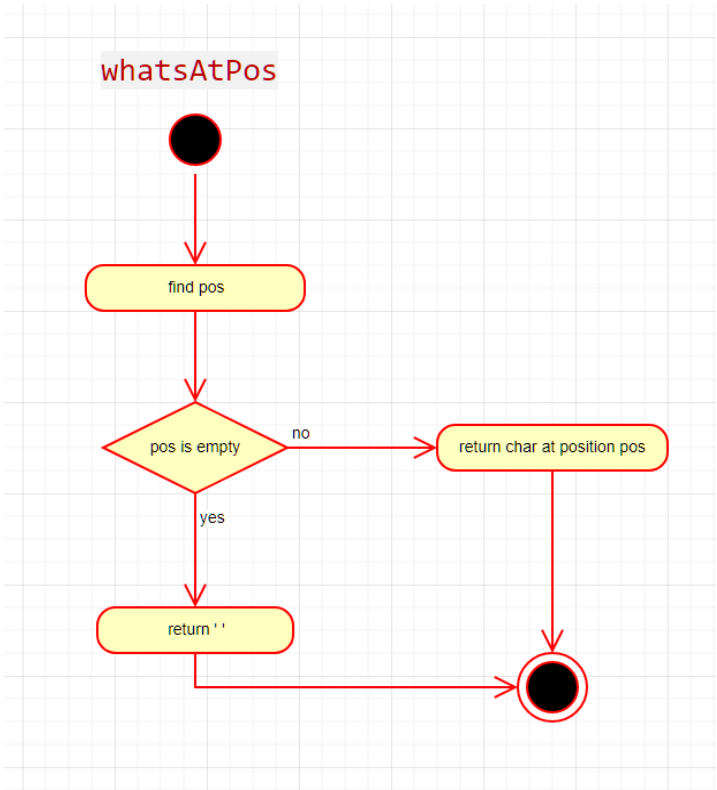
`GameBoard`



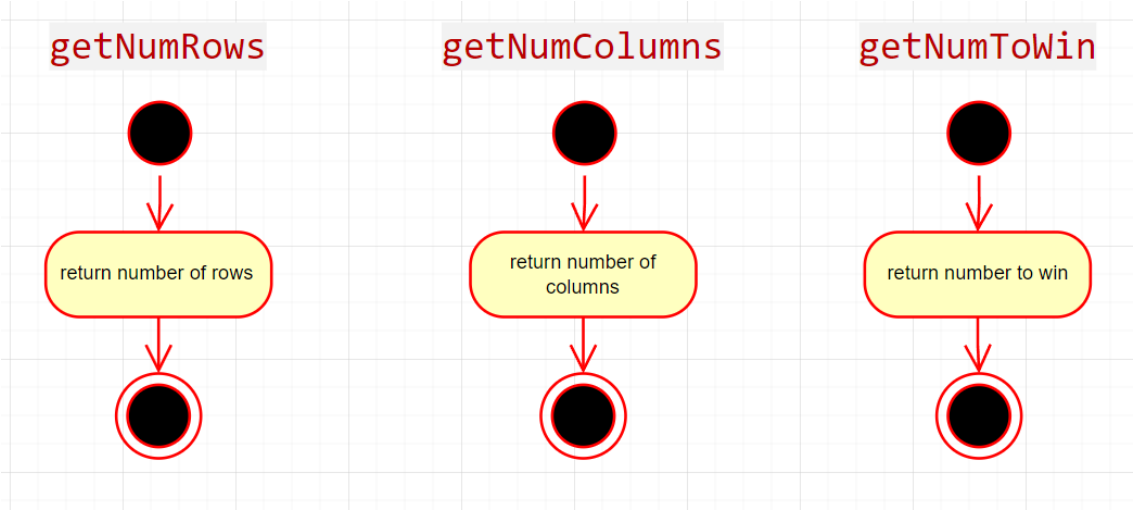
placeToken



whatsAtPos

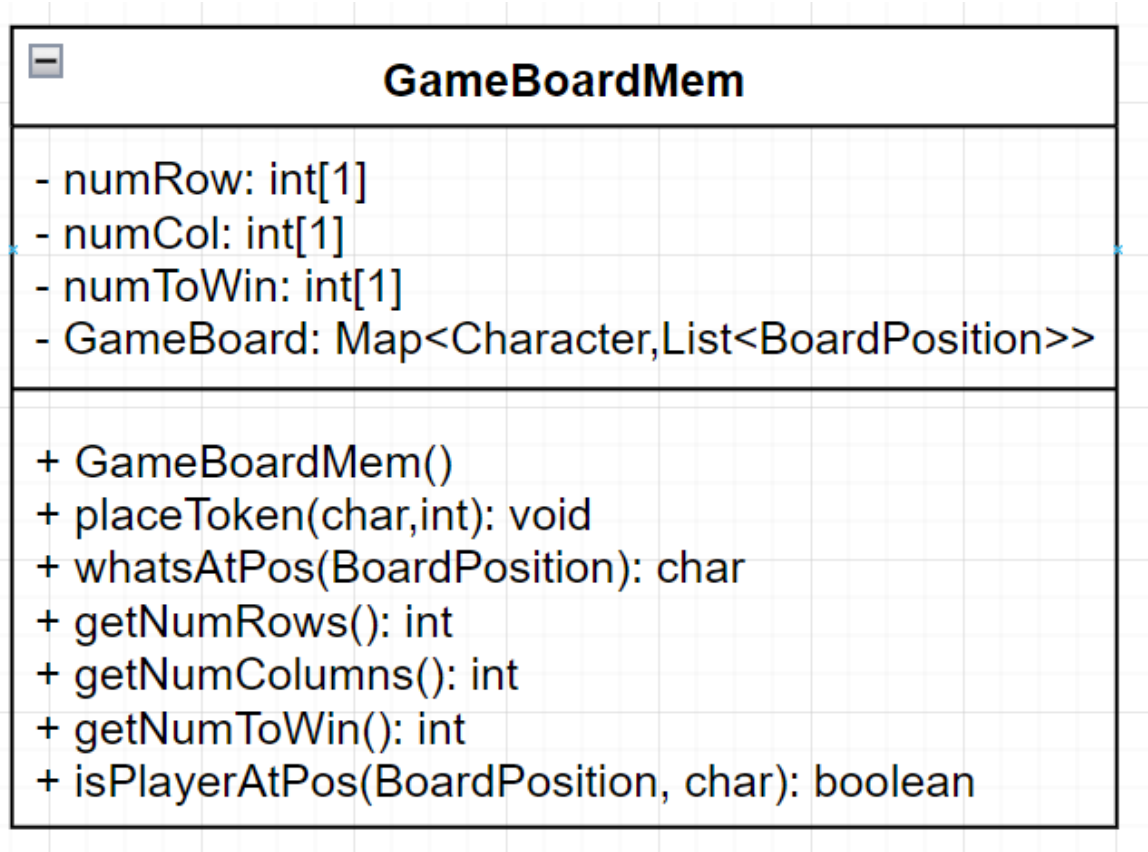


`getNumRows` `getNumColumns` `getNumToWin`



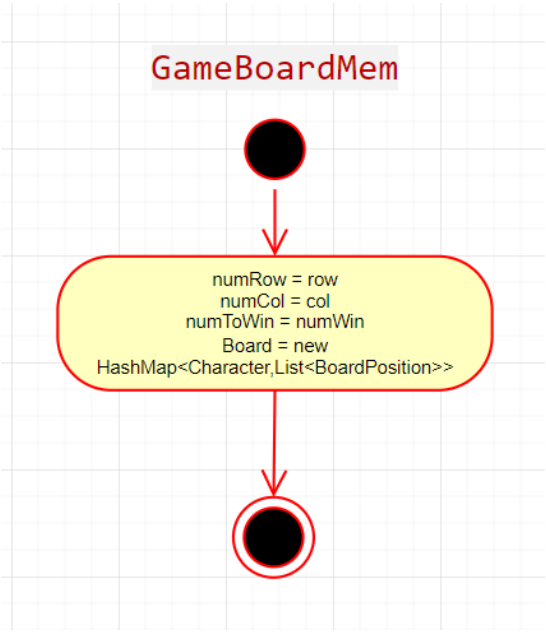
Class 3.1: `GameBoardMem:`

Class diagram

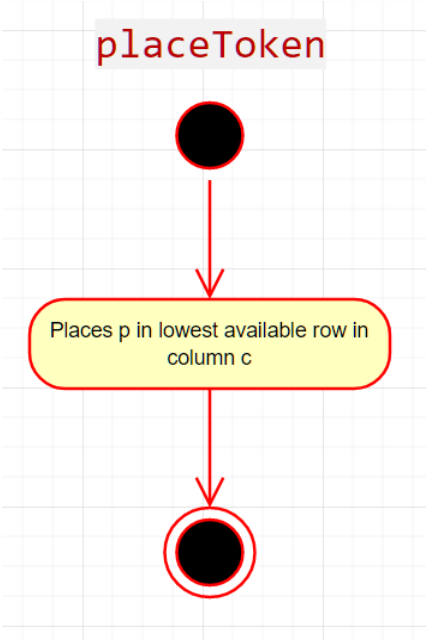


Activity diagrams

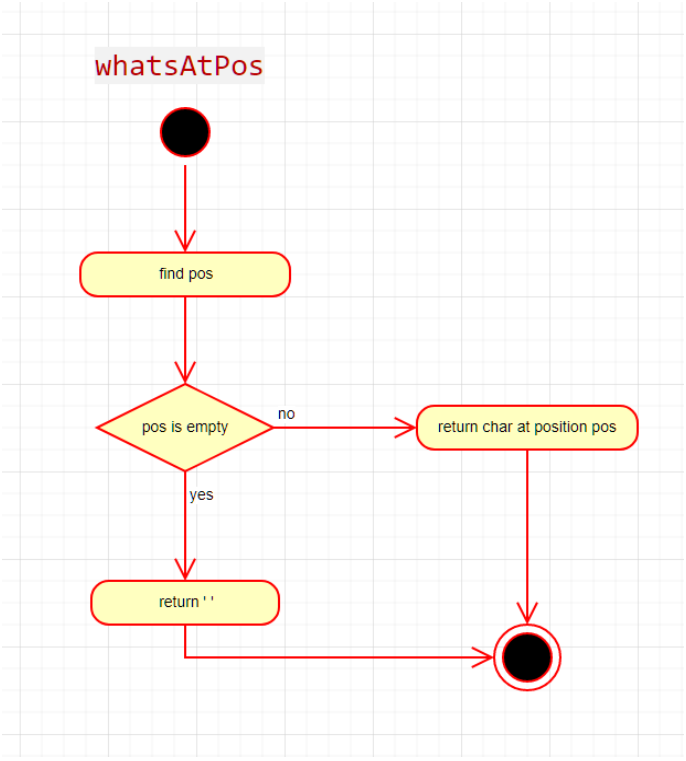
`GameBoardMem`



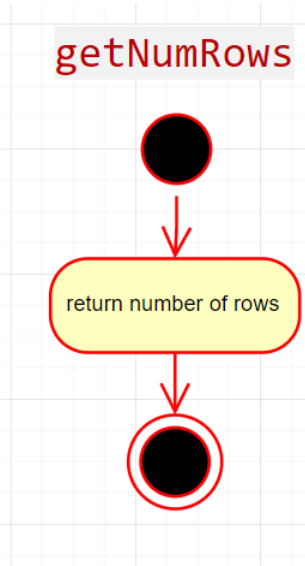
placeToken



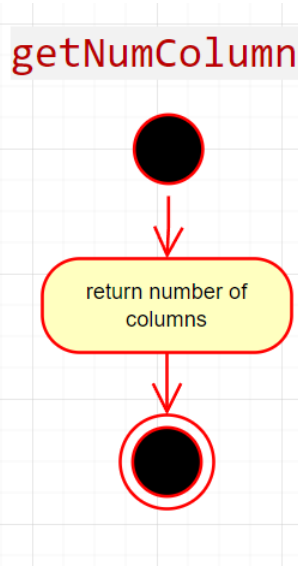
whatsAtPos



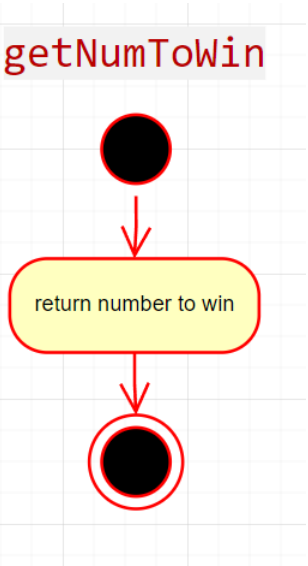
getNumRows



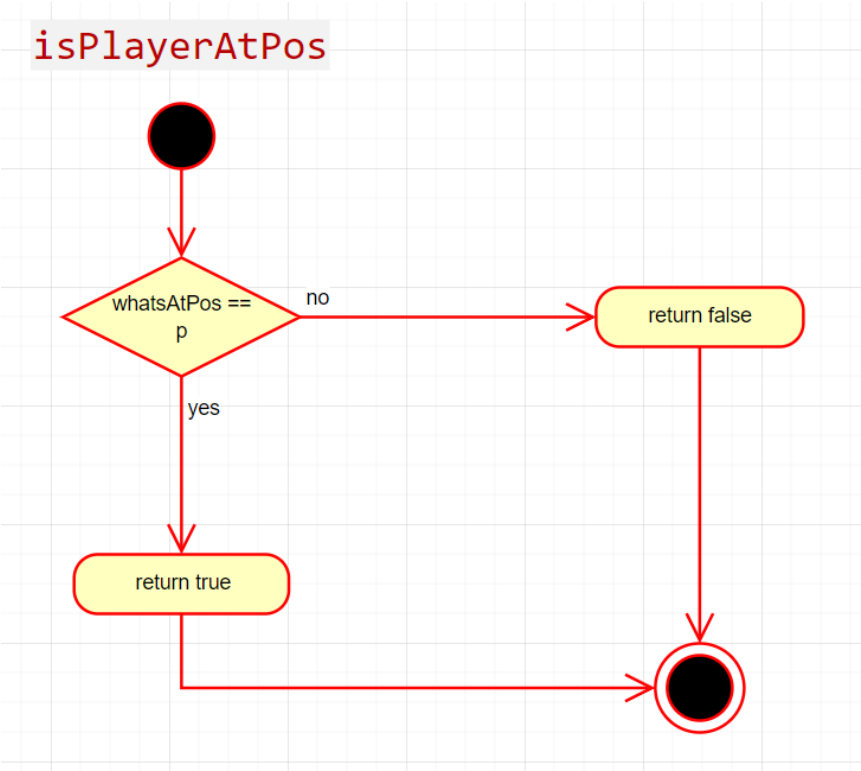
getNumColumns



getNumToWin

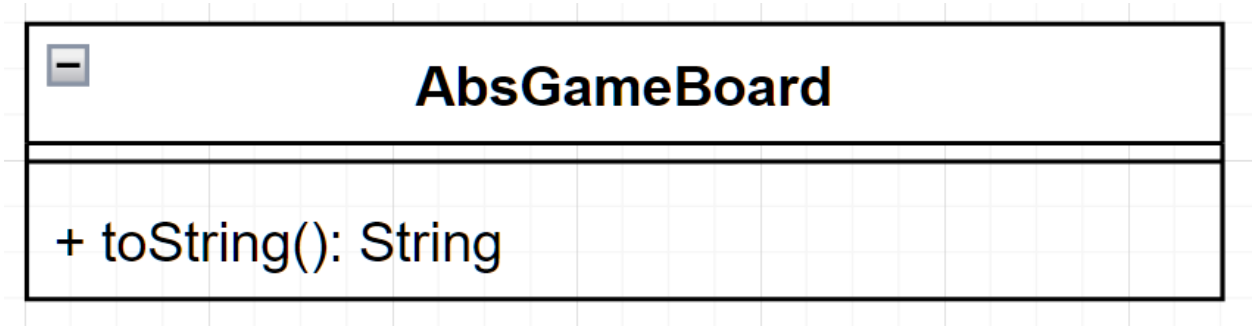


isPlayerAtPos (@Override)



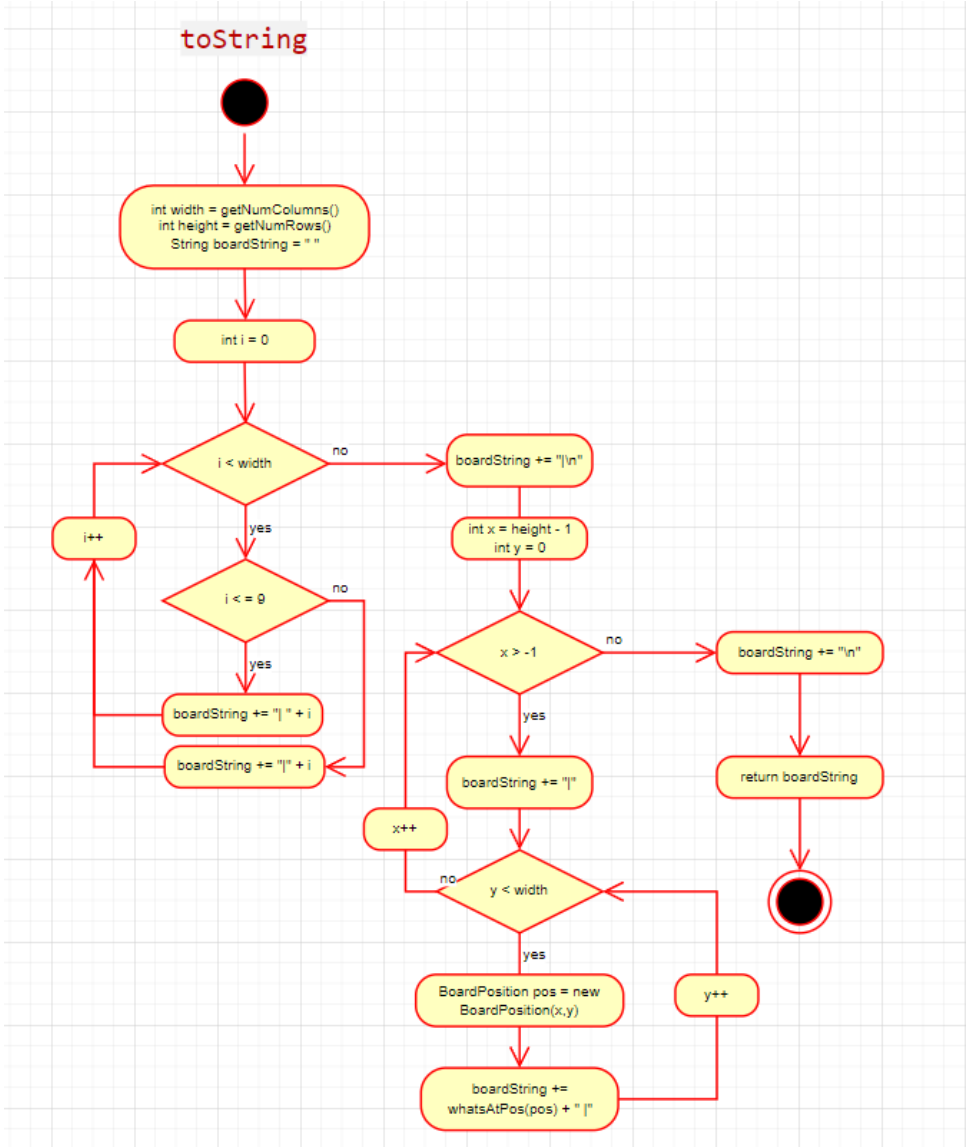
Class 3.2: `AbsGameBoard:`

Class diagram



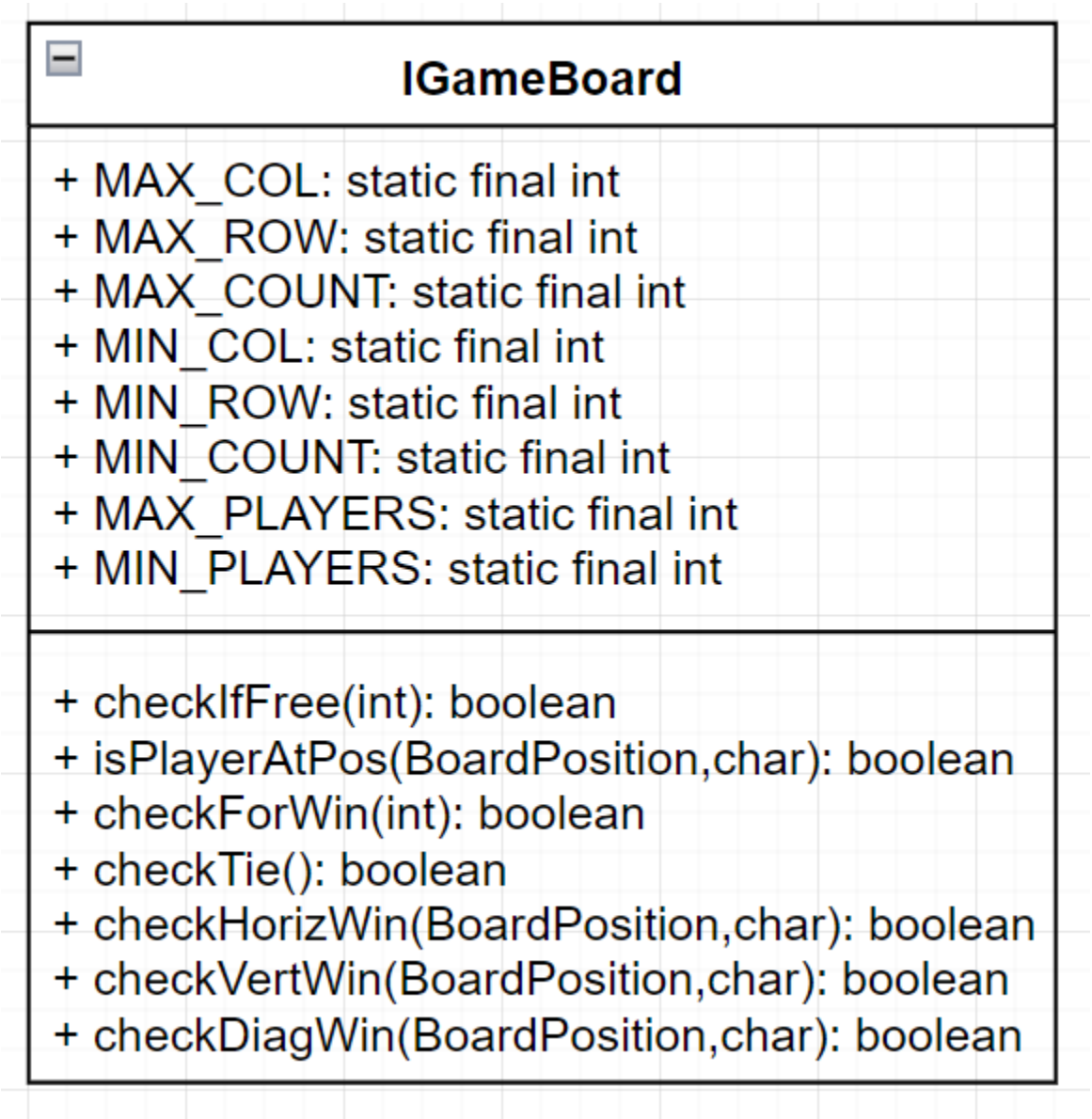
Activity diagrams

`toString`



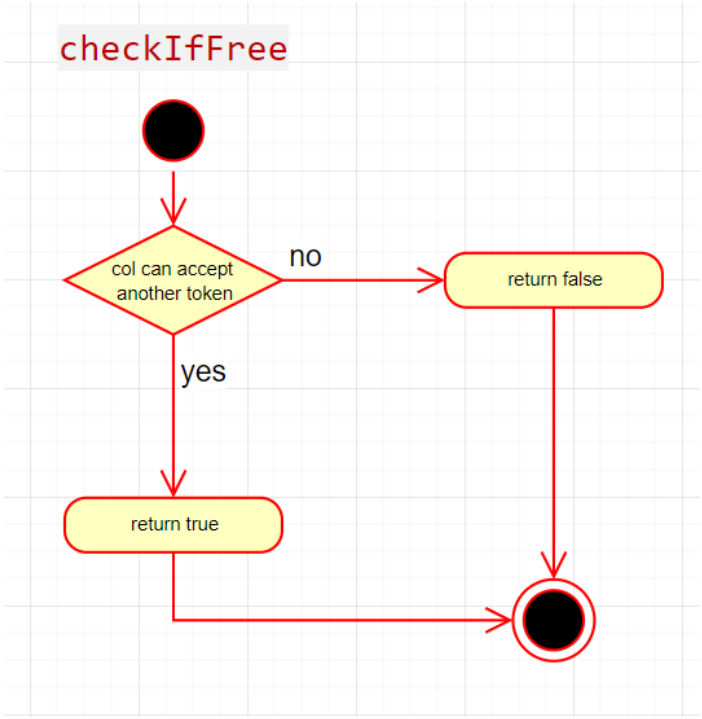
Class 3.3: `IGameBoard:`

Class diagram

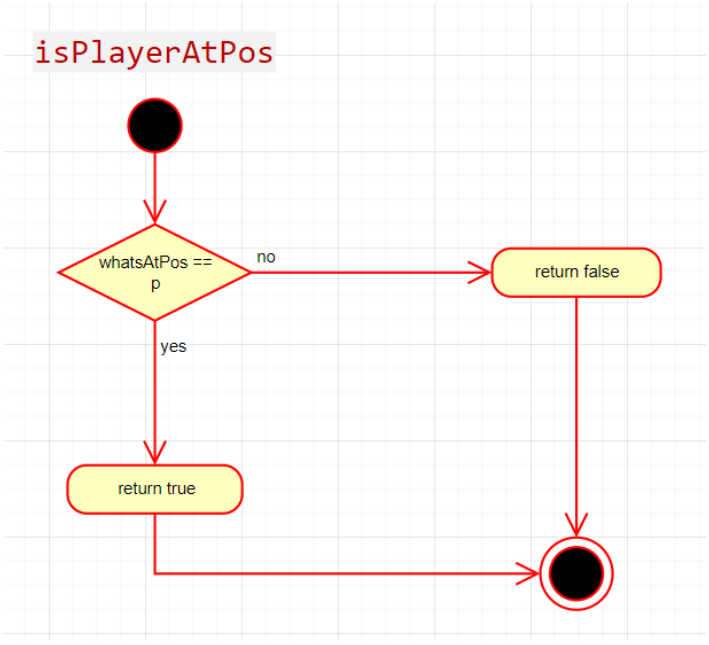


Activity diagrams

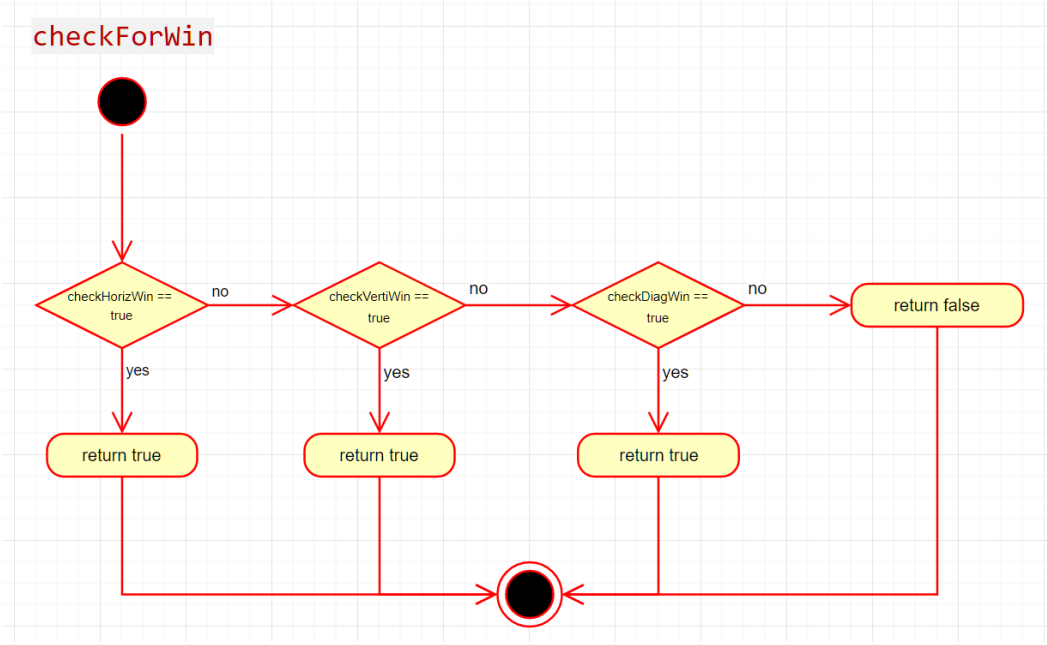
`checkIfFree`



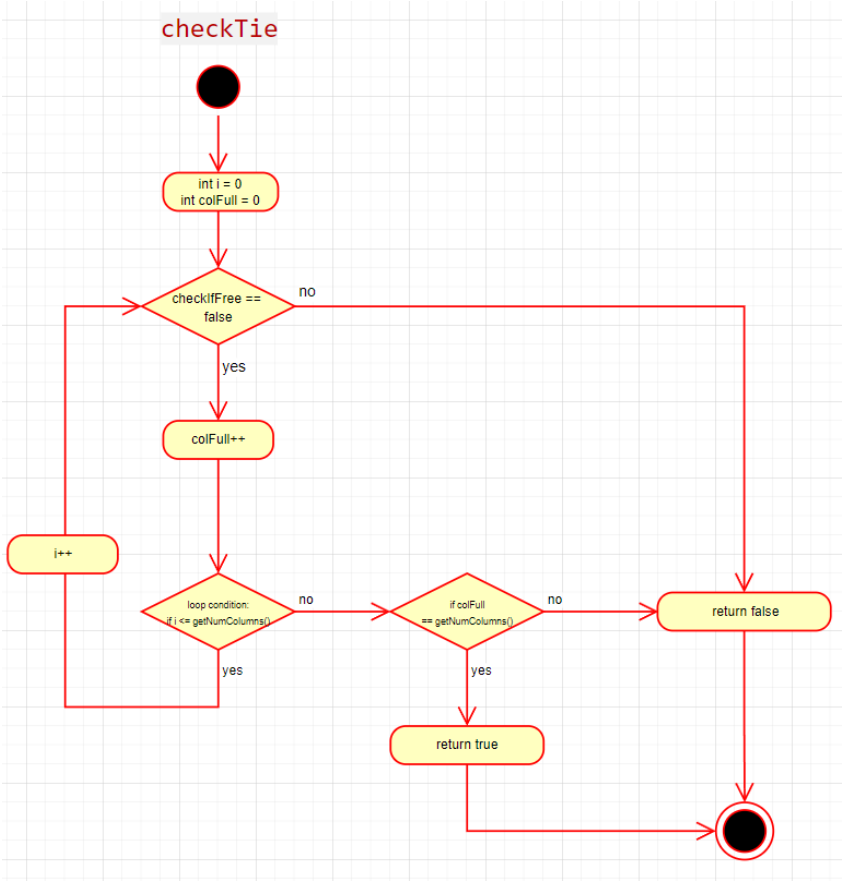
`isPlayerAtPos`



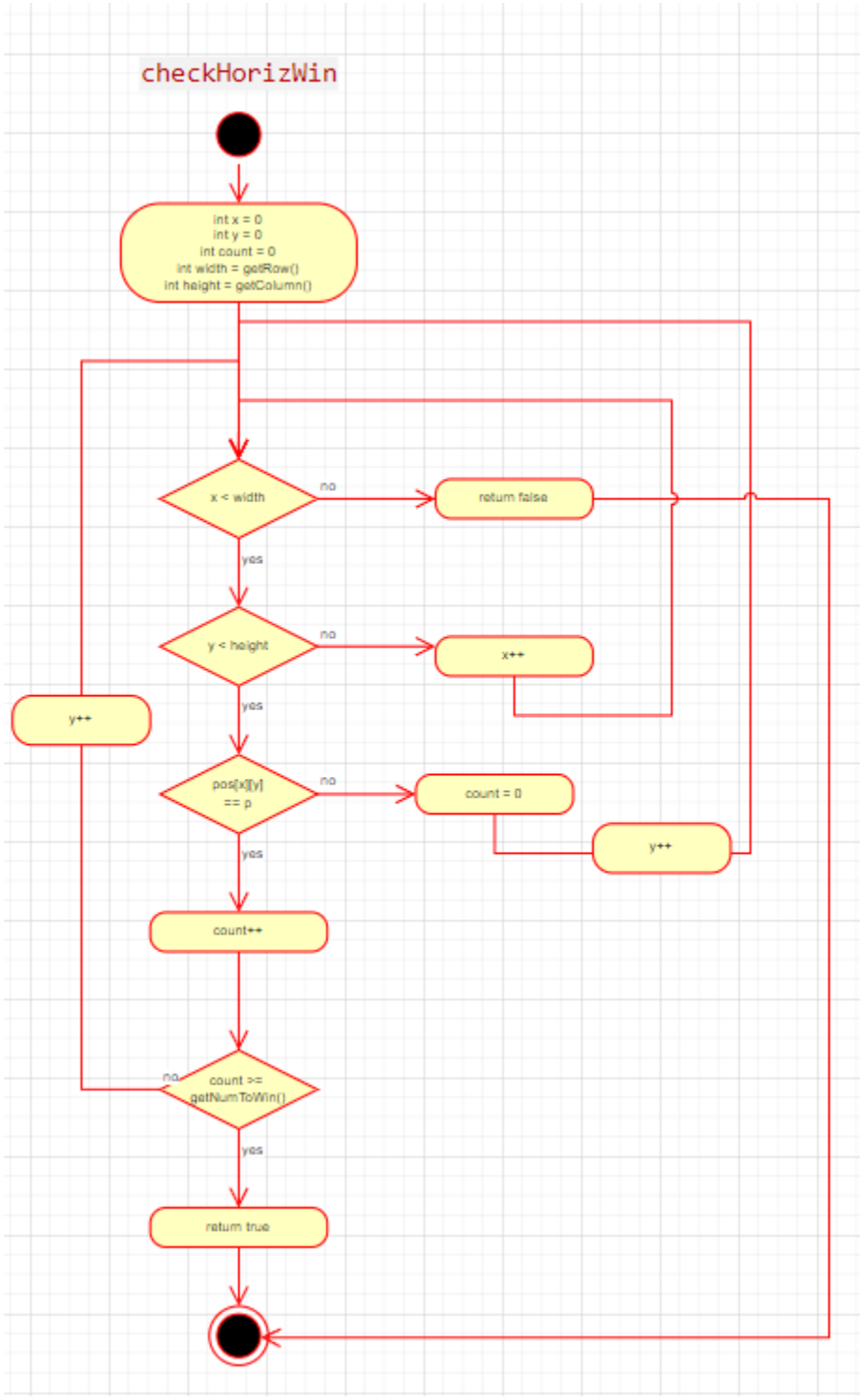
checkForWin



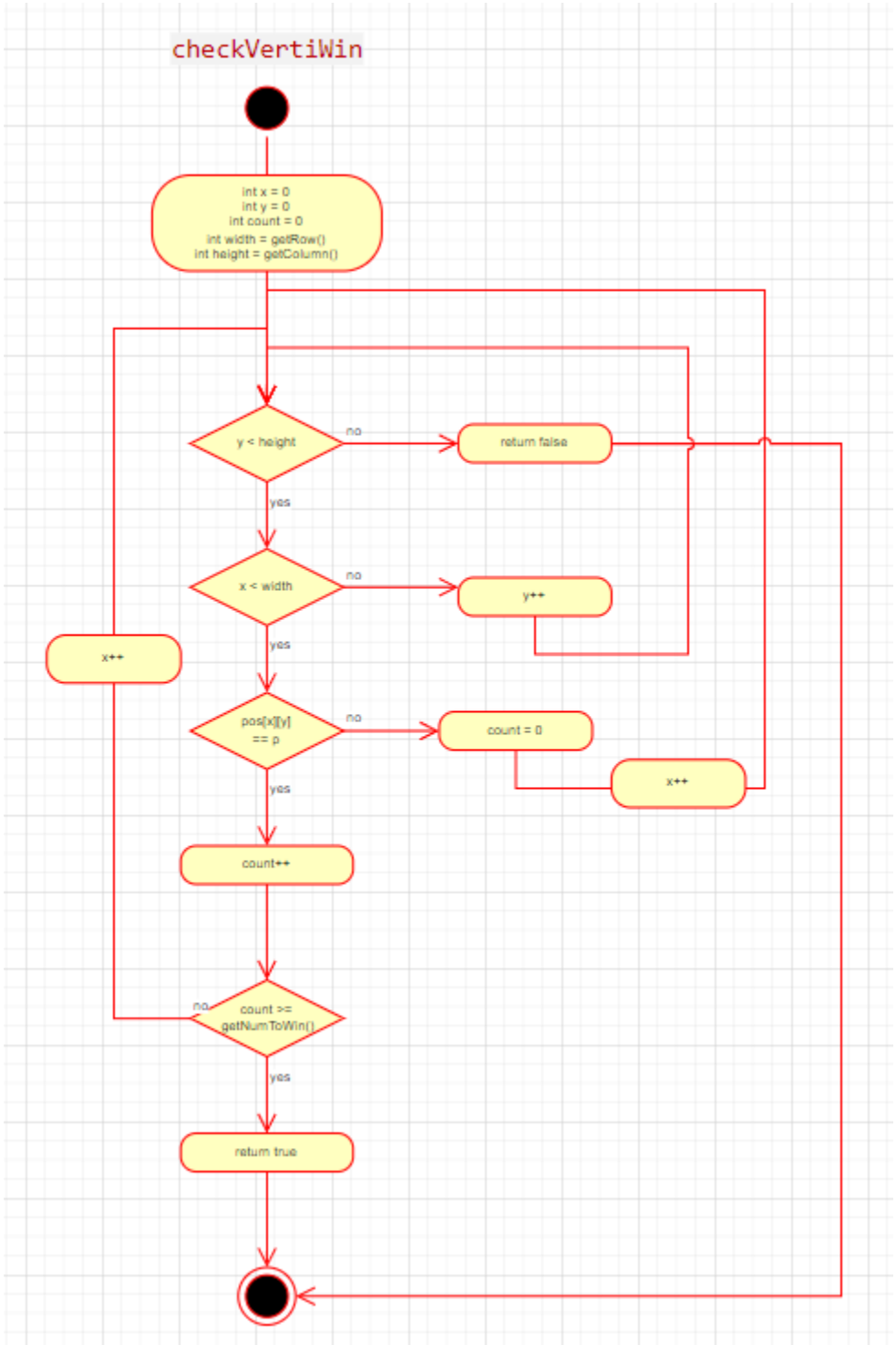
checkTie



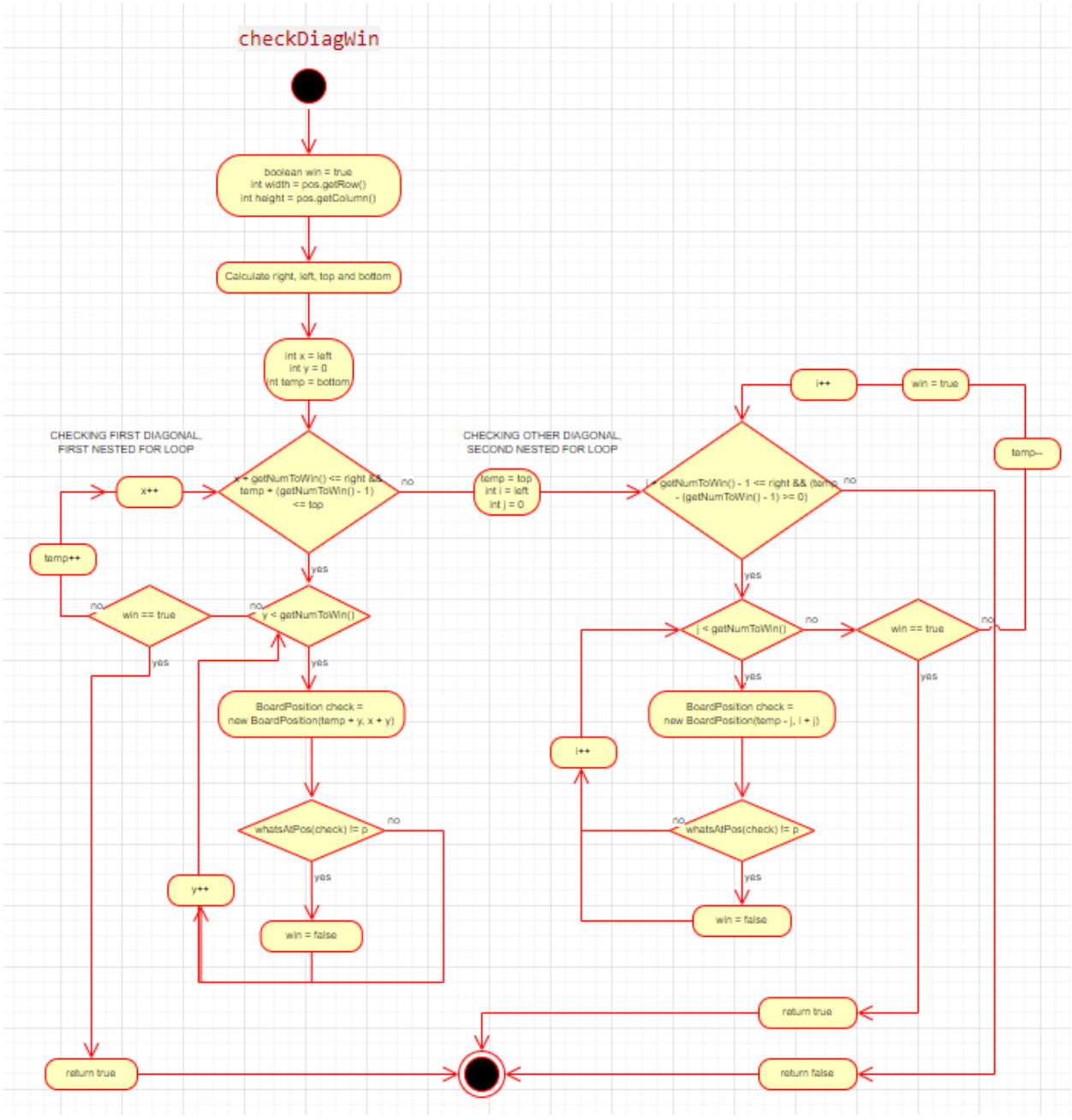
checkHorizWin



checkVertiWin



checkDiagWin



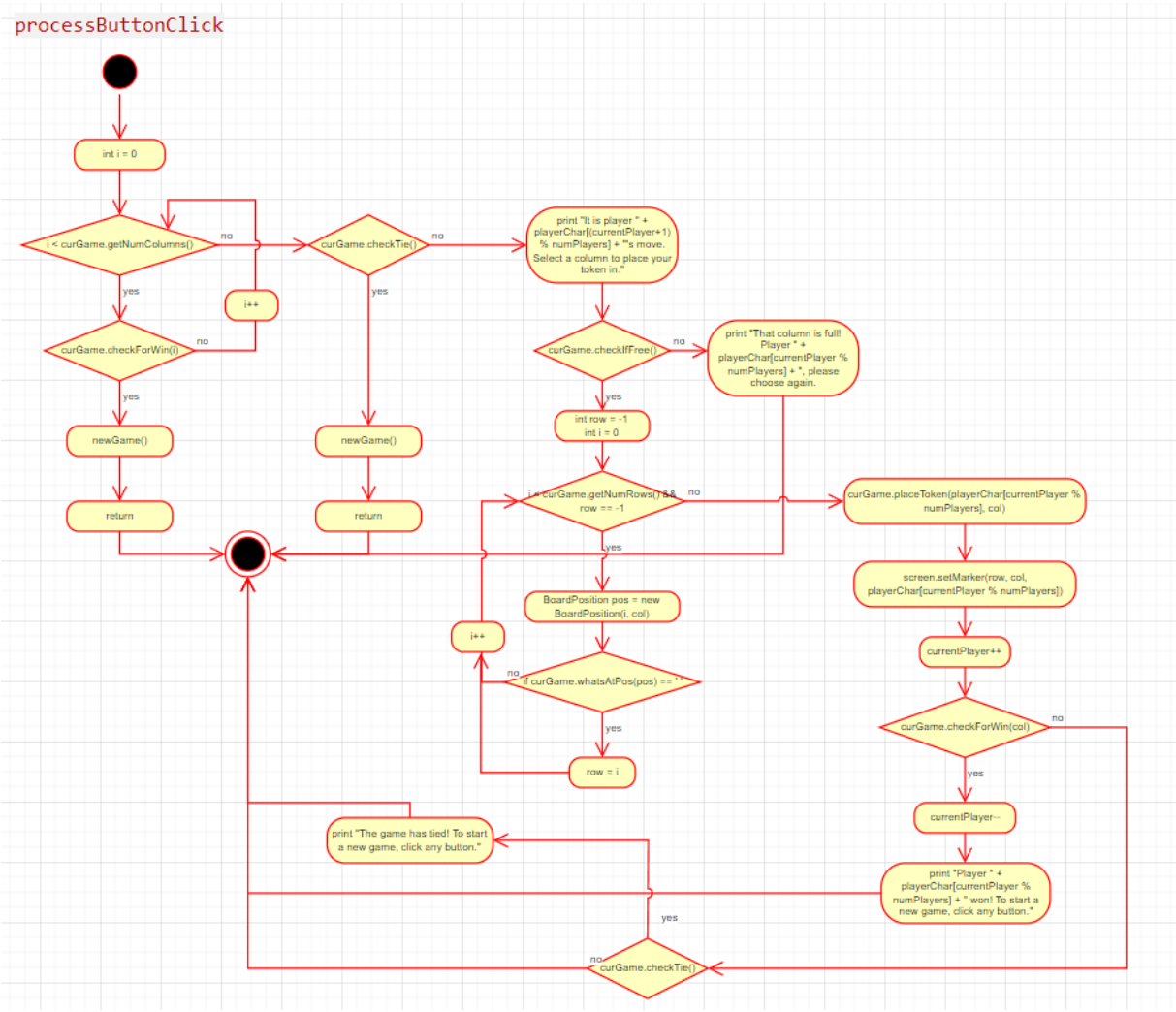
Class 4.0: `ConnectXController:`

Class diagram

ConnectXController
- curGame: IGameBoard[1] - screen: ConnectXView[1] - numPlayers: int[1] - playerChar: char[1] - currentPlayer: int[1] + MAX_PLAYERS: static final int
+ ConnectXController(IGameBoard, ConnectXView, int) + processButtonClick(int): void - newGame(): void

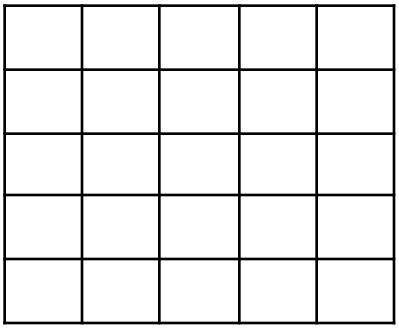
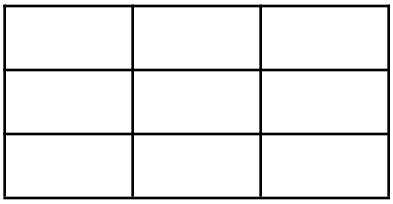
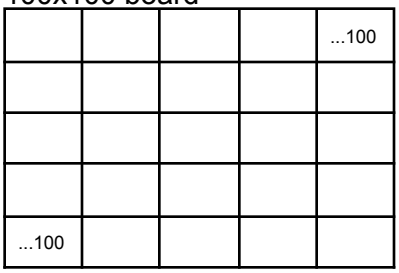
Activity diagrams

`processButtonClick`

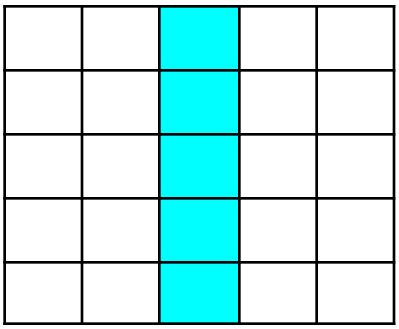
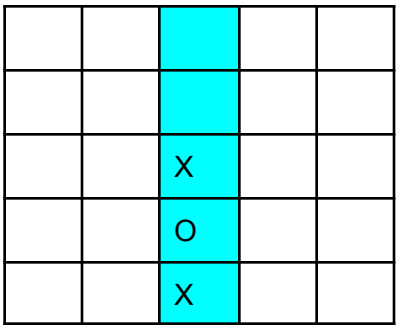
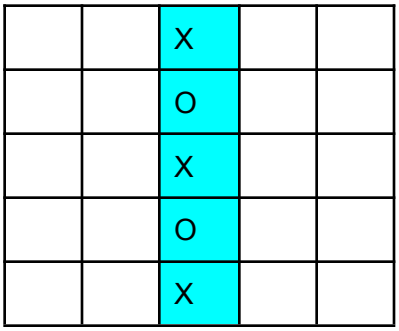


Test Cases

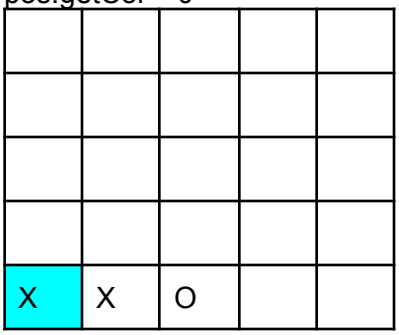
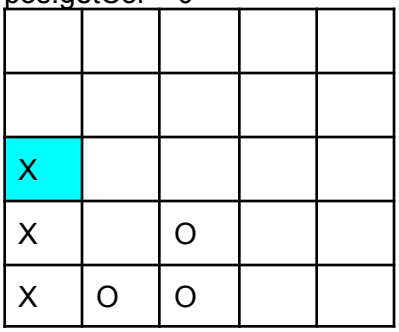
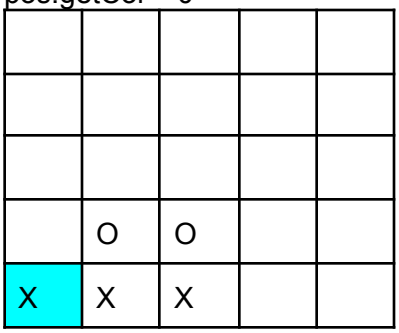
`GameBoard IGameBoard() - Constructor`

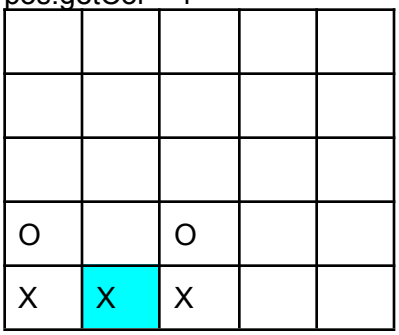
INPUT	OUTPUT	REASON
State: (number to win = 3) row = 5 col = 5	State: 	This test case is unique and distinct because it creates an empty GameBoard filled with empty spaces. **Function name:** `testConstructor_make_board`
State: (number to win = 3) row = 3 col = 3	State: 	This test case is unique and distinct because it creates an empty GameBoard of the smallest possible size filled with empty spaces. **Function name:** `testConstructor_min_size`
State: (number to win = 25) row = 100 col = 100	State: 100x100 board 	This test case is unique and distinct because it creates an empty GameBoard of the largest possible size filled with empty spaces. **Function name:** `testConstructor_max_size`

`boolean checkIfFree(int c)`

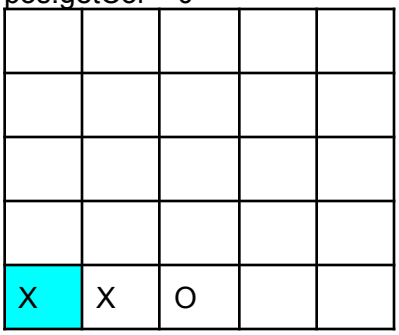
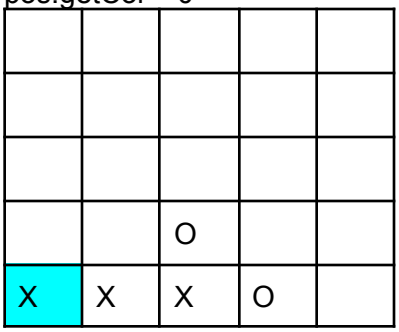
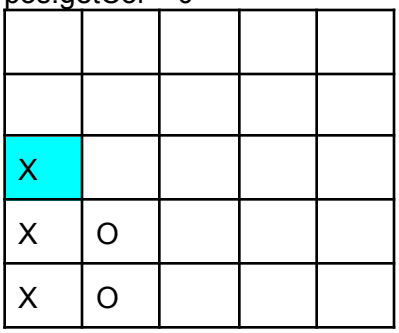
INPUT	OUTPUT	REASON
State: (number to win = 3) c = 2 	State: unchanged Returns true	This test case is unique and distinct because it checks if a column that has no tiles placed into it is free. **Function name:** `testCheckIfFree_empty_board`
State: (number to win = 3) c = 2 	State: unchanged Returns true	This test case is unique and distinct because it checks if a column that has some tiles in it (but is not full) is free. **Function name:** `testCheckIfFree_column_not_full`
State: (number to win = 3) c = 2 	State: unchanged Returns false	This test case is unique and distinct because it checks if a column that is full is free. **Function name:** `testCheckIfFree_column_full`

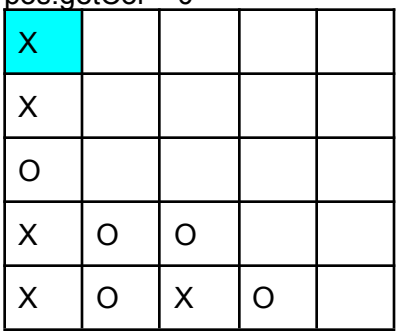
`boolean checkHorizWin(BoardPosition pos, char p)`

INPUT	OUTPUT	REASON
State: (num to win = 3) p = X pos.getRow = 0 pos.getCol = 0  	State: unchanged Returns false	This test case is unique and distinct because it's testing when there is no win. **Function name:** `testCheckHorizWin_no_win`
State: (num to win = 3) p = X pos.getRow = 2 pos.getCol = 0  	State: unchanged Returns false	This test case is unique and distinct because it's testing when there is no horizontal win, but there is a vertical win. **Function name:** `testCheckHorizWin_no_horiz_win`
State: (num to win = 3) p = X pos.getRow = 0 pos.getCol = 0  	State: unchanged Returns true	This test case is unique and distinct because it's testing when there is a horizontal win when the last marker was placed on the edge. **Function name:** `testCheckHorizWin_win_from_edge`

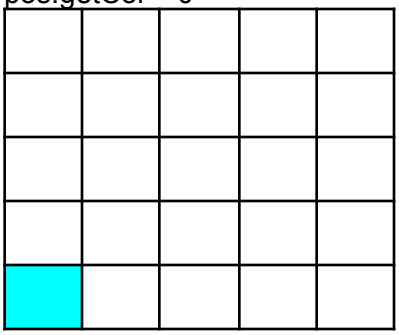
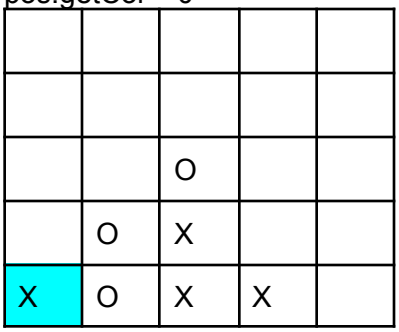
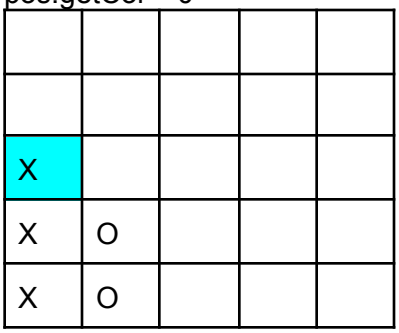
State: (num to win = 3) p = X pos.getRow = 0 pos.getCol = 1  	State: unchanged Returns true	This test case is unique and distinct because it's testing when there is a horizontal win when the last marker was placed in the middle. **Function name:** `testCheckHorizWin_win_from_middle`

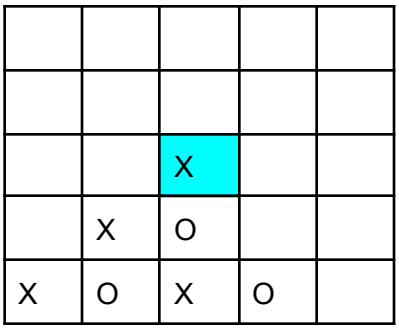
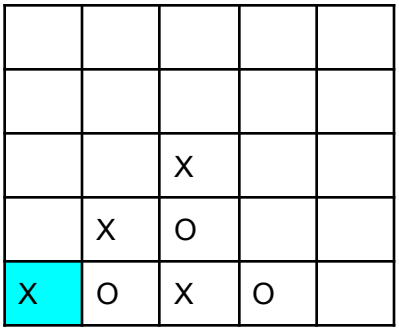
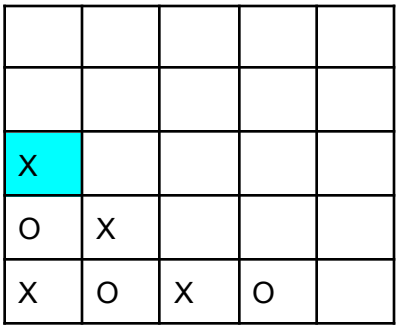
`boolean checkVertWin(BoardPosition pos, char p)`

INPUT	OUTPUT	REASON
State: (num to win = 3) p = X pos.getRow = 0 pos.getCol = 0  	State: unchanged Returns false	This test case is unique and distinct because it's testing when there is no win. **Function name:** `testCheckVertWin_no_win`
State: (num to win = 3) p = X pos.getRow = 0 pos.getCol = 0  	State: unchanged Returns false	This test case is unique and distinct because it's testing when there is no vertical win, but there is a horizontal win. **Function name:** `testCheckVertWin_no_vert_win`
State: (num to win = 3) p = X pos.getRow = 2 pos.getCol = 0  	State: unchanged Returns true	This test case is unique and distinct because it's testing when there is a vertical win when the last marker was placed on the top. **Function name:** `testCheckVertWin_win_from_top`

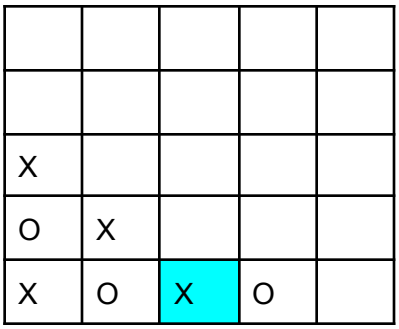
State: (num to win = 3) p = X pos.getRow = 4 pos.getCol = 0  	State: unchanged Returns false	This test case is unique and distinct because it's testing if there is a vertical win in a column that is full (after the user tries to place in the full column and if checkIfFree is returned false, make sure that it doesn't still check that or count it as a win). **Function name:** `testCheckVertWin_full_column`

`boolean checkDiagWin(BoardPosition pos, char p)`

INPUT	OUTPUT	REASON
State: (num to win = 3) p = X pos.getRow = 0 pos.getCol = 0 	State: unchanged Returns false	This test case is unique and distinct because it's checking to see if there is a diagonal win on a completely empty board. **Function name:** `testCheckDiagWin_empty_board`
State: (num to win = 3) p = X pos.getRow = 0 pos.getCol = 0 	State: unchanged Returns false	This test case is unique and distinct because it's testing when there is no win. **Function name:** `testCheckDiagWin_no_win`
State: (num to win = 3) p = X pos.getRow = 2 pos.getCol = 0 	State: unchanged Returns false	This test case is unique and distinct because it's testing when there is a win (vertical in this case) but no diagonal win. **Function name:** `testCheckDiagWin_no_diag_win`

State: (num to win = 3) p = X pos.getRow = 2 pos.getCol = 2  	State: unchanged Returns true	This test case is unique and distinct because it's testing when there is a diagonal win from left to right, where the token was placed in the top. **Function name:** `testCheckDiagWin_left_to_right_from_top`
State: (num to win = 3) p = X pos.getRow = 0 pos.getCol = 0  	State: unchanged Returns true	This test case is unique and distinct because it's testing when there is a diagonal win from left to right, where the token was placed in the bottom. **Function name:** `testCheckDiagWin_left_to_right_from_bottom`
State: (num to win = 3) p = X pos.getRow = 2 pos.getCol = 0  	State: unchanged Returns true	This test case is unique and distinct because it's testing when there is a diagonal win from right to left, where the token was placed in the top. **Function name:** `testCheckDiagWin_right_to_left_from_top`

<table>
<tr>
<td>State: (num to win = 3)
p = X
pos.getRow = 0
pos.getCol = 2

 
</td>
<td>State: unchanged

Returns true</td>
<td>This test case is unique and distinct because it’s testing when there is a diagonal win from right to left, where the token was placed in the bottom.

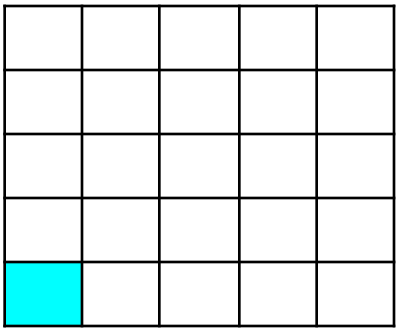
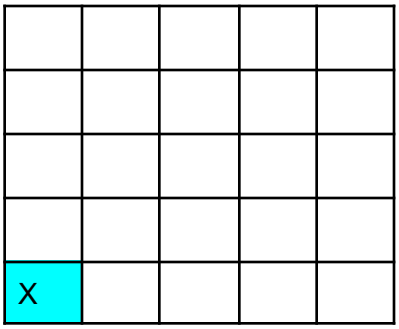
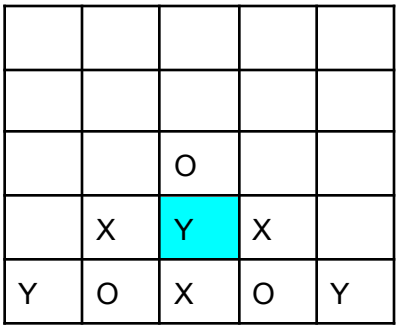
Function name:
<code>testCheckDiagWin_right_to_left_from_bottom</code></td>
</tr>
</table>

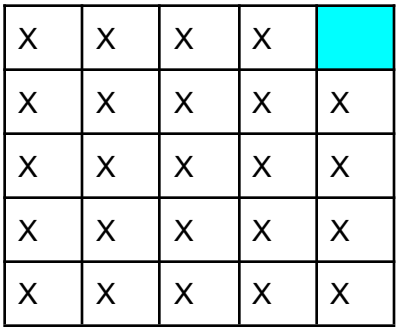
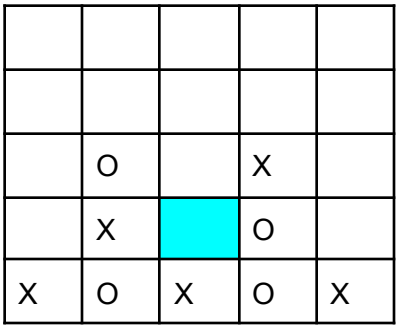
`boolean checkTie()`

INPUT	OUTPUT	REASON
State: (num to win = 3) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |	State: unchanged Returns false	This test case is unique and distinct because it's testing when there is a completely empty board, so no tie. **Function name:** `testCheckTie_empty_board`
State: (num to win = 3) | | | | | | | | | | | | | | | | | | | | | | | | | X | | | |	State: unchanged Returns false	This test case is unique and distinct because it's testing when there is only one marker on the board. **Function name:** `testCheckTie_not_full_board`
State: (num to win = 3) | X | X | X | X | X | | X | X | X | X | X | | X | X | X | X | X | | X | X | X | X | X | | X | X | X | X | X |	State: unchanged Returns true	This test case is unique and distinct because it's testing when there is a completely full board, with no wins. **Function name:** `testCheckTie_full_board`

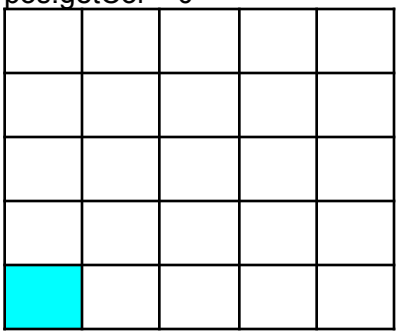
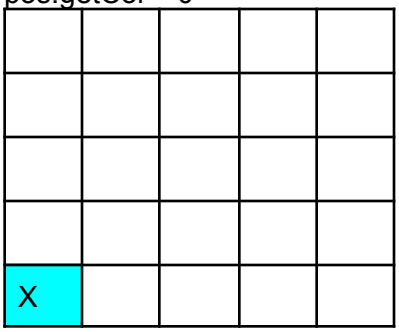
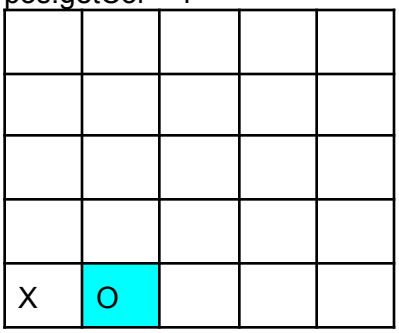
State: (num to win = 3) X X X X X X X X X X X X X X X	State: unchanged Returns false	This test case is unique and distinct because it's testing when there are several full columns, but still room on the board. **Function name:** `testCheckTie_full_columns`

`char whatsAtPos(BoardPosition pos)`

INPUT	OUTPUT	REASON
State: (num to win = 3) pos.getRow = 0 pos.getCol = 0 	State: unchanged Returns ‘ ‘	This test case is unique and distinct because it’s testing what is at a position in a completely empty board. **Function name:** `testWhatsAtPos_empty_board`
State: (num to win = 3) pos.getRow = 0 pos.getCol = 0 	State: unchanged Returns ‘X’	This test case is unique and distinct because it’s testing what’s at a single occupied position in an otherwise empty board. **Function name:** `testWhatsAtPos_character`
State: (num to win = 3) pos.getRow = 1 pos.getCol = 2 	State: unchanged Returns ‘Y’	This test case is unique and distinct because it’s testing when the player character is surrounded by other characters. **Function name:** `testWhatsAtPos_character_surrounded`

State: (num to win = 3) pos.getRow = 4 pos.getCol = 4  	State: unchanged Returns ‘ ’	This test case is unique and distinct because it’s testing when there is only one empty space on an otherwise full board. **Function name:** `testWhatsAtPos_almost_full_board`
State: (num to win = 3) pos.getRow = 1 pos.getCol = 2  	State: unchanged Returns ‘ ‘	This test case is unique and distinct because it’s testing when there is a space amongst several characters. **Function name:** `testWhatsAtPos_space_in_between`

`boolean isPlayerAtPos(BoardPosition pos, char player)`

INPUT	OUTPUT	REASON
State: (num to win = 3) player = X pos.getRow = 0 pos.getCol = 0 	State: unchanged Returns false	This test case is unique and distinct because it's testing whether the player is at the position in the board while the board is empty. **Function name:** `testIsPlayerAtPos_empty_board`
State: (num to win = 3) player = X pos.getRow = 0 pos.getCol = 0 	State: unchanged Returns true	This test case is unique and distinct because it's checking if the player is in the position on an otherwise empty board. **Function name:** `testIsPlayerAtPos_single_character`
State: (num to win = 3) player = X pos.getRow = 0 pos.getCol = 1 	State: unchanged Returns false	This test case is unique and distinct because it's checking if the player X is in a position that is occupied, but not by X. **Function name:** `testIsPlayerAtPos_wrong_character`

<table>
<tr>
<td>State: (num to win = 3)
player = G
pos.getRow = 0
pos.getCol = 0

<table><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td></td></tr></table></td>
<td>State: unchanged

Returns false</td>
<td>This test case is unique and distinct because it’s testing if a player that is not one of the players playing is at one of the positions.

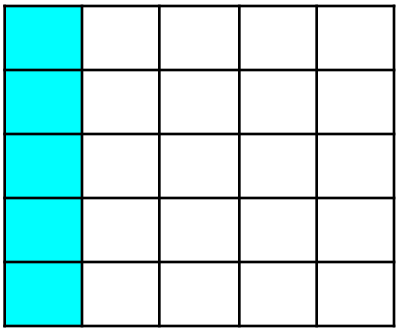
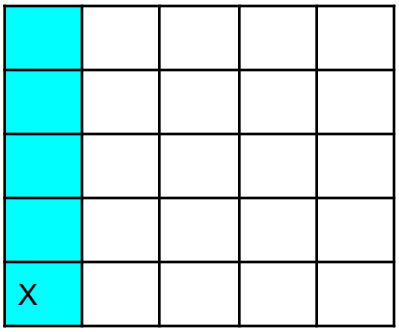
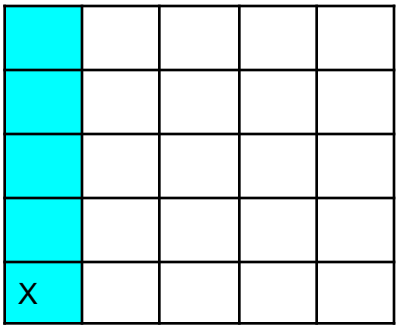
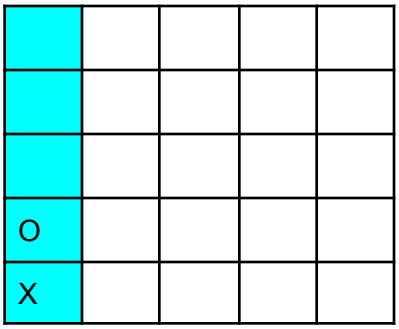
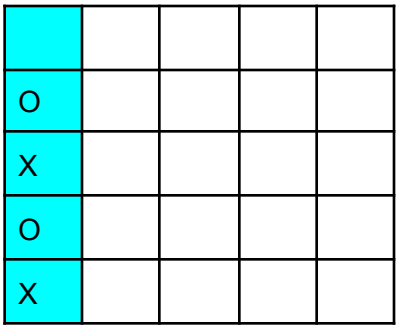
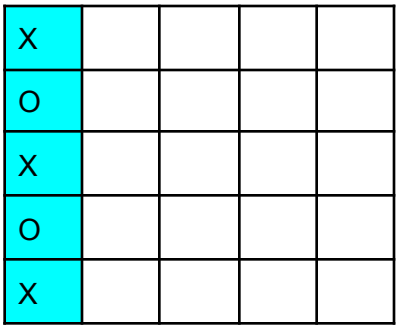
Function name:
<code>testIsPlayerAtPos_invalid_player</code></td>
</tr>
<tr>
<td>State: (num to win = 3)
player = X
pos.getRow = 4
pos.getCol = 4

<table><tr><td>X</td><td>X</td><td>X</td><td>X</td><td></td></tr><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr></table></td>
<td>State: unchanged

Returns false</td>
<td>This test case is unique and distinct because it’s testing if the player, X, is at the only empty position in an otherwise full board.

Function name:
<code>testIsPlayerAtPos_almst_full_board</code></td>
</tr>
</table>

`void placeToken(char p, int c)`

INPUT	OUTPUT	REASON
State: (num to win = 3) p = X c = 0 	State: 	This test case is unique and distinct because it is testing placing a token on an empty board. **Function name:** `testPlaceToken_empty_board`
State: (num to win = 3) p = O c = 0 	State: 	This test case is unique and distinct because it is testing placing a token on top of another. **Function name:** `testPlaceToken_on_top_of_character`
State: (num to win = 3) p = X c = 0 	State: 	This test case is unique and distinct because it is placing a token into a column that was almost full, and still places the token properly. **Function name:** `testPlaceToken_column_almost_full`

<table>
<tr>
<td>State: (num to win = 3)
p = X
c = 4

<table><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr></table></td>
<td>State:

<table><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr><tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr></table></td>
<td>This test case is unique and distinct because it’s testing placing tokens to fill the entire board up completely.

Function name:
<code>testPlaceToken_till_full</code></td>
</tr>
<tr>
<td>State: (num to win = 3)
p = X
c = 4

<table><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td></td></tr></table></td>
<td>State:

<table><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr></table></td>
<td>This test case is unique and distinct because it’s testing placing a token in the only open spot in an almost-full row, and makes sure that the token still gets placed in the correct location (bottom-most position)

Function name:
<code>testPlaceToken_row_almost_full</code></td>
</tr>
</table>