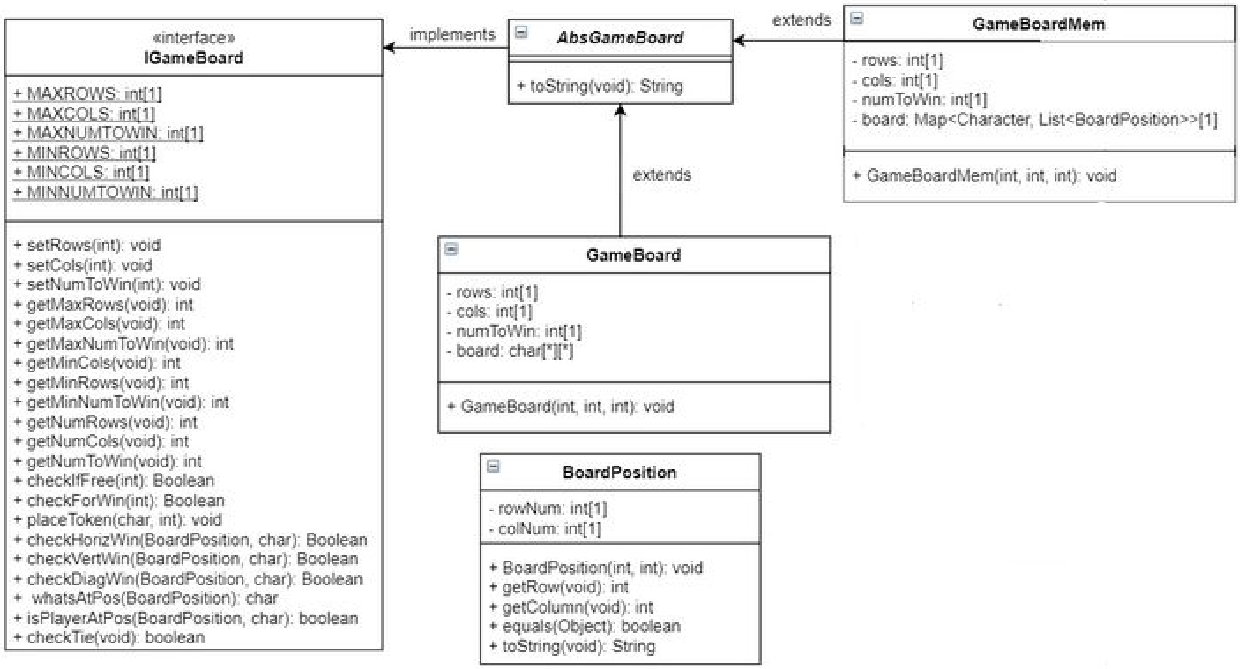
Requirements Analysis:

- Functional Requirements:
    - As a player, I must be able to see the board so that I know the current board state.
    - As a player, I must be able to choose the total number of players so that I can play with other people.
    - As a player, I must be able to choose the number of rows in the board so I can make the game board the size of my choosing.
    - As a player, I must be able to choose an invalid number of rows so I know when the game board is getting too big or too small.
    - As a player, I must be able to choose the number of columns in the board so I can make the game board the size of my choosing.
    - As a player, I must be able to choose an invalid number of columns so I know when the game board is getting too big or too small.
    - As a player, I must be able to choose how many consecutive tokens are required to win so I can set how easily I can win the game.
    - As a player, I must be able to choose an invalid number of consecutive tokens required to win so I know when the game is too difficult or too easy to win.
    - As a player, I must be able to choose a column in which to place my token.
    - As a player, I must be able to make my column choice after my opponents if they did not win so that I can continue playing the game.
    - As a player, I must be able to win horizontally in order to win.
    - As a player, I must be able to win vertically in order to win.
    - As a player, I must be able to win diagonally in order to win.
    - As a player, I must be able to see who won the game so that I know if I won or lost.
    - As a player, I must be able to see if the game is a tie, so that I know the game ended in a tie.
    - As a player, I must have the option to play a new game if I want to play more than one game.
    - As a player, I must be able to choose new game settings whenever I start a new game in case I want to change any settings from the previous game.

- Nonfunctional Requirements:
    - The game must be programmed using Java.
    - The bottom left corner of the board is (0, 0).
    - The maximum size of the board is 100 x 100.
    - The minimum size of the board is 3 x 3.
    - The size of the game board is determined by the player.
    - The maximum tokens in a row to win is 25.
    - The minimum tokens in a row to win is 3.
    - The number of tokens in a row to win is determined by the player.
    - The maximum number of players is 10.

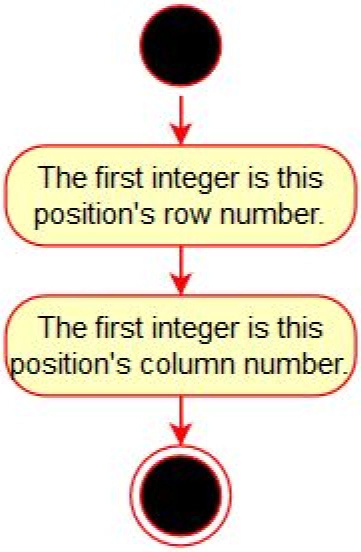
- ○ The minimum number of players is 2.
- ○ No two players may choose the same token.
- ○ Player 1 always goes first.

Class Diagrams:



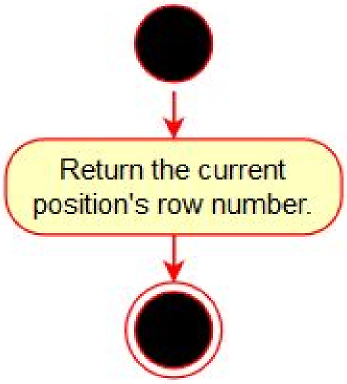
BoardPosition Class:

public BoardPosition(int row, int column):

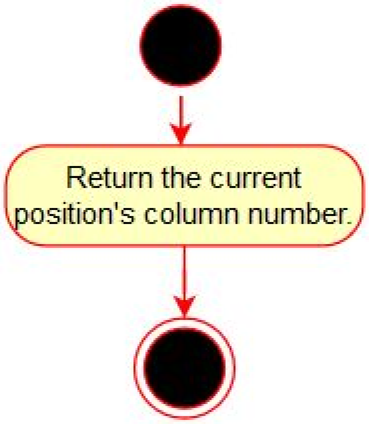


The first integer is this position's row number.
The first integer is this position's column number.

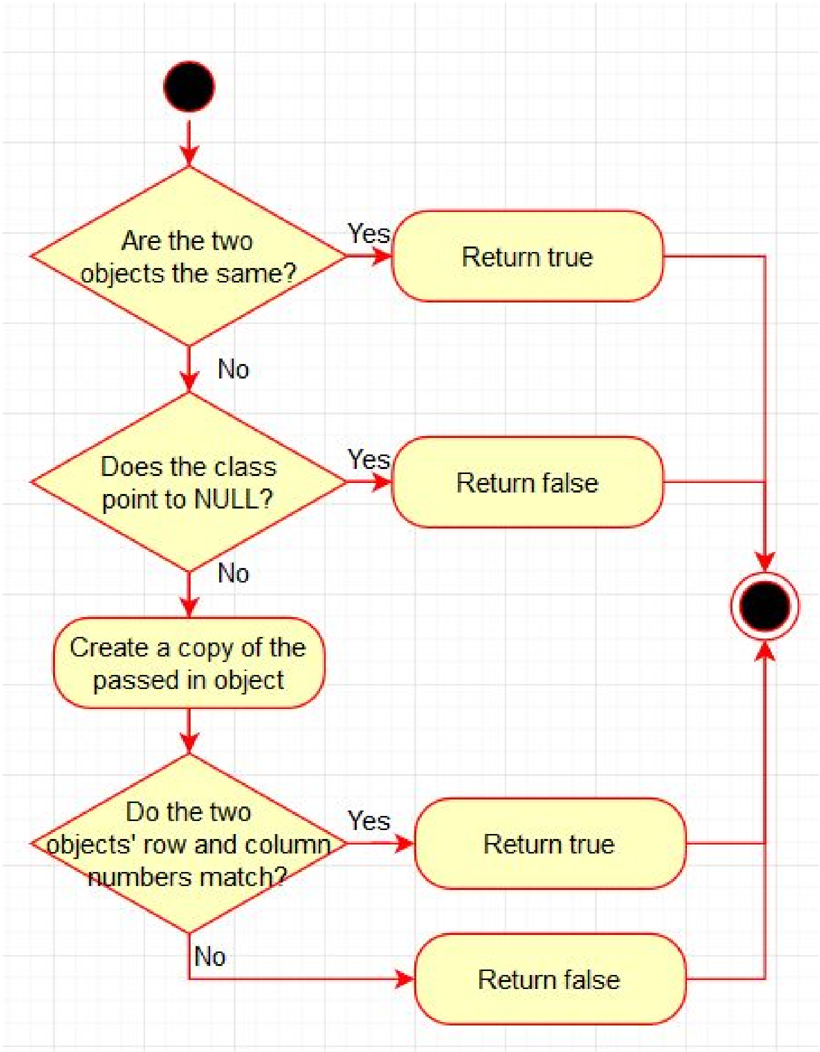
public int getRow():



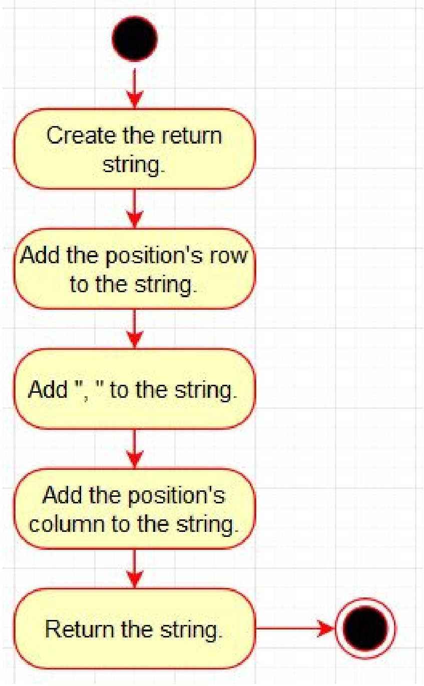
public int getColumn():



public bool equals():

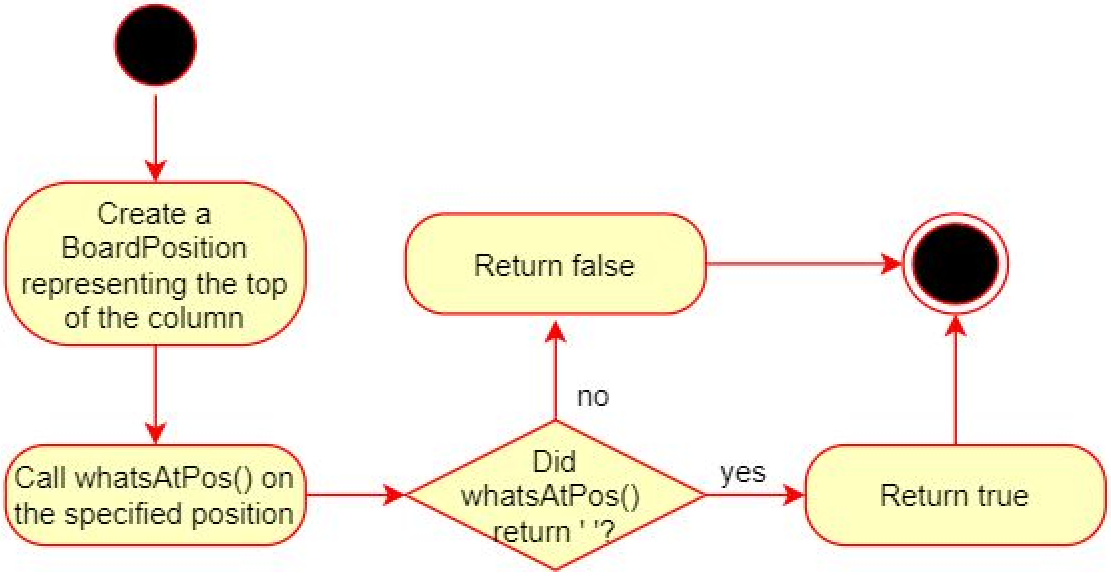

Are the two objects the same?
Yes
Return true
No
Does the class point to NULL?
Yes
Return false
No
Create a copy of the passed in object
Do the two objects' row and column numbers match?
Yes
Return true
No
Return false

public String toString():

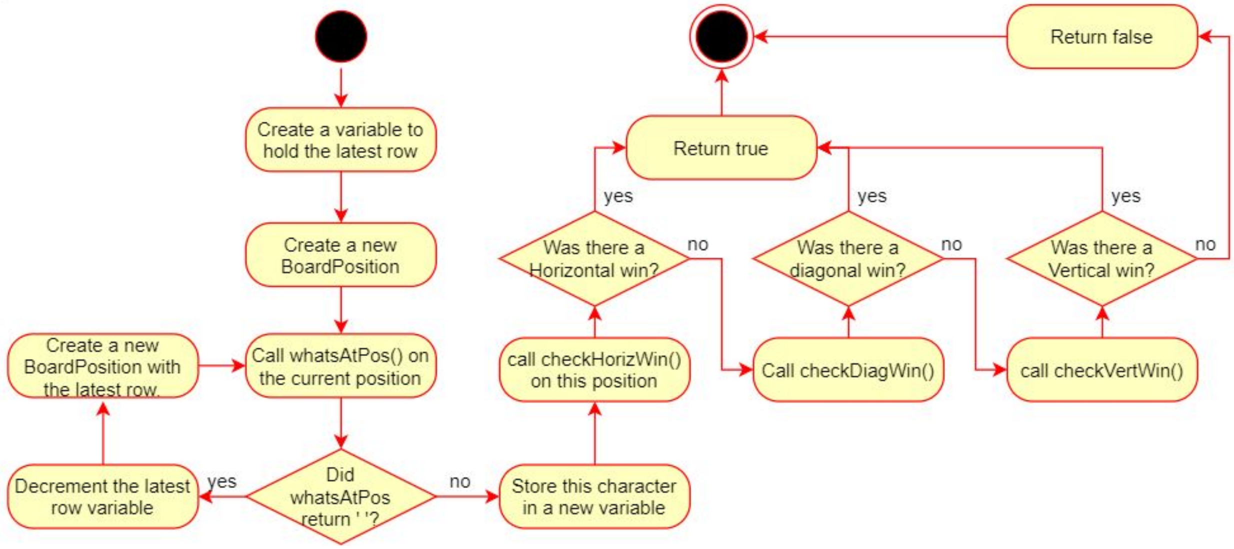

Create the return string.
Add the position's row to the string.
Add ", " to the string.
Add the position's column to the string.
Return the string.

IGameBoard Interface:

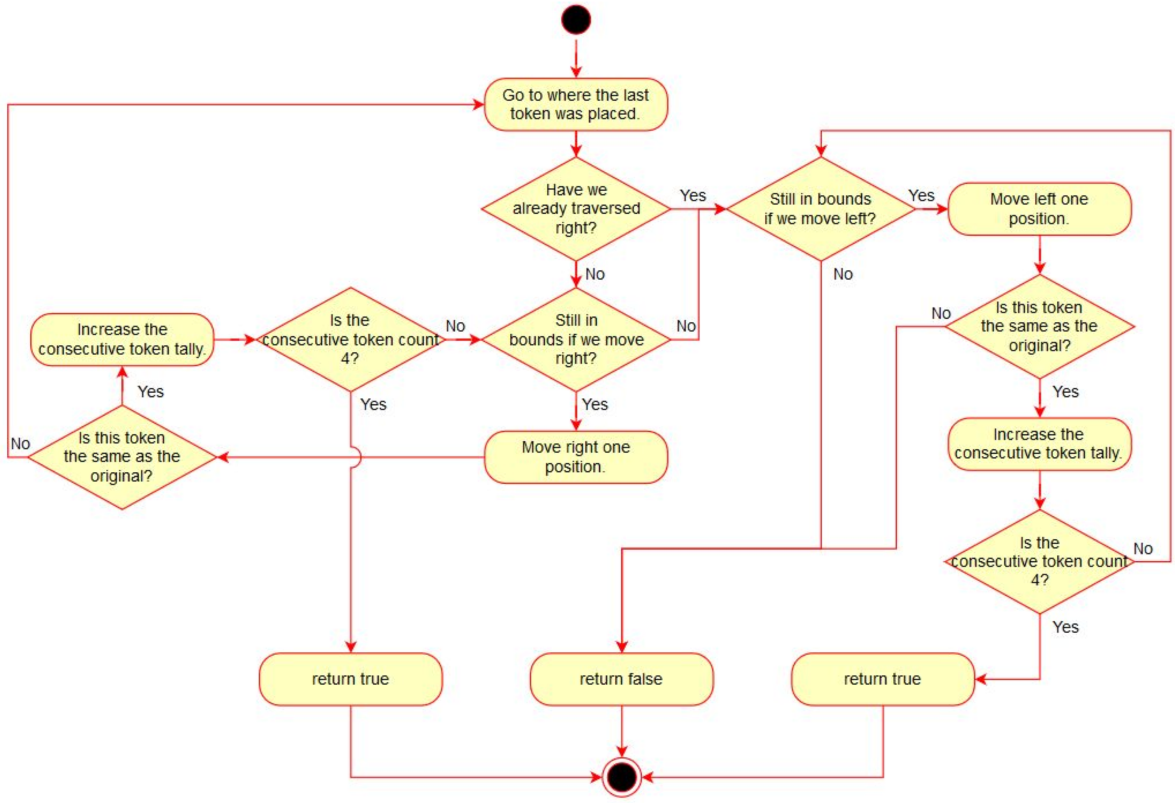
public boolean checkIfFree(int c):


Create a BoardPosition representing the top of the column
Call whatsAtPos() on the specified position
Did whatsAtPos() return ' '?
no
yes
Return false
Return true

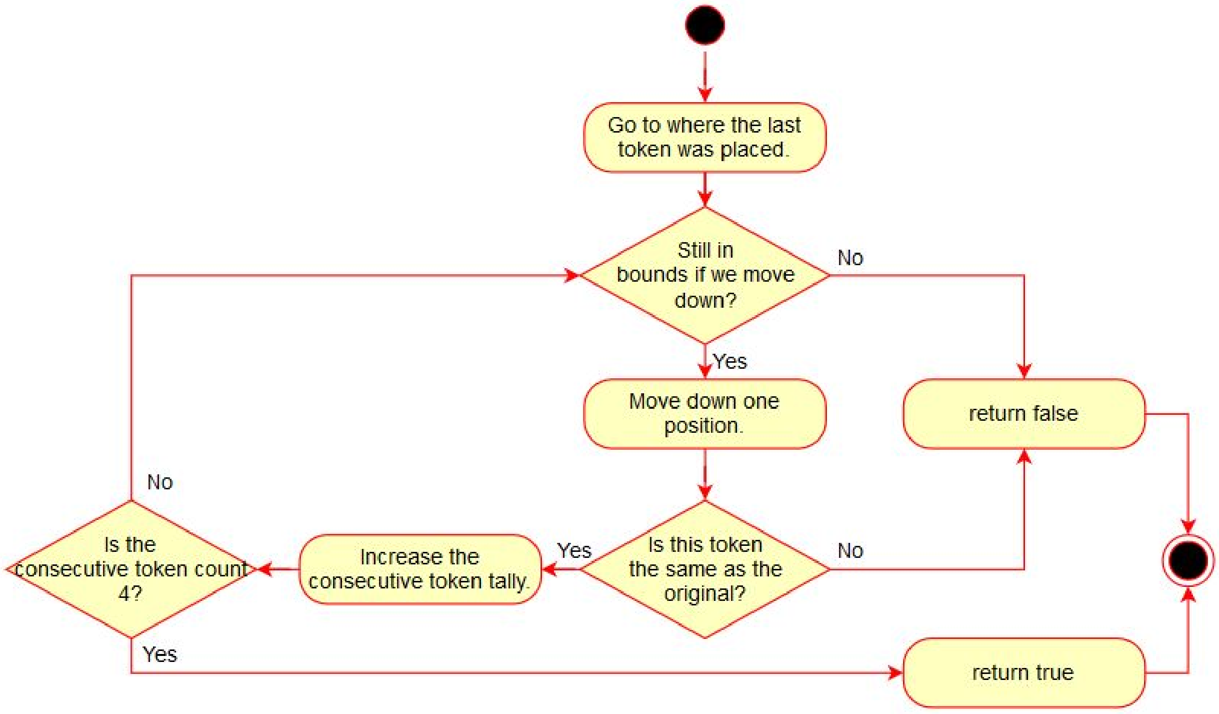
public boolean checkForWin(int c):


Create a variable to hold the latest row
Create a new BoardPosition
Call whatsAtPos() on the current position
Did whatsAtPos return ' '?
yes
Decrement the latest row variable
Create a new BoardPosition with the latest row.
no
Store this character in a new variable
call checkHorizWin() on this position
Was there a Horizontal win?
yes
Return true
no
Call checkDiagWin()
Was there a diagonal win?
yes
no
call checkVertWin()
Was there a Vertical win?
yes
no
Return false

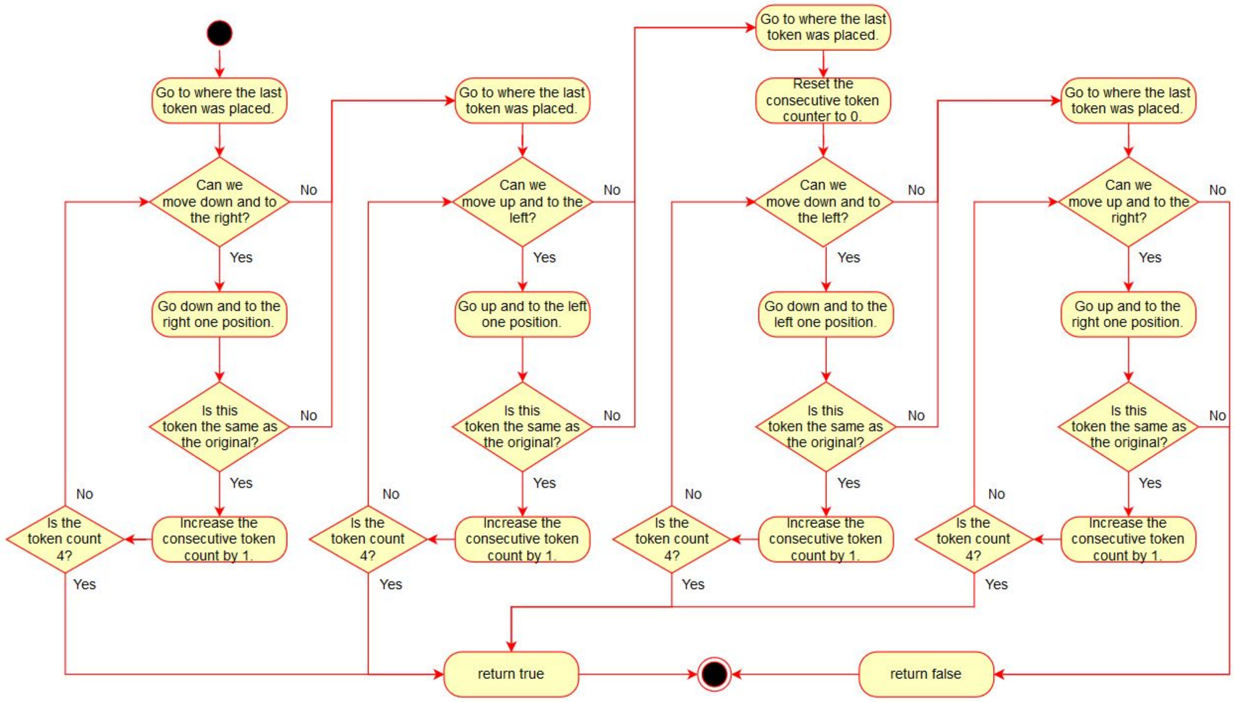
public boolean checkHorizWin(BoardPosition pos, char p):


Go to where the last token was placed.
Have we already traversed right?
Yes
No
Still in bounds if we move right?
No
Yes
Move right one position.
Is this token the same as the original?
Yes
No
Increase the consecutive token tally.
Is the consecutive token count 4?
No
Yes
return true
Still in bounds if we move left?
Yes
No
Move left one position.
Is this token the same as the original?
No
Yes
Increase the consecutive token tally.
Is the consecutive token count 4?
No
Yes
return false
return true

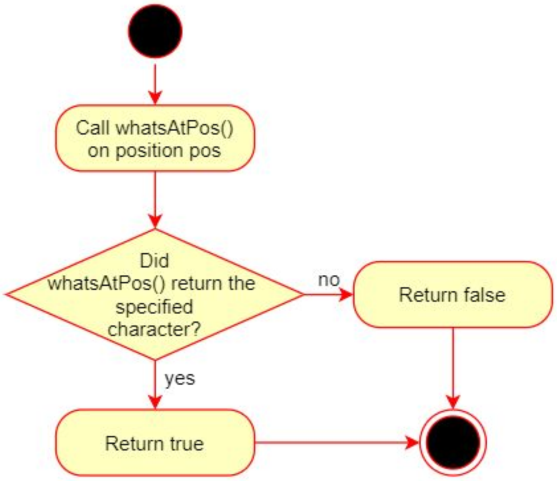
public boolean checkVertWin(BoardPosition pos, char p):


Go to where the last token was placed.
Still in bounds if we move down?
No
Yes
Move down one position.
return false
No
Is the consecutive token count 4?
Increase the consecutive token tally.
Yes
Is this token the same as the original?
No
Yes
return true

public boolean checkDiagWin(BoardPosition pos, char p):

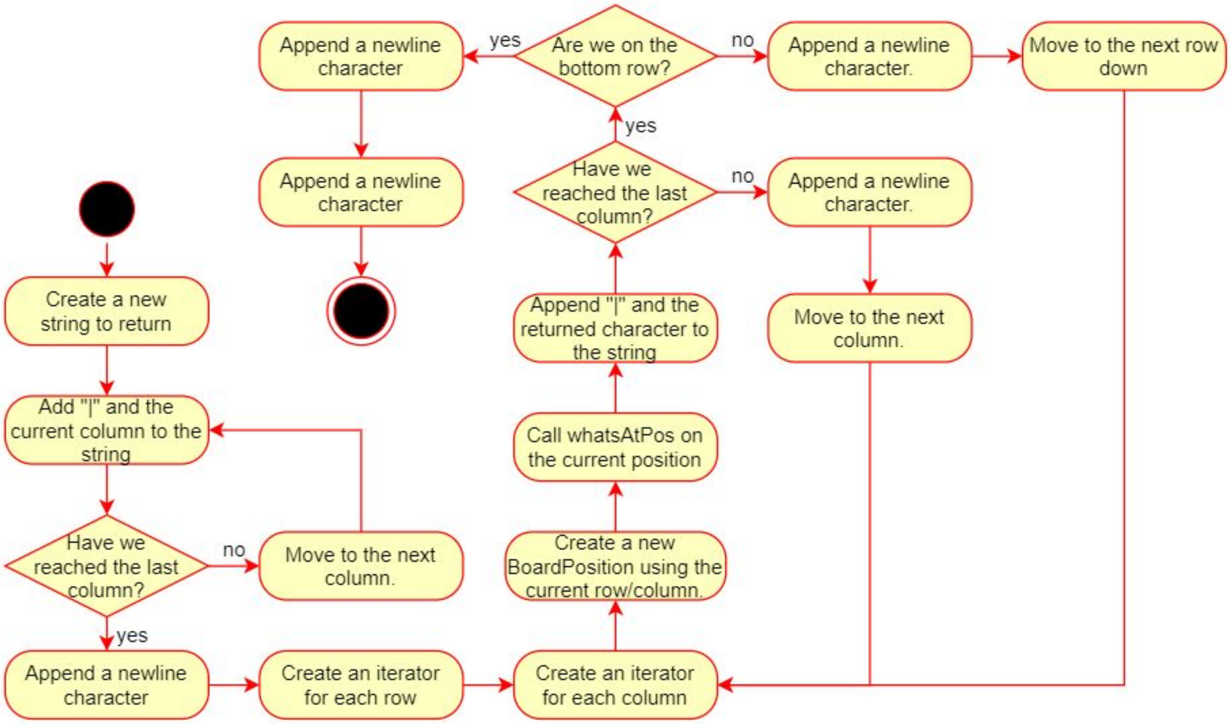

Go to where the last token was placed.
Can we move down and to the right?
No
Yes
Go down and to the right one position.
Is this token the same as the original?
No
Yes
Increase the consecutive token count by 1.
Is the token count 4?
No
Yes
Go to where the last token was placed.
Can we move up and to the left?
No
Yes
Go up and to the left one position.
Is this token the same as the original?
No
Yes
Increase the consecutive token count by 1.
Is the token count 4?
No
Yes
Go to where the last token was placed.
Reset the consecutive token counter to 0.
Can we move down and to the left?
No
Yes
Go down and to the left one position.
Is this token the same as the original?
No
Yes
Increase the consecutive token count by 1.
Is the token count 4?
No
Yes
Go to where the last token was placed.
Can we move up and to the right?
No
Yes
Go up and to the right one position.
Is this token the same as the original?
No
Yes
Increase the consecutive token count by 1.
Is the token count 4?
No
Yes
return true
return false

public boolean isPlayerAtPos(BoardPosition pos):


Call whatsAtPos() on position pos
Did whatsAtPos() return the specified character?
no
Return false
yes
Return true

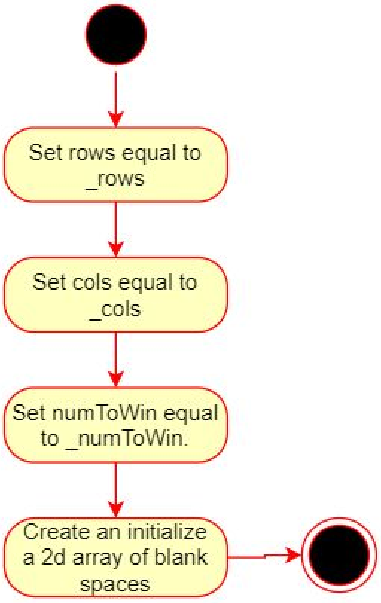
AbsGameBoard:

String toString():

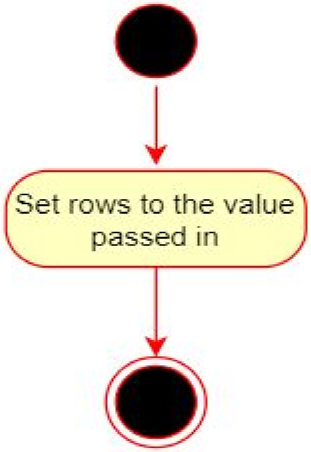
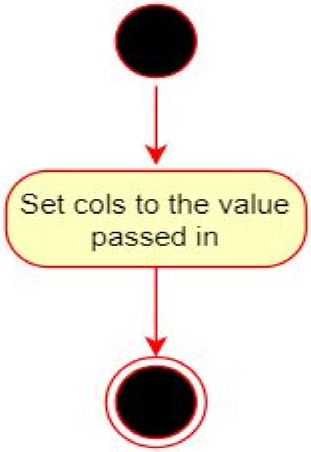
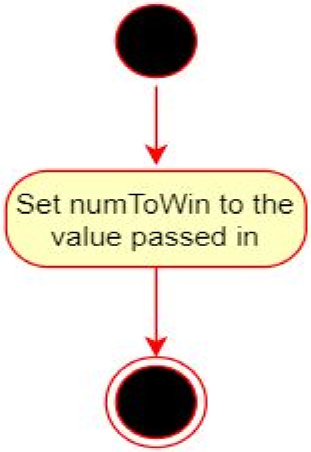

Append a newline character
yes
Are we on the bottom row?
no
Append a newline character.
Move to the next row down
yes
Append a newline character
Have we reached the last column?
no
Append a newline character.
Create a new string to return
Append "|" and the returned character to the string
Move to the next column.
Add "|" and the current column to the string
Call whatsAtPos on the current position
Have we reached the last column?
no
Move to the next column.
Create a new BoardPosition using the current row/column.
yes
Append a newline character
Create an iterator for each row
Create an iterator for each column

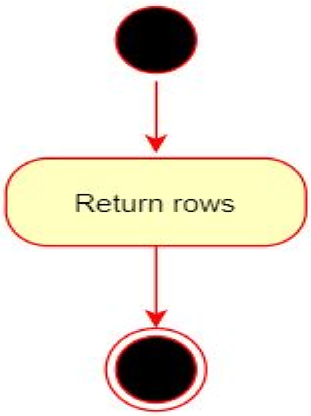
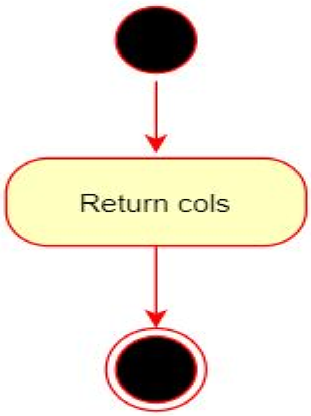
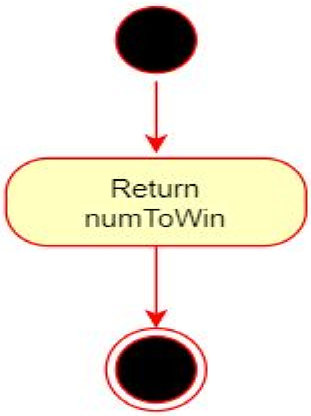
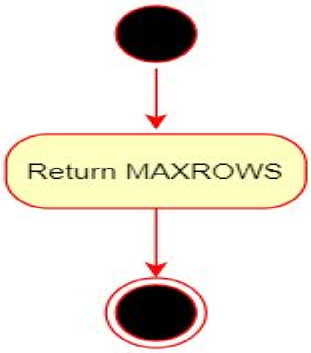
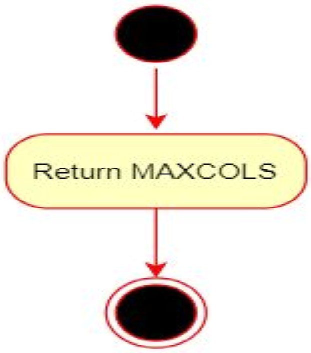
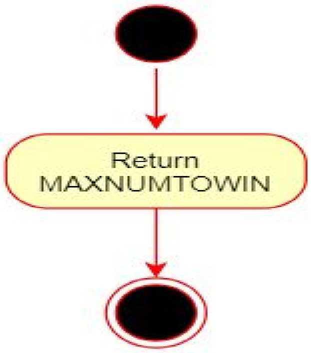
GameBoard:

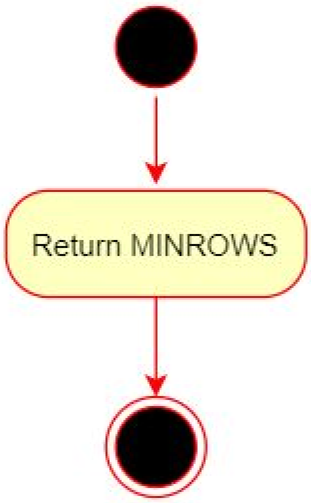
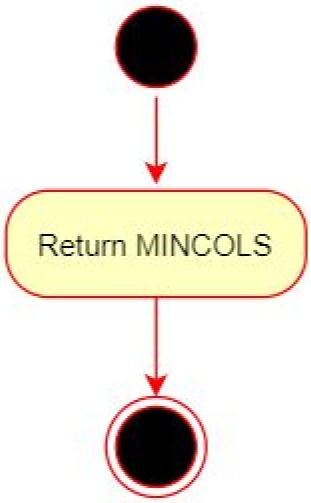
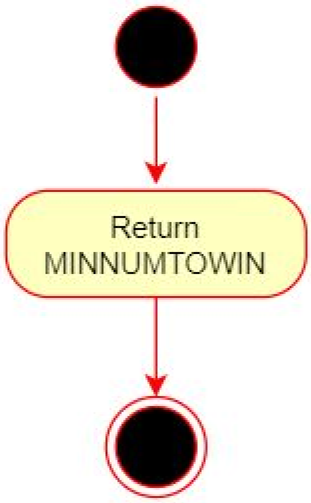
public GameBoard():



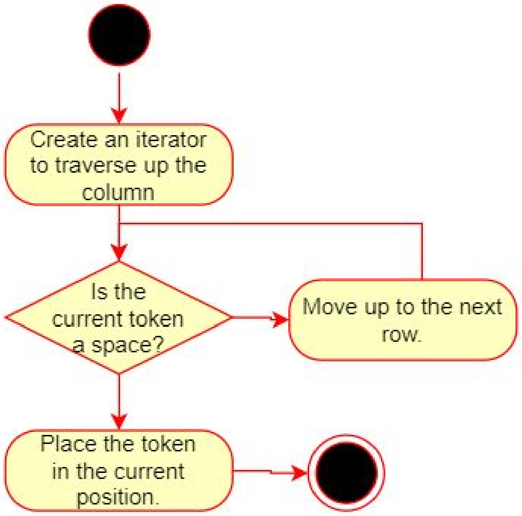
Set rows equal to _rows
Set cols equal to _cols
Set numToWin equal to _numToWin.
Create an initialize a 2d array of blank spaces

Void setRows()	Void setCols()	Void setNumToWin()
 	 	

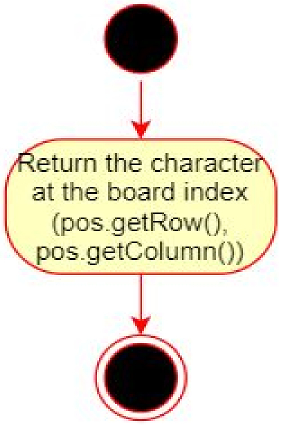
Int getNumRows()	Int getNumCols()	Int getNumToWin()
		
Int getMaxRows()	Int getMaxCols()	Int getMaxNumToWin()
		

Int getMinRows()	Int getMinCols()	Int getMinNumToWin()
		

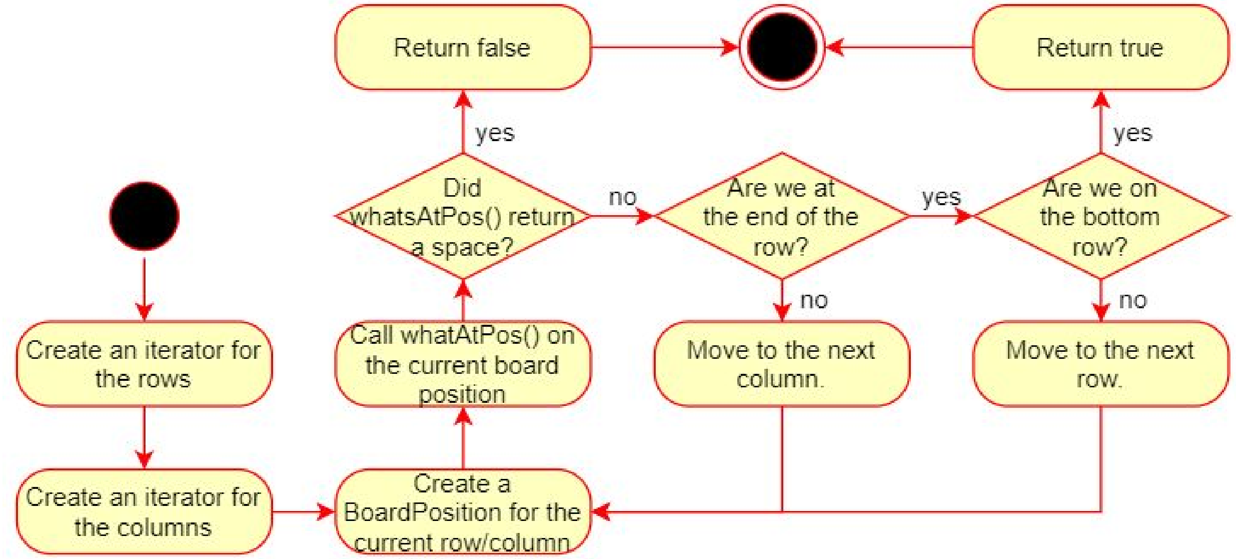
Void placeToken():


Create an iterator to traverse up the column
Is the current token a space?
Move up to the next row.
Place the token in the current position.

Char whatsAtPos():

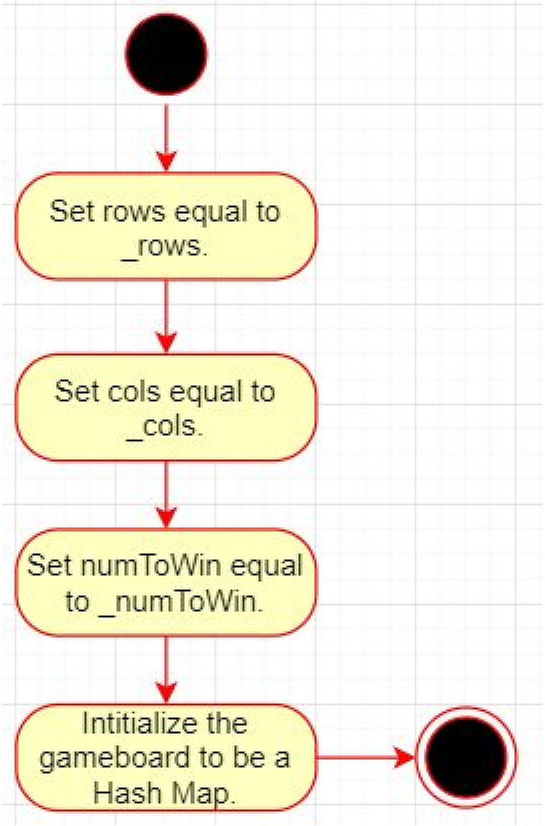

Return the character at the board index (pos.getRow(), pos.getColumn())

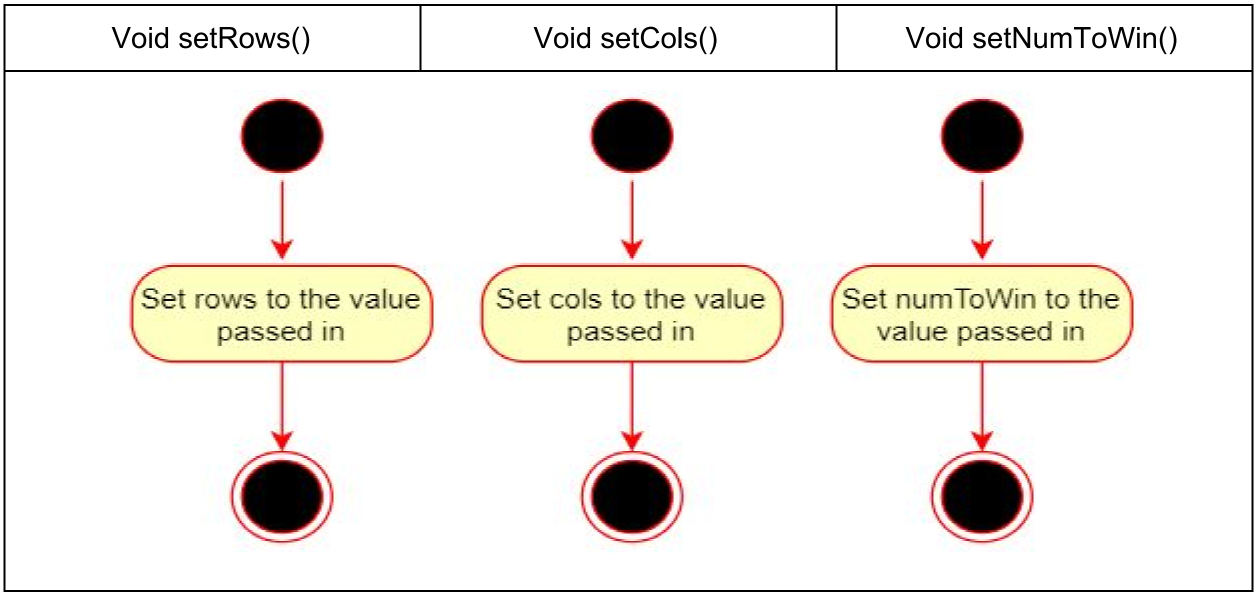
boolean checkTie():

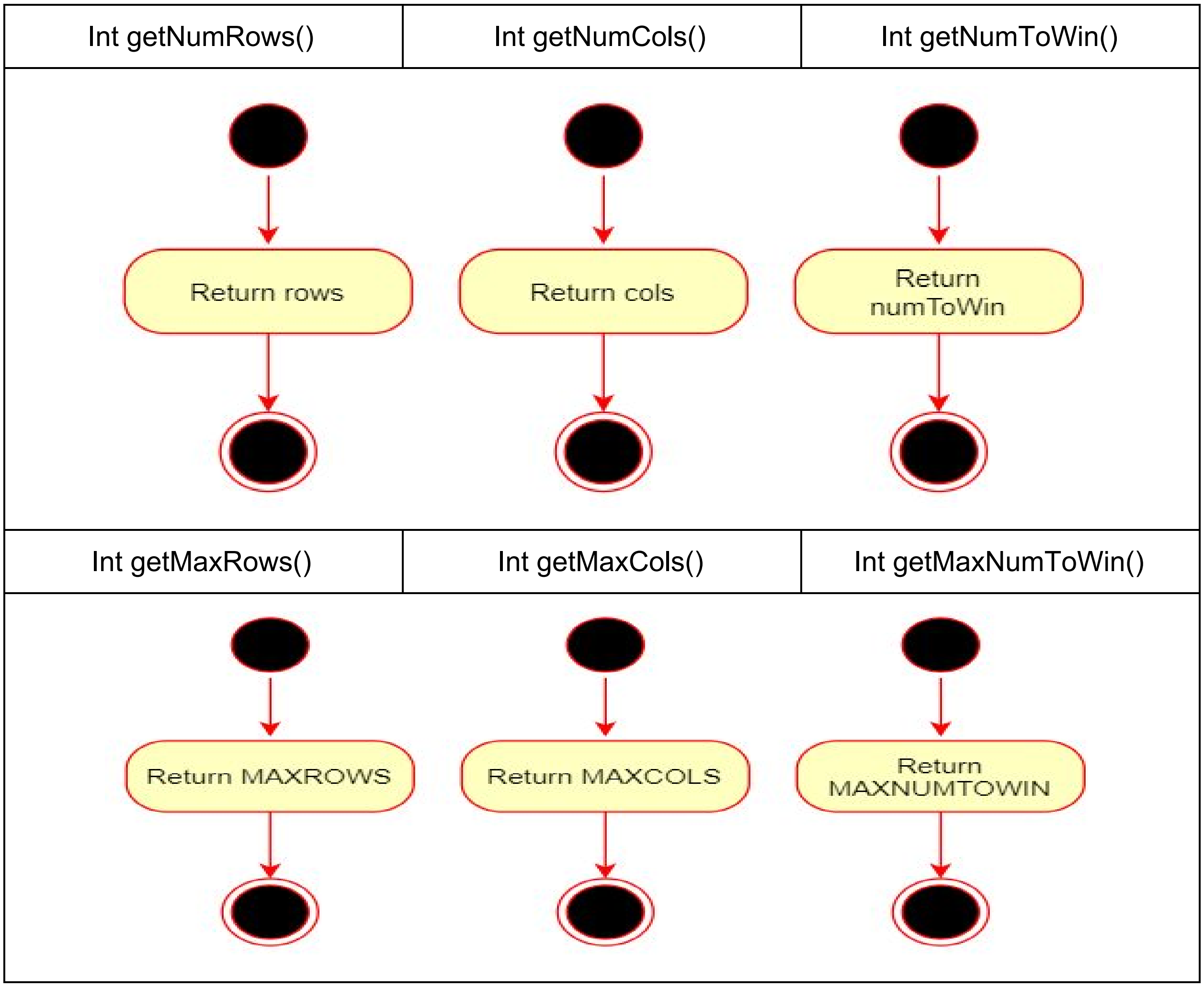

Return false
Return true
yes
yes
Did whatsAtPos() return a space?
no
Are we at the end of the row?
yes
Are we on the bottom row?
no
no
Create an iterator for the rows
Call whatAtPos() on the current board position
Move to the next column.
Move to the next row.
Create an iterator for the columns
Create a BoardPosition for the current row/column

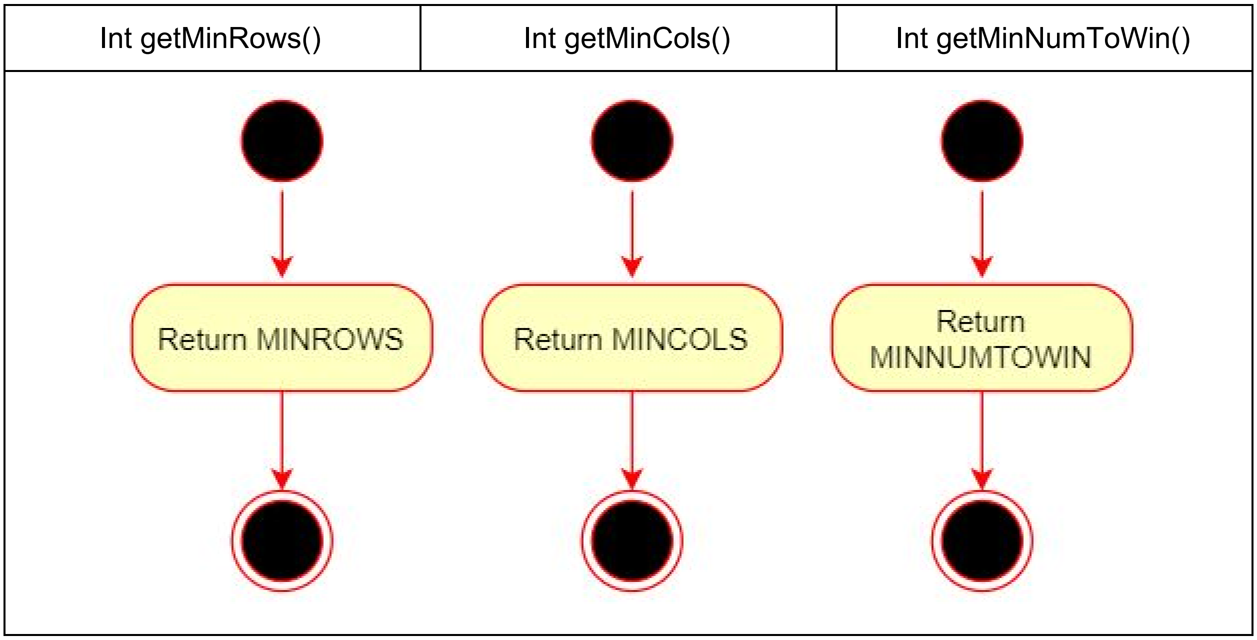
GameBoardMem:

Void GameBoardMem:

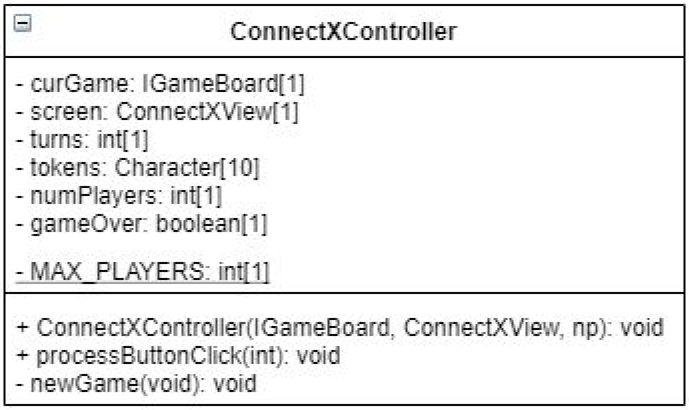

Set rows equal to _rows.
Set cols equal to _cols.
Set numToWin equal to _numToWin.
Intitialize the gameboard to be a Hash Map.


Void setRows()
Void setCols()
Void setNumToWin()
Set rows to the value passed in
Set cols to the value passed in
Set numToWin to the value passed in


Int getNumRows()
Int getNumCols()
Int getNumToWin()
Return rows
Return cols
Return numToWin
Int getMaxRows()
Int getMaxCols()
Int getMaxNumToWin()
Return MAXROWS
Return MAXCOLS
Return MAXNUMTOWIN



ConnectXController:



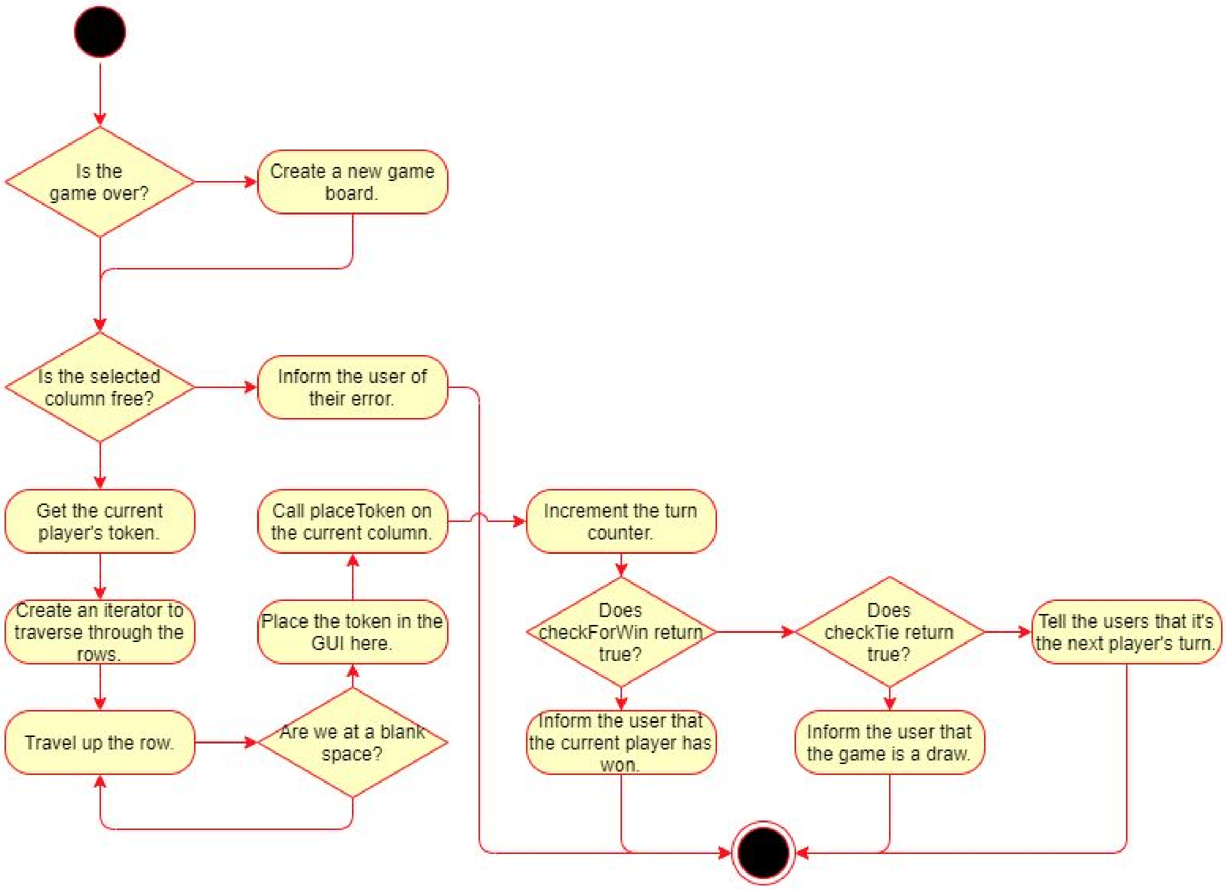
ConnectXController

- curGame: IGameBoard[1]
- screen: ConnectXView[1]
- turns: int[1]
- tokens: Character[10]
- numPlayers: int[1]
- gameOver: boolean[1]

- MAX_PLAYERS: int[1]

+ ConnectXController(IGameBoard, ConnectXView, np): void
+ processButtonClick(int): void
- newGame(void): void

public void processButtonClick(int col)


Is the game over?
Create a new game board.
Is the selected column free?
Inform the user of their error.
Get the current player's token.
Create an iterator to traverse through the rows.
Travel up the row.
Are we at a blank space?
Place the token in the GUI here.
Call placeToken on the current column.
Increment the turn counter.
Does checkForWin return true?
Inform the user that the current player has won.
Does checkTie return true?
Inform the user that the game is a draw.
Tell the users that it's the next player's turn.

Test Cases:

GameBoard(int rows, int cols, int numToWin)
GameBoardMem(int rows, int cols, int numToWin)

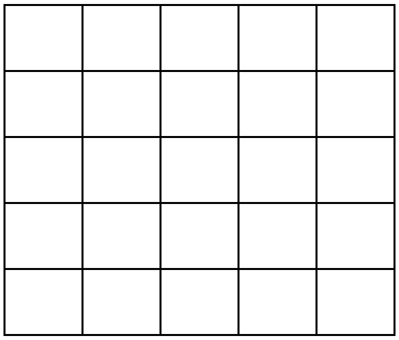
INPUT	OUTPUT	REASONING

Rows = 3
Cols = 3
numToWin = 3
	gameBoard.numRows = 3
	gameBoard.numCols = 3
	gameBoard.numToWin = 3
	Game board of size 3 x 3
		The preconditions state that the three inputs have to be greater than or equal to their respective minimum values. This tests minimum values for all three inputs.
		
		Function:
		testGameBoardMinVals
		testGameBoardMemMinVals

Rows = 100
Cols = 100
numToWin = 25
	gameBoard.numRows = 100
	gameBoard.numCols = 100
	gameBoard.numToWin = 25
	Game board of size 100 x 100
		The preconditions state that the three inputs have to be less than or equal to their respective maximum values. This tests maximum values for all three inputs.
		
		Function:
		testGameBoardMaxVals
		testGameBoardMemMaxVals

Rows = 5
Cols = 7
numToWin = 4
	gameBoard.numRows = 5
	gameBoard.numCols = 7
	gameBoard.numToWin = 4
	Game board of size 5 x 7
		This is a test to see if the constructor can create a gameboard of a non-square shape, or where the number of rows and columns are not equal.
		
		Function:
		testGameBoardUnevenVals
		testGameBoardMemUnevenVals

boolean checkIfFree(int rows, int cols, int numToWin)

INPUT	**OUTPUT**	**REASONING**
Col = 4 State: 	checkIfFree = true State of board is unchanged	This test checks to see if the function can determine if an empty column is “free”. Function: testCheckIfFreeEmpty

<table>
<tr>
<td>Col = 0
State:
<table>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>O</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
<tr><td>O</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
</table>
</td>
<td>checkIfFree = true

State of board is unchanged</td>
<td>This test checks to see if the function can determine if an empty column only has 1 remaining empty space.

Function:
testCheckIfFreeAlmostFull</td>
</tr>
<tr>
<td>Col = 0
State:
<table>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
<tr><td>O</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
<tr><td>O</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
</table>
</td>
<td>checkIfFree = false

State of board is unchanged</td>
<td>This test checks to see if the function can recognize

Function:
testCheckIfFreeFull</td>
</tr>
</table>

boolean checkHorizWin(BoardPosition pos, char p)

<table>
<tr><th>INPUT</th><th>OUTPUT</th><th>REASONING</th></tr>
<tr>
<td>Pos = (1, 0)
P = X
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>O</td><td>O</td><td>O</td></tr>
<tr><td></td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
</table>
numToWin = 4</td>
<td>checkHorizWin = true

State of board is unchanged</td>
<td>This test case checks to see if checkHorizWin can find a winner when the token is placed on the far left of the winning sequence.

testHorizLastTokenOnLeft</td>
</tr>
</table>

<table>
<tr>
<td>Pos = (3, 0)
P = X
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>O</td><td>O</td><td>O</td><td></td><td></td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td></td></tr>
</table>
numToWin = 4</td>
<td>checkHorizWin = true

State of board is unchanged</td>
<td>This test case checks to see if checkHorizWin can find a winner when the token is placed on the far right side of the winning sequence.

testHorizLastTokenOnRight</td>
</tr>
<tr>
<td>Pos = (3, 1)
P = X
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>O</td><td></td><td>O</td><td>O</td><td></td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td></td></tr>
</table>
numToWin = 4</td>
<td>checkHorizWin = true

State of board is unchanged</td>
<td>This test case checks to see if checkHorizWin can find a winner when the token is placed in the middle of the winning sequence, requiring it to traverse both ways.

testHorizLastTokenInMiddle</td>
</tr>
<tr>
<td>Pos = (3, 0)
P = X
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>O</td><td>O</td><td></td><td></td><td></td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>O</td><td>X</td></tr>
</table>
</td>
<td>checkHorizWin = false

Board state is unchanged</td>
<td>This test case checks to see if checkHorizWin will declare a sequence of length numToWin-1 wins. This should not be the case.

testHorizNonWinner</td>
</tr>
</table>

checkVertWin(BoardPosition pos, char p)

<table>
<tr><th>INPUT</th><th>OUTPUT</th><th>REASONING</th></tr>
<tr><td>Pos = (3, 3)
P = X
State:<table>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
<tr><td></td><td></td><td></td><td>X</td><td>O</td></tr>
<tr><td></td><td></td><td>O</td><td>X</td><td>O</td></tr>
</table>numToWin = 4</td><td>checkVertWin = true

Board state is unchanged</td><td>This test case checks to see if checkVerWin can find a winner when the winning sequence is on the very bottom of the column.

testVertBottomOfCol</td></tr>
<tr><td>Pos = (3, 4)
P = X
State:<table>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
<tr><td></td><td></td><td>O</td><td>X</td><td></td></tr>
<tr><td></td><td></td><td>O</td><td>X</td><td></td></tr>
<tr><td></td><td></td><td>O</td><td>X</td><td></td></tr>
<tr><td></td><td></td><td>X</td><td>O</td><td></td></tr>
</table>numToWin = 4</td><td>checkVertWin = true

Board state is unchanged</td><td>This test case checks to see if checkVertWin can find a winner when the winning sequence is on the very top of the column.

testVertTopOfCol</td></tr>
<tr><td>Pos = (3, 4)
P = X
State:<table>
<tr><td></td><td></td><td></td><td>X</td><td></td></tr>
<tr><td></td><td></td><td></td><td>O</td><td></td></tr>
<tr><td></td><td></td><td>O</td><td>X</td><td></td></tr>
<tr><td></td><td></td><td>O</td><td>X</td><td></td></tr>
<tr><td></td><td></td><td>O</td><td>X</td><td>X</td></tr>
</table>numToWin = 4</td><td>checkVertWin = false

Board state is unchanged</td><td>This test case checks to see if checkVertWin can make the correct decision regarding a winning sequence that has been interrupted by a non-matching character.

testVertNonWinnerSplit</td></tr>
</table>

<table>
<tr><td>Pos = (0, 2
P = X
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>X</td><td></td><td></td><td></td><td></td></tr><tr><td>X</td><td>O</td><td></td><td></td><td></td></tr><tr><td>X</td><td>O</td><td></td><td></td><td></td></tr></table>numToWin = 4</td><td>checkVertWin = false

Board state is unchanged</td><td>This test case checks to see if checkVertWin can make the correct decision regarding a non-winning sequence of length numToWin-1.

testVertNonWinnerTooShort</td></tr>
</table>

boolean checkDiagWin(BoardPosition pos, char p)

<table>
<tr><th>INPUT</th><th>OUTPUT</th><th>REASONING</th></tr>
<tr><td>Pos = (3, 3)
P = X
State:
<table><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td>X</td><td></td></tr><tr><td></td><td></td><td>X</td><td>O</td><td></td></tr><tr><td>O</td><td>X</td><td>X</td><td>X</td><td></td></tr><tr><td>X</td><td>O</td><td>O</td><td>O</td><td></td></tr></table>numToWin = 4</td><td>checkDiagWin = true

Board state is unchanged</td><td>This test case checks to see if checkDiagWin can make the correct decision regarding a winning sequence that begins in the left bottom corner of the board and goes up and to the right.

testDiagEndNW_SE</td></tr>
<tr><td>Pos = (1, 3)
P = X
State:
<table><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td>X</td><td></td><td></td><td></td></tr><tr><td></td><td>O</td><td>X</td><td></td><td></td></tr><tr><td></td><td>O</td><td>O</td><td>X</td><td></td></tr><tr><td>X</td><td>X</td><td>O</td><td>O</td><td>X</td></tr></table>numToWin = 4</td><td>checkDiagWin = true

Board state is unchanged</td><td>This test case checks to see if checkDiagWin can make the correct decision regarding a winning sequence that begins in the right bottom corner of the board and goes up and to the left.

testDiagEndNE_SW</td></tr>
</table>

<table>
<tr>
<td>Pos = (2, 2)
P = X
State:
<table><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td>X</td><td></td></tr><tr><td></td><td></td><td>X</td><td>O</td><td></td></tr><tr><td></td><td>X</td><td>O</td><td>X</td><td></td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td>O</td></tr></table>numToWin = 4</td>
<td>checkHorizWin = true

Board state is unchanged</td>
<td>This test case checks to see if checkDiagWin can make the correct decision regarding a winning sequence in which the winning token is placed in the middle of the southwest - northeast diagonal.

testDiagMidNW_SE</td>
</tr>
<tr>
<td>Pos = (2, 2)
P = X
State:
<table><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td>X</td><td></td><td></td><td></td></tr><tr><td></td><td>O</td><td>X</td><td></td><td></td></tr><tr><td></td><td>X</td><td>O</td><td>X</td><td></td></tr><tr><td>O</td><td>O</td><td>X</td><td>O</td><td>X</td></tr></table>numToWin = 4</td>
<td>checkHorizWin = true

Board state is unchanged</td>
<td>This test case checks to see if checkDiagWin can make the correct decision regarding a winning sequence in which the winning token is placed in the middle of the southeast-northwest diagonal.

testDiagMidNE_SW</td>
</tr>
<tr>
<td>Pos = (2, 2)
P = X
State:
<table><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td>X</td><td></td><td></td><td></td></tr><tr><td></td><td>O</td><td>X</td><td>O</td><td></td></tr><tr><td></td><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr></table>numToWin = 4</td>
<td>checkHorizWin = true

Board state is unchanged</td>
<td>This test case checks to see if checkDiagWin can make the correct decision regarding a winning sequence in which the winning token is placed at an intersection between two diagonal sequences. This will prove as to whether or not checkDiagWin will be able to traverse all four directions in one check, and declare the winner as the southeast - northwest diagonal.

testDiagIntersectNE_SW</td>
</tr>
</table>

<table>
<tr><td>Pos = (2, 2)
P = X
State:
<table><tr><td></td><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td>X</td><td></td></tr><tr><td></td><td>O</td><td>X</td><td>O</td><td></td></tr><tr><td>O</td><td>X</td><td>O</td><td>X</td><td></td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr></table>numToWin = 4</td><td>checkHorizWin = true

Board state is unchanged</td><td>This test case checks to see if checkDiagWin can make the correct decision regarding a winning sequence in which the winning token is placed at an intersection between two diagonal sequences. This will prove as to whether or not checkDiagWin will be able to traverse all four directions in one check, and declare the winner as the southwest-northeast diagonal.

testDiagIntersectNW_SE</td></tr>
<tr><td>Pos = (2, 2)
P = X
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>X</td><td></td><td></td></tr><tr><td>O</td><td>X</td><td>X</td><td>O</td><td></td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td></td></tr></table>numToWin = 4</td><td>checkHorizWin = false

Board state is unchanged</td><td>This test case checks to see if checkDiagWin can make the correct decision regarding an incomplete winning sequence. The token placed at (2, 2)

testDiagNoWinner</td></tr>
</table>

boolean checkTie()

<table>
<tr><th>INPUT</th><th>OUTPUT</th><th>REASONING</th></tr>
<tr><td>State:
<table><tr><td></td><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><td></td><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><td></td><td>O</td><td>X</td><td>O</td><td>X</td></tr><tr><td></td><td>O</td><td>X</td><td>O</td><td>X</td></tr><tr><td></td><td>X</td><td>O</td><td>X</td><td>O</td></tr></table></td><td>checkTie = false

Board state is unchanged</td><td>This test case checks to see if checkTie can make the correct decision regarding all but one column being full. That one extra column is all the way on the left completely empty.</td></tr>
</table>

<table>
<tr>
<td>State:
<table><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><tr><td>O</td><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><td>O</td><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr></table></td>
<td>checkTie = false

Board state is unchanged</td>
<td>This test case checks to see if checkTie can make the correct decision regarding all but the top row being full. The top row is completely empty.</td>
</tr>
<tr>
<td>State:
<table><tr><td>X</td><td>O</td><td>X</td><td>O</td><td></td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr><tr><td>O</td><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><td>O</td><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr></table></td>
<td>checkTie = false

Board state is unchanged</td>
<td>This test case checks to see if checkTie can make the correct decision regarding a single space in the top row being empty.</td>
</tr>
<tr>
<td>State:
<table><tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr><tr><td>O</td><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><td>O</td><td>X</td><td>O</td><td>X</td><td>O</td></tr><tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr></table></td>
<td>checkTie = true

Board state is unchanged</td>
<td>This test case checks to see if checkTie can make the correct decision regarding an entirely full board, which automatically results in a tie.</td>
</tr>
</table>

char whatsAtPos(BoardPosition pos)

<table>
<tr><th>INPUT</th><th>OUTPUT</th><th>REASONING</th></tr>
<tr><td>pos = (2, 2)
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></td><td></td><td></td><td></td><td></td></tr>
</table></td><td>whatsAtPos = ‘ ‘

Board state is unchanged</td><td>This test ensures that whatsAtPos will return a blank space whenever the passed in board space hasn’t been used yet.

Function: testWhatsAtPosBlankBoard</td></tr>
<tr><td>pos = (0, 0)
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></td><td></td><td></td><td></td></tr>
</table></td><td>whatsAtPos = ‘X‘

Board state is unchanged</td><td>This test ensures that whatsAtPos will return the proper character at the bottom left of the board.

testWhatsAtPosOneChar</td></tr>
<tr><td>pos = (4, 0)
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></td><td></td><td></td><td>O</td></tr>
</table></td><td>whatsAtPos = ‘O’

Board state is unchanged</td><td>This test ensures that whatsAtPos will return the proper character at the bottom right of the board.

testWhatsAtPosFarChar</td></tr>
</table>

<table>
<tr>
<td>pos = (0, 4)
State:
<table><tr><td>X</td><td></td><td></td><td></td><td></td></tr><tr><td>O</td><td></td><td></td><td></td><td></td></tr><tr><td>X</td><td></td><td></td><td></td><td></td></tr><tr><td>O</td><td></td><td></td><td></td><td></td></tr><tr><td>X</td><td></td><td></td><td></td><td>O</td></tr></table></td>
<td>whatsAtPos = ‘X’

Board state is unchanged</td>
<td>This test ensures that whatsAtPos will return the proper character at the bottom right of the board.</td>
</tr>
<tr>
<td>pos = (4, 4)
State:
<table><tr><td>X</td><td></td><td></td><td></td><td>O</td></tr><tr><td>O</td><td></td><td></td><td></td><td>X</td></tr><tr><td>X</td><td></td><td></td><td></td><td>O</td></tr><tr><td>O</td><td></td><td></td><td></td><td>X</td></tr><tr><td>X</td><td></td><td></td><td></td><td>O</td></tr></table></td>
<td>whatsAtPos = ‘O’

Board state is unchanged</td>
<td>This test ensures that whatsAtPos will return the proper character at the top right of the board.

testWhatsAtPosFarColumnFull</td>
</tr>
</table>

boolean isPlayerAtPos(BoardPosition pos, char p)

<table>
<tr><th>INPUT</th><th>OUTPUT</th><th>REASONING</th></tr>
<tr>
<td>pos = (0, 0)
p= X
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></td><td></td><td></td><td></td><td></td></tr></table></td>
<td>isPlayerAtPos = false

Board state is unchanged</td>
<td>This test checks to see if isPlayerAtPos can handle checking a blank space in the gameBoard.</td>
</tr>
</table>

<table>
<tr>
<td>pos = (0, 0)
p= X
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></td><td></td><td></td><td></td></tr></table></td>
<td>isPlayerAtPos = true

Board state is unchanged</td>
<td>This test checks to see if isPlayerAtPos can correctly determine the character at the bottom of the board and successfully make the comparison.

testIsPlayerAtPosOneToken</td>
</tr>
<tr>
<td>pos = (0, 4)
P = X
State:
<table><tr><td>D</td><td></td><td></td><td></td><td></td></tr><tr><td>C</td><td></td><td></td><td></td><td></td></tr><tr><td>B</td><td></td><td></td><td></td><td></td></tr><tr><td>A</td><td></td><td></td><td></td><td></td></tr><tr><td>X</td><td></td><td></td><td></td><td></td></tr></table></td>
<td>isPlayerAtPos = false

Board state is unchanged</td>
<td>This test checks to see if isPlayerAtPos can correctly determine the character at the top of the board and successfully recognize that two characters are not the same.

testIsPlayerAtPosOneFullColumnFalse</td>
</tr>
<tr>
<td>pos = (0, 4)
P = D
State:
<table><tr><td>D</td><td></td><td></td><td></td><td>B</td></tr><tr><td>C</td><td></td><td></td><td></td><td>A</td></tr><tr><td>B</td><td></td><td></td><td></td><td>O</td></tr><tr><td>A</td><td></td><td></td><td></td><td>X</td></tr><tr><td>X</td><td></td><td></td><td></td><td>O</td></tr></table></td>
<td>isPlayerAtPos = true

Board state is unchanged</td>
<td>This test checks to see if isPlayerAtPos can correctly determine the character at the top of the board and successfully recognize that two characters are the same. This also tests for characters that are not X or O.

testIsPlayerAtPosOneFullColumnTrue</td>
</tr>
</table>

<table>
<tr>
<td>pos = (2, 2)
p = X
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>D</td><td>E</td><td></td><td></td></tr><tr><td></td><td>A</td><td>B</td><td>C</td><td></td></tr><tr><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td></tr></table></td>
<td>isPlayerAtPos = false

Board state is unchanged</td>
<td>This test checks to see if isPlayerAtPos can correctly handle a blank space at the top of a column. Since the precondition is p != ‘ ‘, this should always return false.

testIsPlayerAtPosMiddleOfBoard</td>
</tr>
</table>

void placeToken(char p, int c)

<table>
<tr><th>INPUT</th><th>OUTPUT</th><th>REASONING</th></tr>
<tr>
<td>p = X
c = 0
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></td><td></td><td></td><td></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></td><td></td><td></td><td></td></tr></table></td>
<td>This test checks to see if placeToken places the token in the correct column. This will also be the first token in the column.

Function:
testPlaceTokenFirstTokenOnBoard</td>
</tr>
<tr>
<td>p = O
c = 4
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></td><td></td><td></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></td><td></td><td></td><td>O</td></tr></table></td>
<td>This test case checks to see if placeToken works in the highest numbered column. This is also the first token in that column.

testPlaceTokenFirstTokenInFarColumn</td>
</tr>
</table>

<table>
<tr>
<td>p = O
c = 0
State:<table>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
<tr><td>O</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td>O</td></tr>
</table></td>
<td>State:<table>
<tr><td>O</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
<tr><td>O</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td>O</td></tr>
</table></td>
<td>This test case checks to see if placeToken works when a column is almost full. It should place the final token in the column.

testPlaceTokenFinalTokenInFirstColumn</td>
</tr>
<tr>
<td>p = O
c = 4
State:<table>
<tr><td>O</td><td></td><td></td><td></td><td></td></tr>
<tr><td>X</td><td></td><td></td><td></td><td>X</td></tr>
<tr><td>O</td><td></td><td></td><td></td><td>O</td></tr>
<tr><td>X</td><td></td><td></td><td></td><td>X</td></tr>
<tr><td>X</td><td></td><td></td><td></td><td>O</td></tr>
</table></td>
<td>State:<table>
<tr><td>O</td><td></td><td></td><td></td><td>O</td></tr>
<tr><td>X</td><td></td><td></td><td></td><td>X</td></tr>
<tr><td>O</td><td></td><td></td><td></td><td>O</td></tr>
<tr><td>X</td><td></td><td></td><td></td><td>X</td></tr>
<tr><td>X</td><td></td><td></td><td></td><td>O</td></tr>
</table></td>
<td>This test case checks to see if placeToken works on the highest-ordered spot in the board: (numRows -1, numCols-1).

testPlaceTokenFinalTokenInFinalColumn</td>
</tr>
<tr>
<td>p = X
c = 2
State:<table>
<tr><td>X</td><td>O</td><td></td><td>O</td><td>X</td></tr>
<tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr>
<tr><td>O</td><td>X</td><td>O</td><td>X</td><td>O</td></tr>
<tr><td>O</td><td>X</td><td>O</td><td>X</td><td>O</td></tr>
<tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr>
</table></td>
<td>State:<table>
<tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr>
<tr><td>X</td><td>O</td><td>X</td><td>O</td><td>X</td></tr>
<tr><td>O</td><td>X</td><td>O</td><td>X</td><td>O</td></tr>
<tr><td>O</td><td>X</td><td>O</td><td>X</td><td>O</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 checks to see if placeToken can place the final token in the gameboard, in the final remaining column.</td>
</tr>
</table>