CPSC 2150 Project Four Report

Jake Macdonald

Functional Requirements:

1. As a player I can enter the desired column and row location for my marker to progress the game.
2. As a player I need to have a unique marker type to distinguish myself.
3. As a player I need the marker type to switch value for every other move to make this a versus game.
4. As a player up to 10 players may register into a game to make it competitive.
5. As a player two or more players must be playing the game to make it competitive.
6. As a player, players should place markers in the same ordering as player symbol selection.
7. As a player I must be able to specify the dimensions of the board for the board to be dynamic.
8. As a player the dimensions of a board must be at maximum 100 X 100 to prevent absurd playthroughs
9. As a player the dimensions of a board must be at minimum 3 X 3 to prevent absurd playthroughs
10. As a player I need indication of inappropriate input so that I may correct the mistake.
11. As a player I need an updating display of the board so I may watch the game progress.
12. As a player I need to know when win conditions have been satisfied so the game may conclude.
13. As a player I need to know when the game has been drawn so I may be aware to restart the game.
14. As a player I need to be prompted after concluding to game to begin another, so I don't need to rerun the application.
15. As a player I need the ability to place and store a marker at desired positions to play the game.
16. As a player I need the board to be designed withing a grid of specified size to play a specific variety of tic-tac-toe.
17. As a player I must be able to indicate how many rows to win for games to be dynamic.
18. As a player the number of markers I enter must be an element within the range [3,25] to prevent absurd playthroughs.
19. As a player I should be given the option to chose between a time or memory efficient run of the game so I can play better.
20. As a player I need the win condition to be a line of specified number of adjacent similar markings so to play a specific variety of tic-tac-toe.
21. As a player I need the gameboard display to be expressed in a readable manner so I may bear greater witness.
22. As a player, the unmarked locations of the board should possess the default value, ' ' so it may be clear where available places positions are.
23. As a player, the placed marker value should be any value entered by players that are capitalized alphabetical characters.

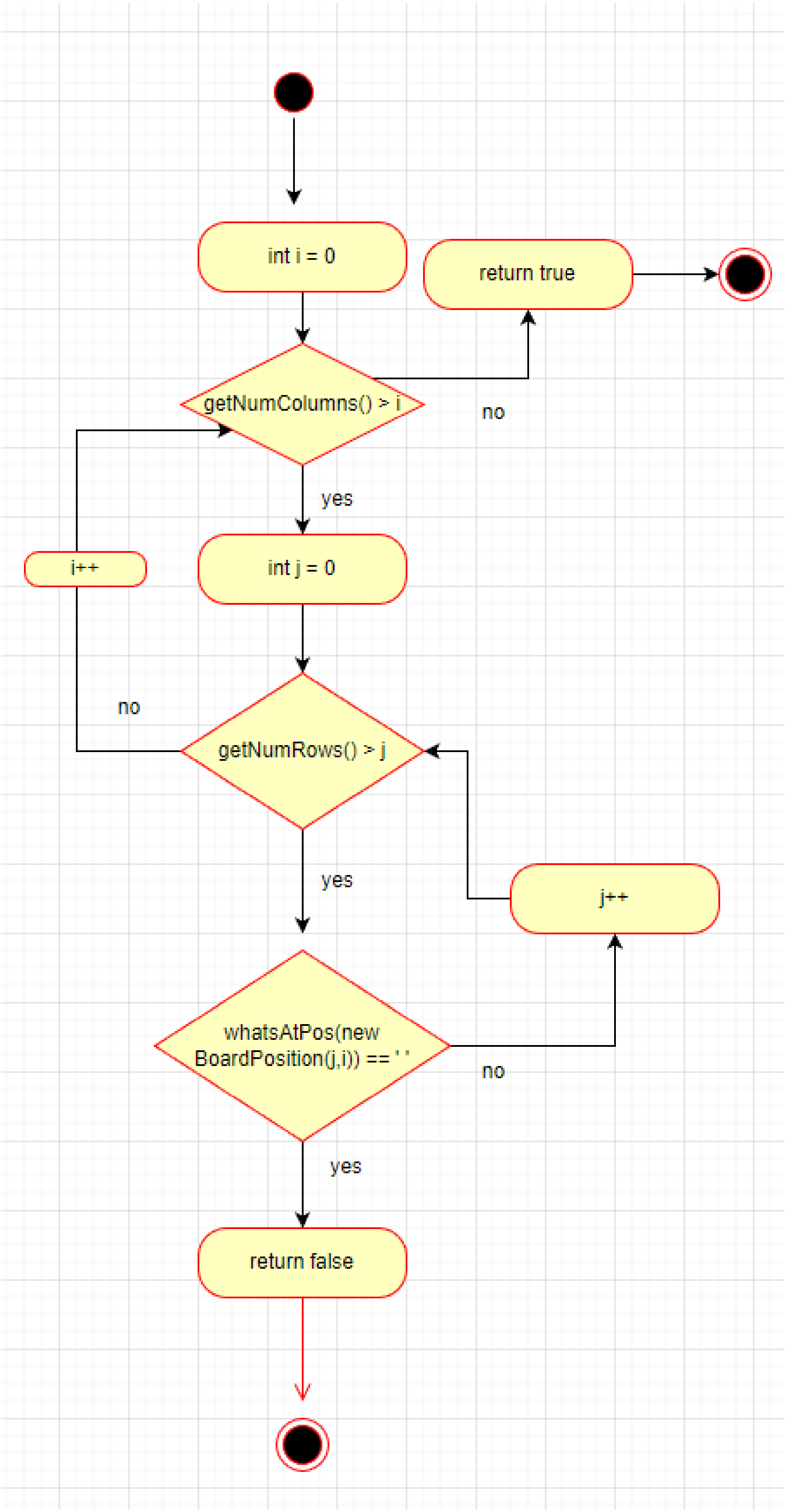
Non-Functional Requirements

1. The application must be developed in Java.
2. The application must function in the Ubuntu v.20 environment.
3. The class, GameScreen will possess the only main function of the program.
4. The application must exclusively use three classes: GameBoard, GameBoardMem, GameScreen, and BoardPosition.
5. The application must exclusively use three classes: GameBoard, GameBoardMem, GameScreen, and BoardPosition.
6. The IGameBoard interface must be used.
7. The AbsGameBoard abstract class must be used for the overriding toString
8. The class GameBoard must exclusively use methods prescribed within assignment documentation.
9. Attributes of the class, BoardPosition, must exclusively be accessible by getter methods.
10. All U/I interaction will be exclusively preformed within the GameScreen method.
11. BoardPosition will have only one constructor method.
12. BoardPosition attributes may only be set within the constructor.
13. BoardPosition must have a methods overriding the equals() and toString() methods.
14. All attributes of GameBoard must be private unless they are static and final.
15. Gameboard is of size user inputted size
    - I think this is more of a functional requirement than non because this requirement is made explicitly apparent while in use. But I got points off last time for this not being here.
16. GameBoard will extend AbsGameboard.
17. GameBoardMem will extend IGameboard.
18. GameBoard will implement IGameboard.
19. GameBoardMem will implement AbsGameboard.
20. AbsGameBoard implements IGameBoard.
21. Board element (0,0) should be top position of the board
22. No dead code should be present in the project
23. Makefile should have targets : default, run, and clean
24. Two implementations of IGameBoard must be usable, both possessing strength and weaknesses.
25. Have 3 distinct test cases for the constructor
26. Have 3 distinct test cases for checkSpace
27. Have 4 distinct test cases for checkHorizontalWin
28. Have 4 distinct test cases for checkVerticalWin
29. Have 7 distinct test cases for checkDiagonalWin
30. Have 4 distinct test cases for checkForDraw
31. Have 5 distinct test cases for whatsAtPos
32. Have 5 distinct test cases for isPlayerAtPos
33. Have 5 distinct test cases for placeMarker

makefile instructions:

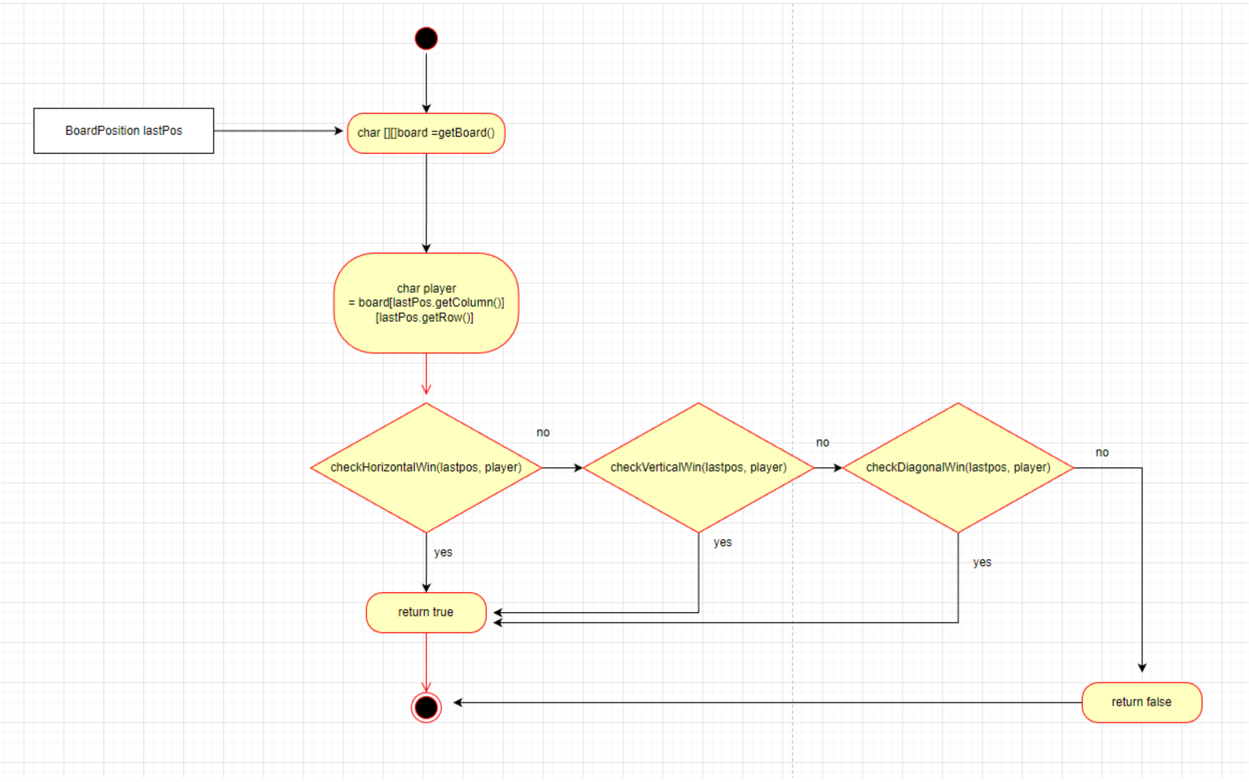
- **make:** Compiles GameBoard, AbsGameBoard, IGameBoard, BoardPosition, and GameScreen
- **make run:** Executes GameScreen.class
- **make clean:** Deletes the GameScreen.class file, and deletes all class files in the models directoy
- **make zip:** zips all files needed for project submission

UML activity Diagrams:

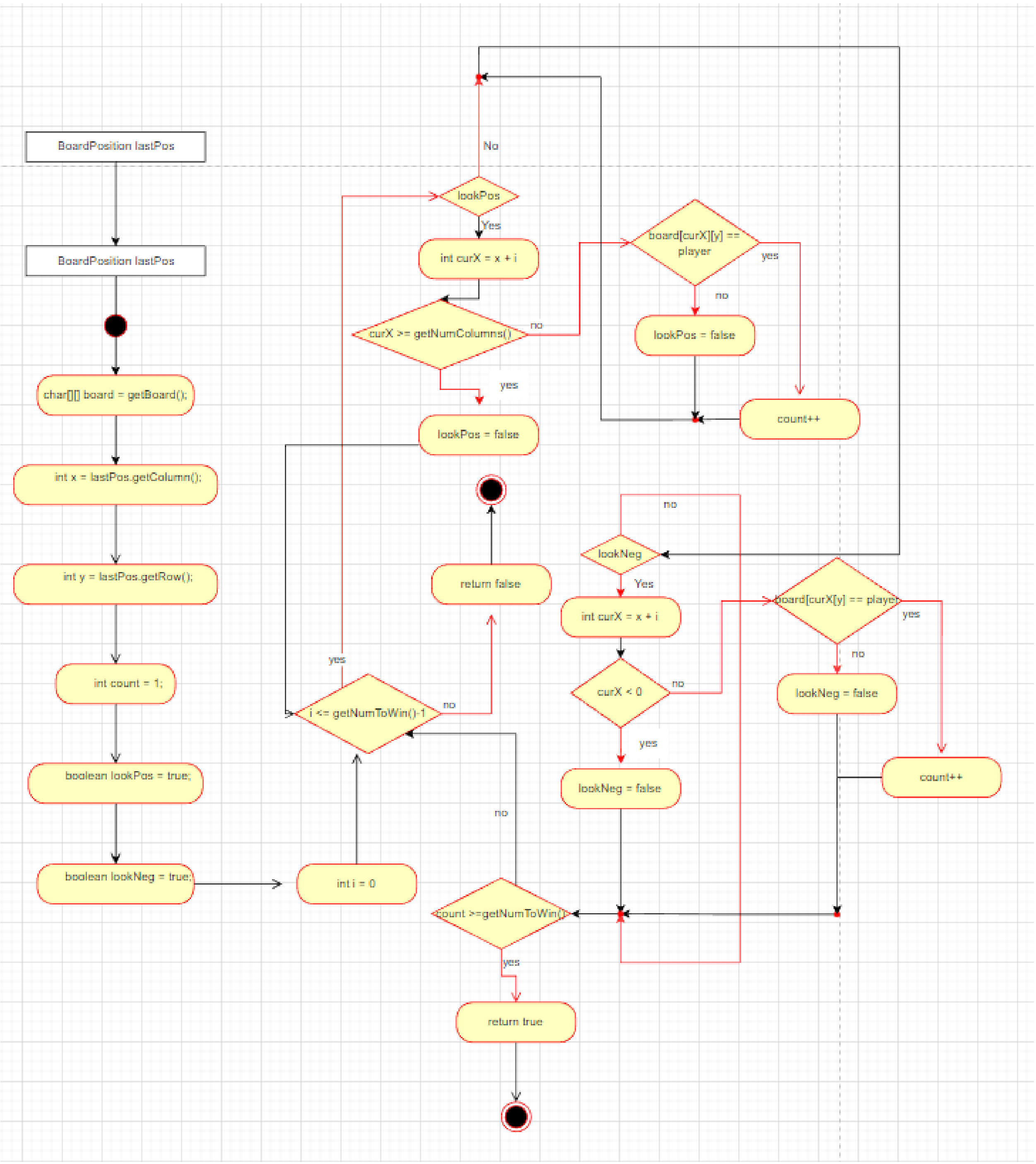


checkForDraw

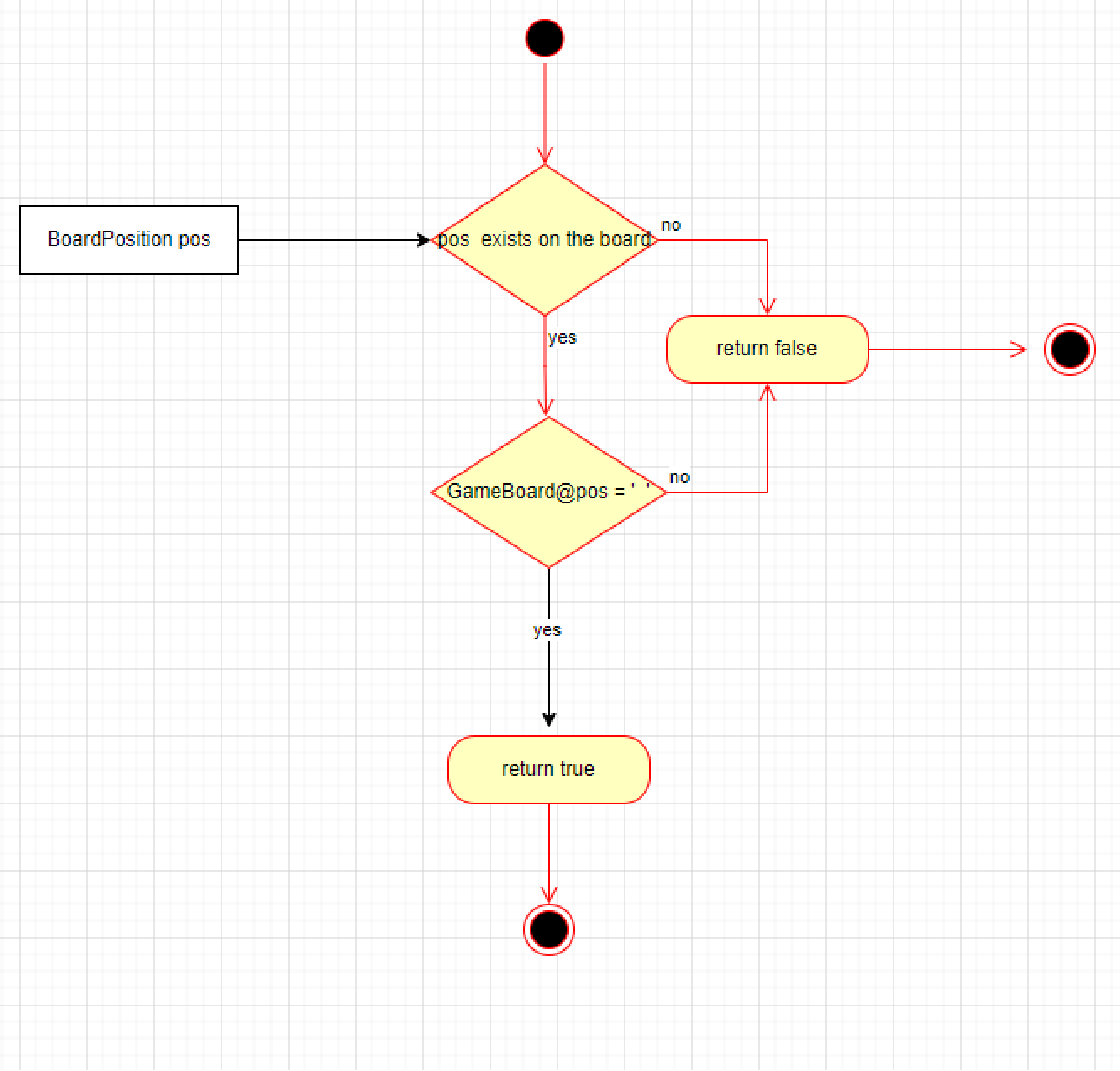
checkForWin


BoardPosition lastPos
char [][]board =getBoard()
char player
= board[lastPos.getColumn()]
[lastPos.getRow()]
checkHorizontalWin(lastpos, player)
no
checkVerticalWin(lastpos, player)
no
checkDiagonalWin(lastpos, player)
no
yes
yes
yes
return true
return false

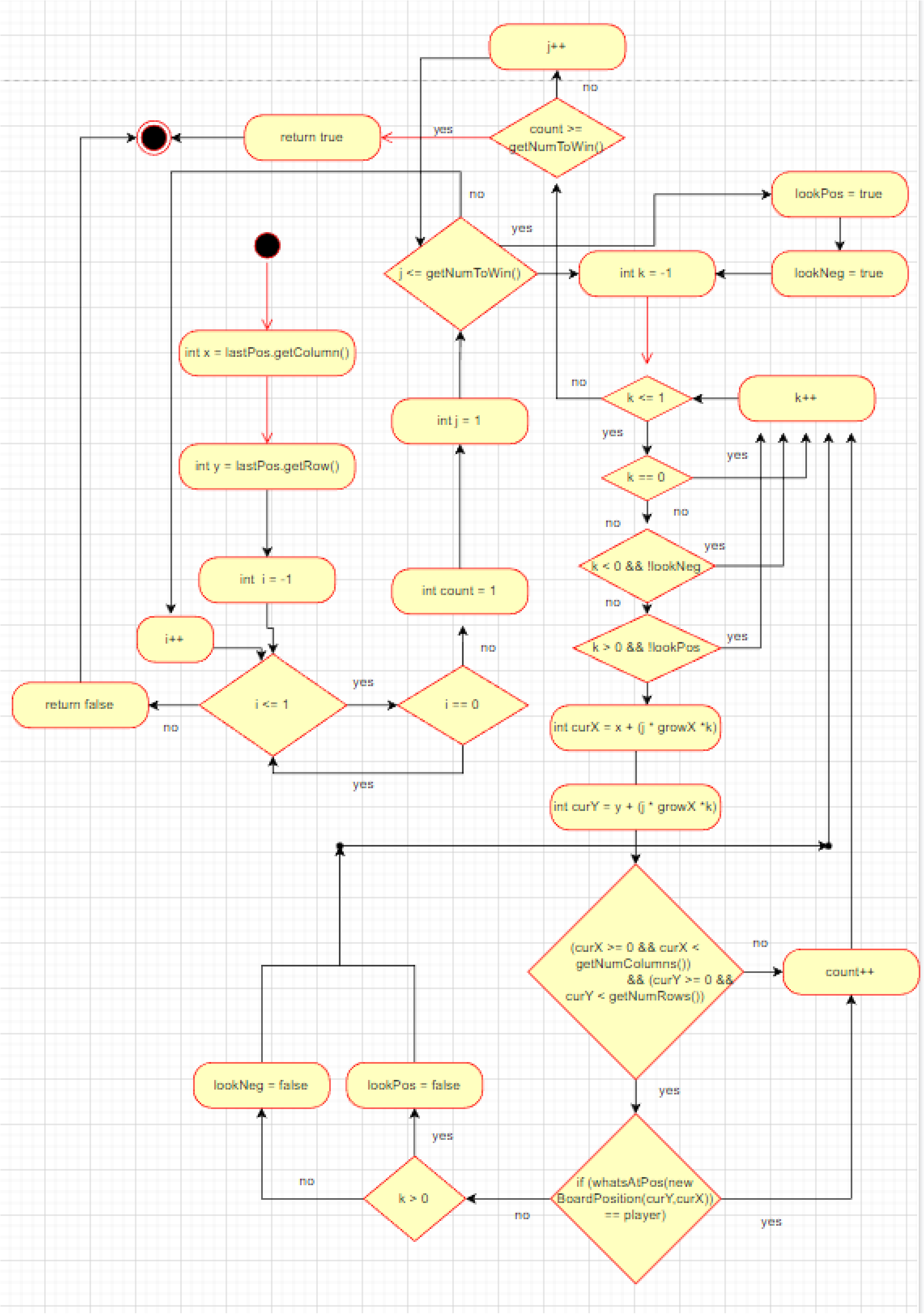
CheckHorizontalWin



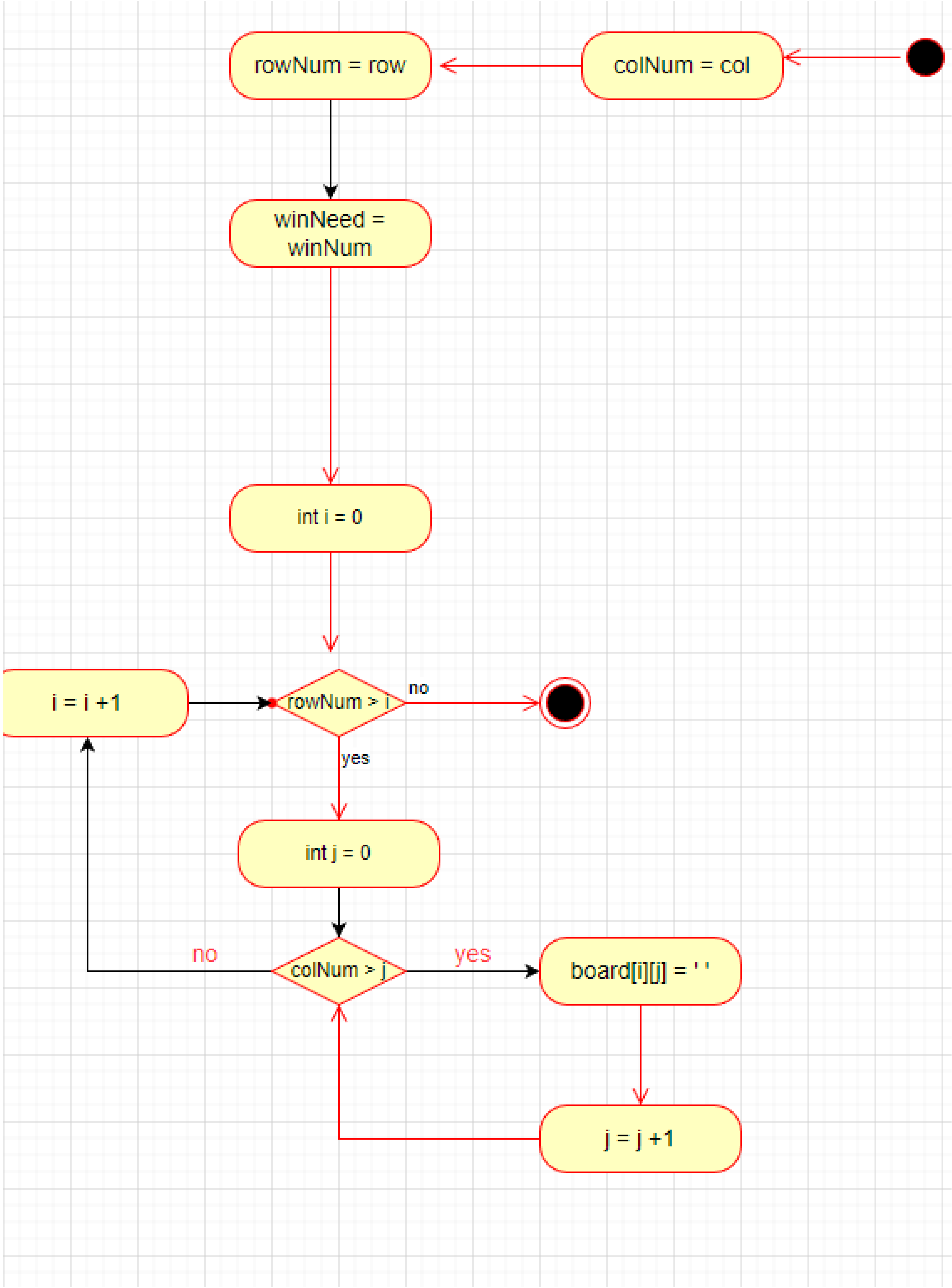
checkSpace


BoardPosition pos
pos exists on the board
no
yes
return false
GameBoard@pos = ' '
no
yes
return true

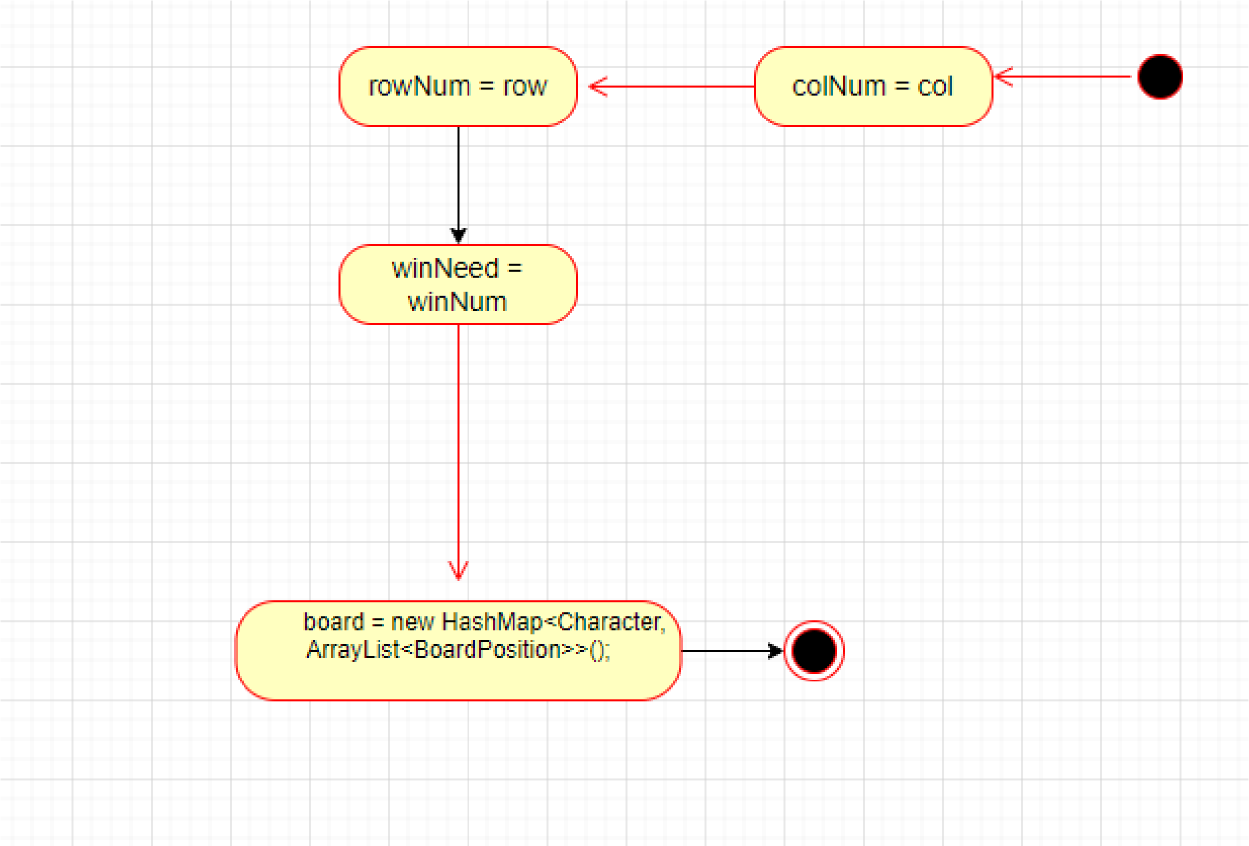
CheckDiagonalWin

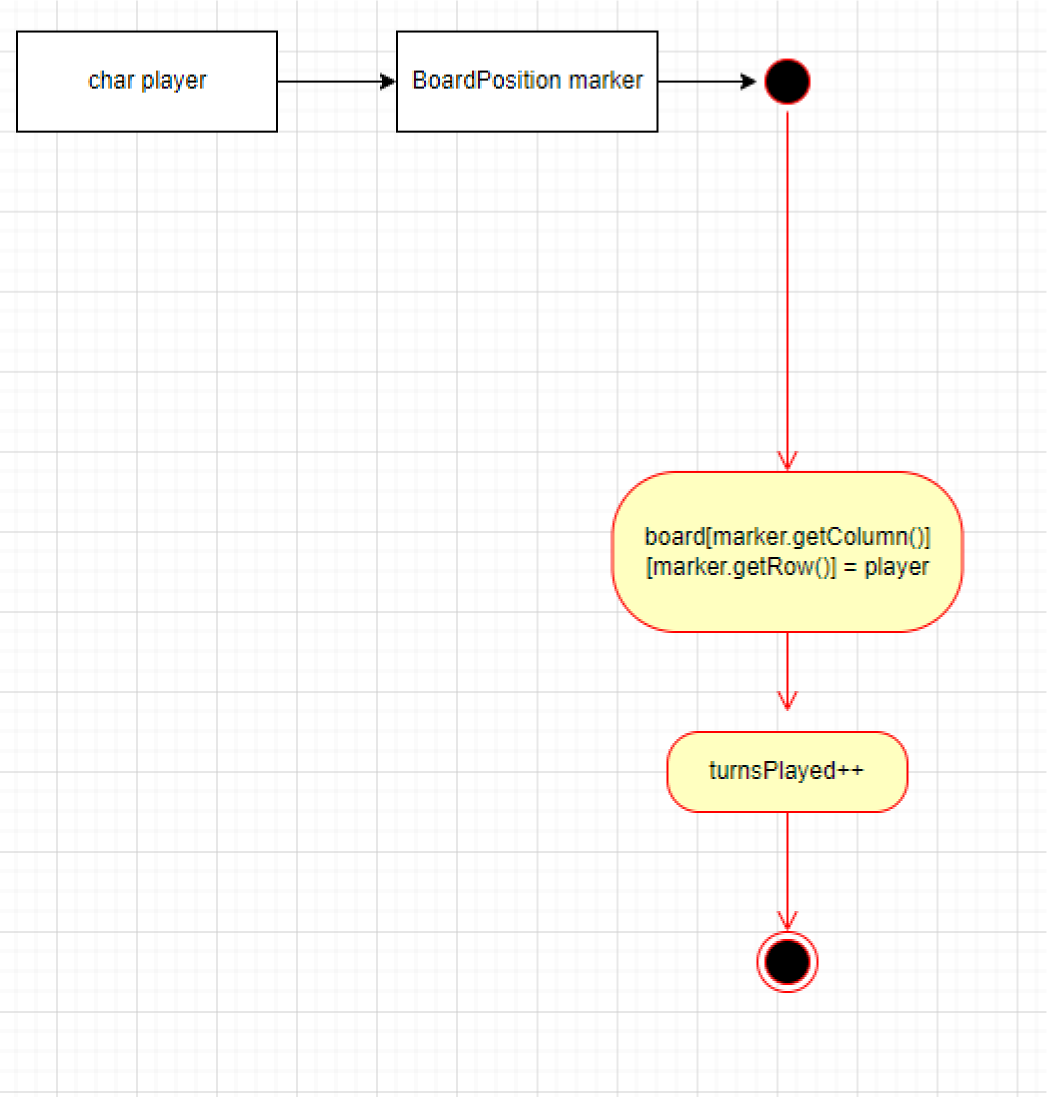


Gameboard(constructor)

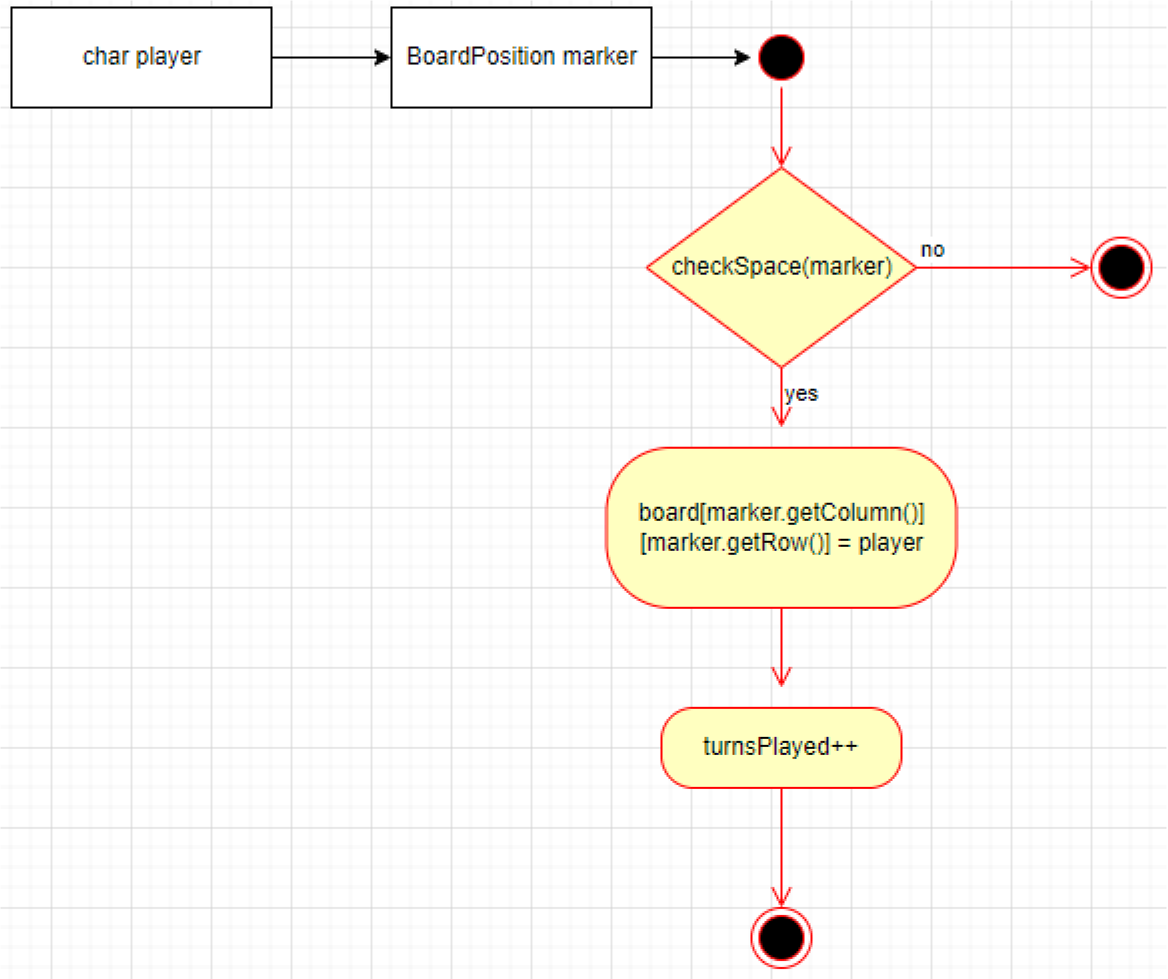


GameBoardMem(constructor)

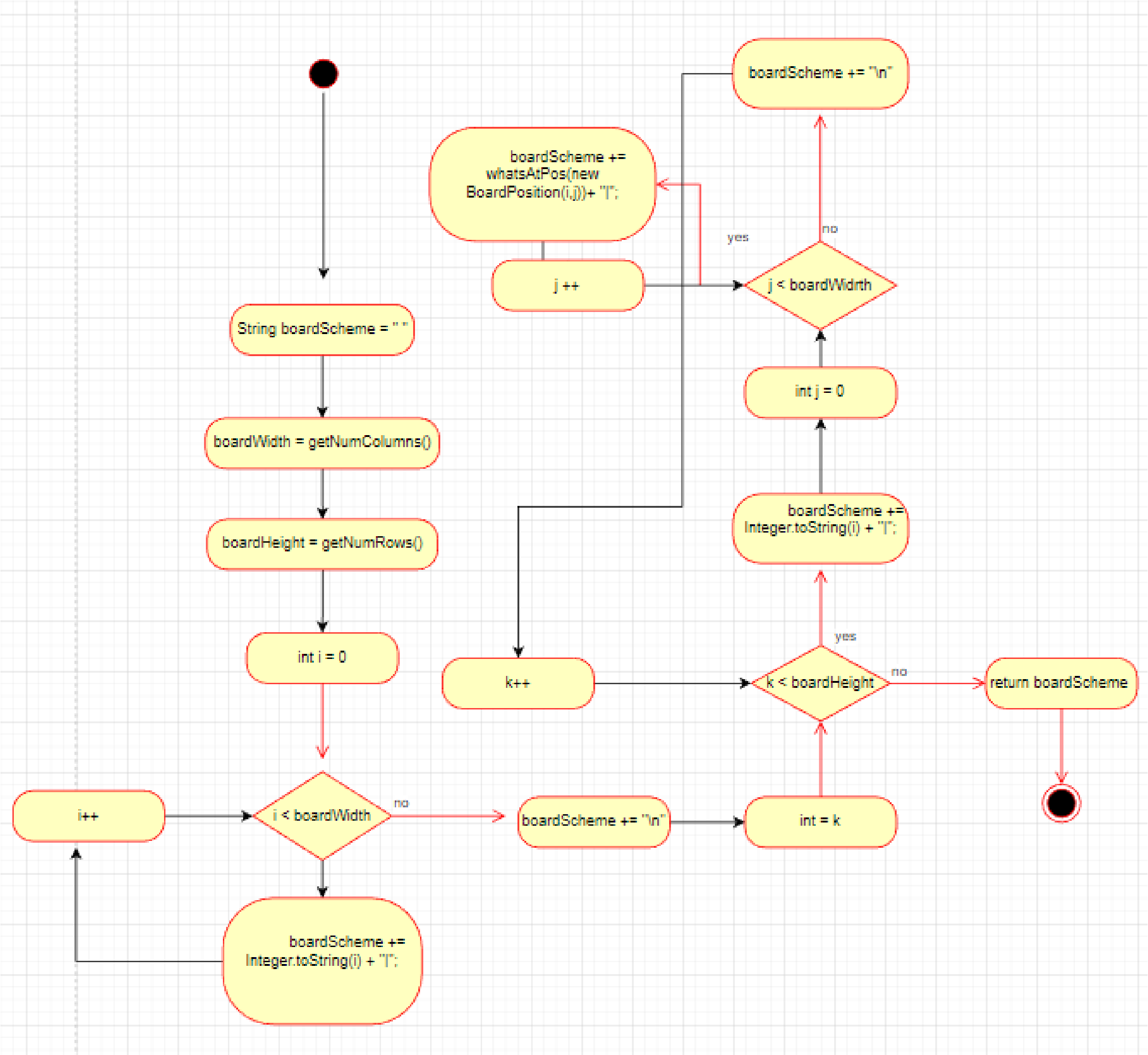




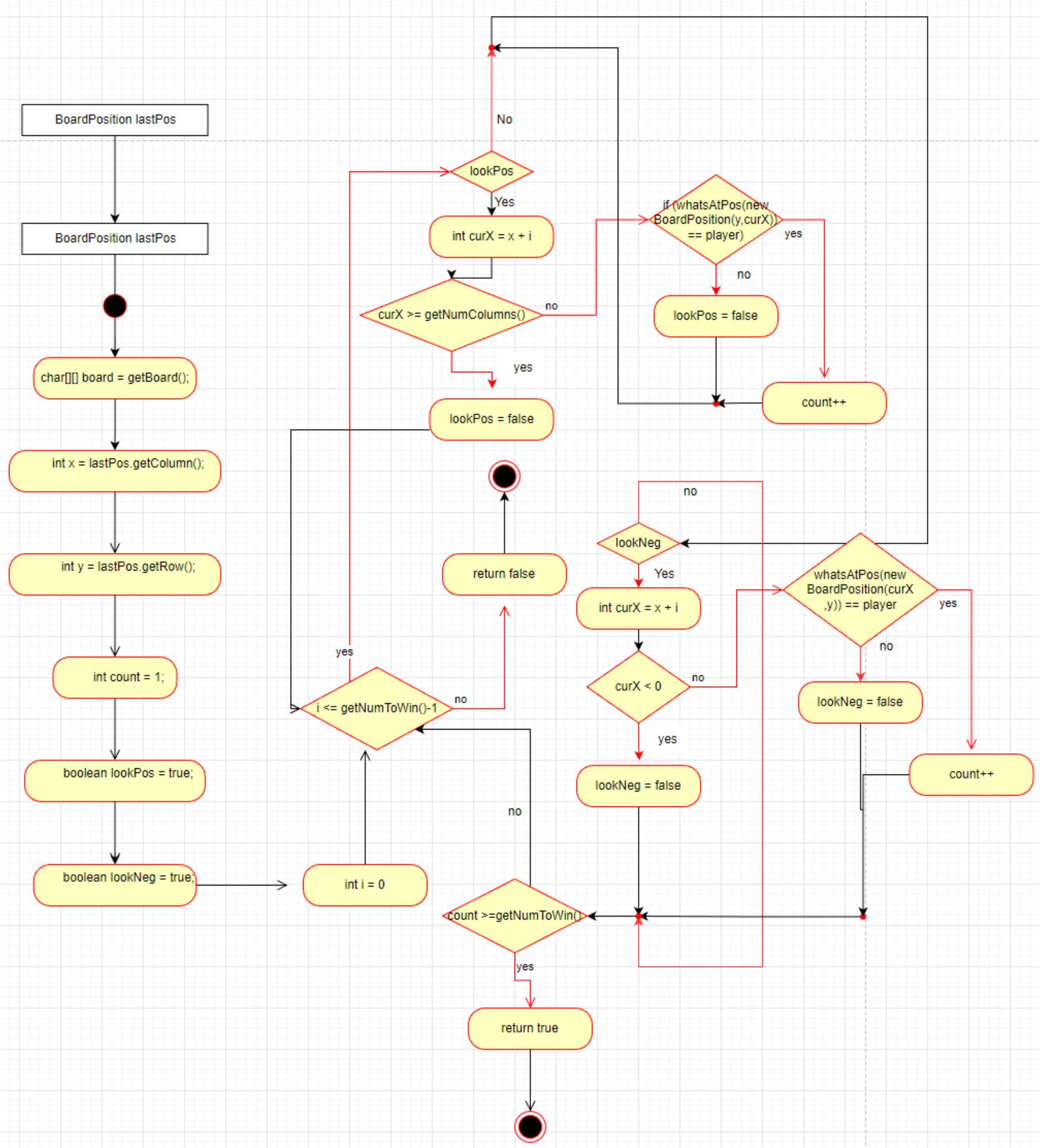
placeMarker

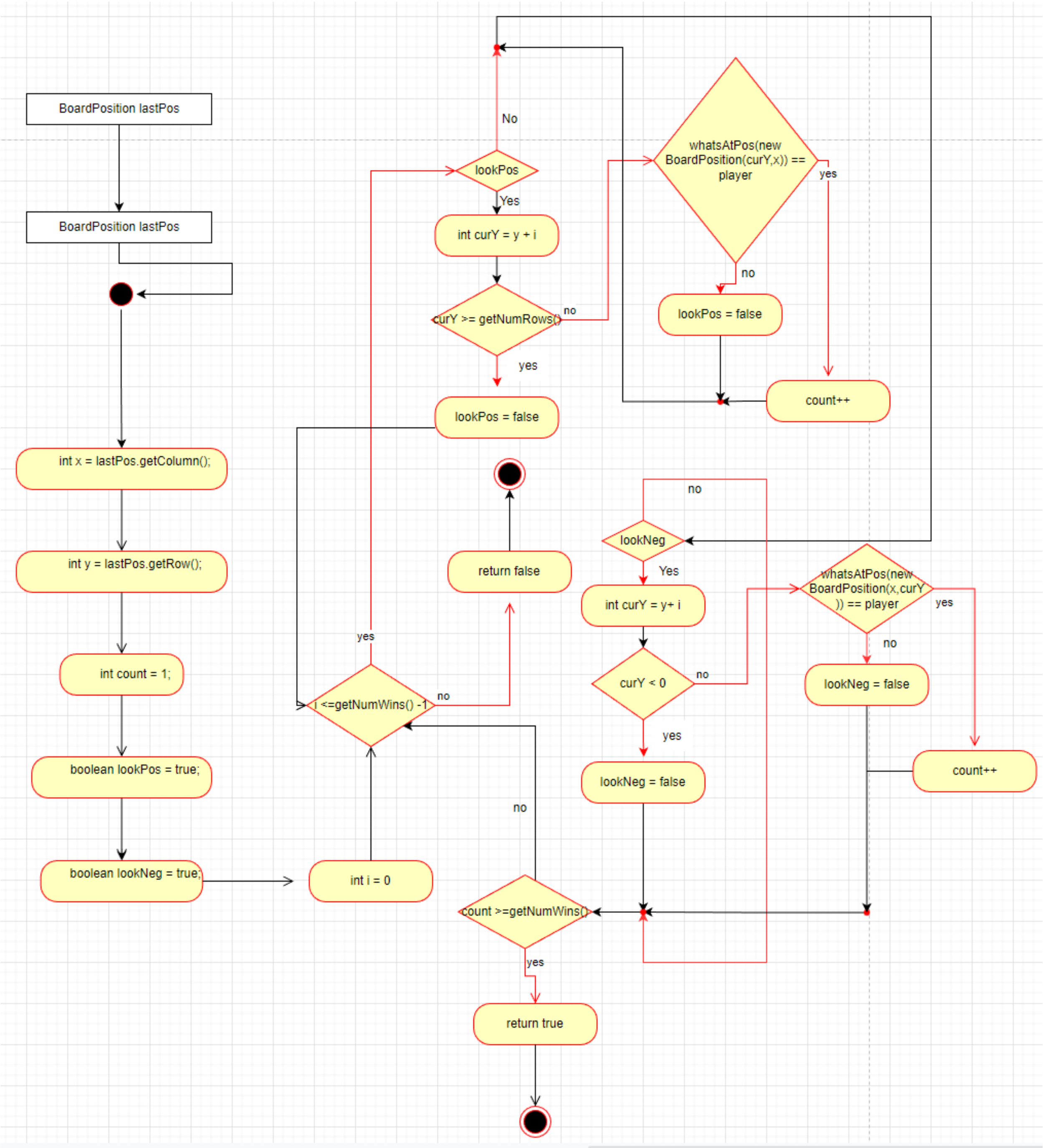

char player
BoardPosition marker
checkSpace(marker)
no
yes
board[marker.getColumn()]
[marker.getRow()] = player
turnsPlayed++

toString

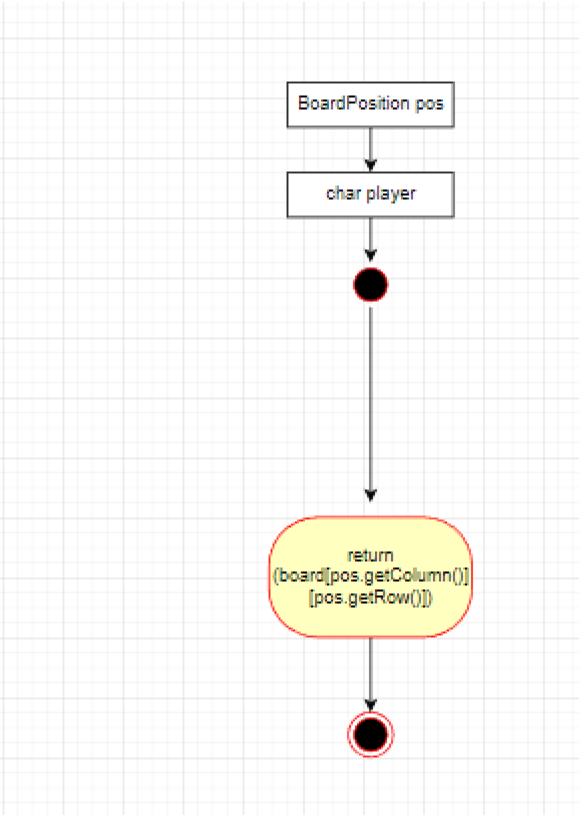

boardScheme += "\n"
boardScheme += whatsAtPos(new BoardPosition(i,j))+ "|";
yes
no
j ++
j < boardWidrth
String boardScheme = " "
int j = 0
boardWidth = getNumColumns()
boardScheme += Integer.toString(i) + "|";
boardHeight = getNumRows()
yes
int i = 0
k++
k < boardHeight
no
return boardScheme
no
i++
i < boardWidth
boardScheme += "\n"
int = k
boardScheme += Integer.toString(i) + "|";

checkHorizontalWin



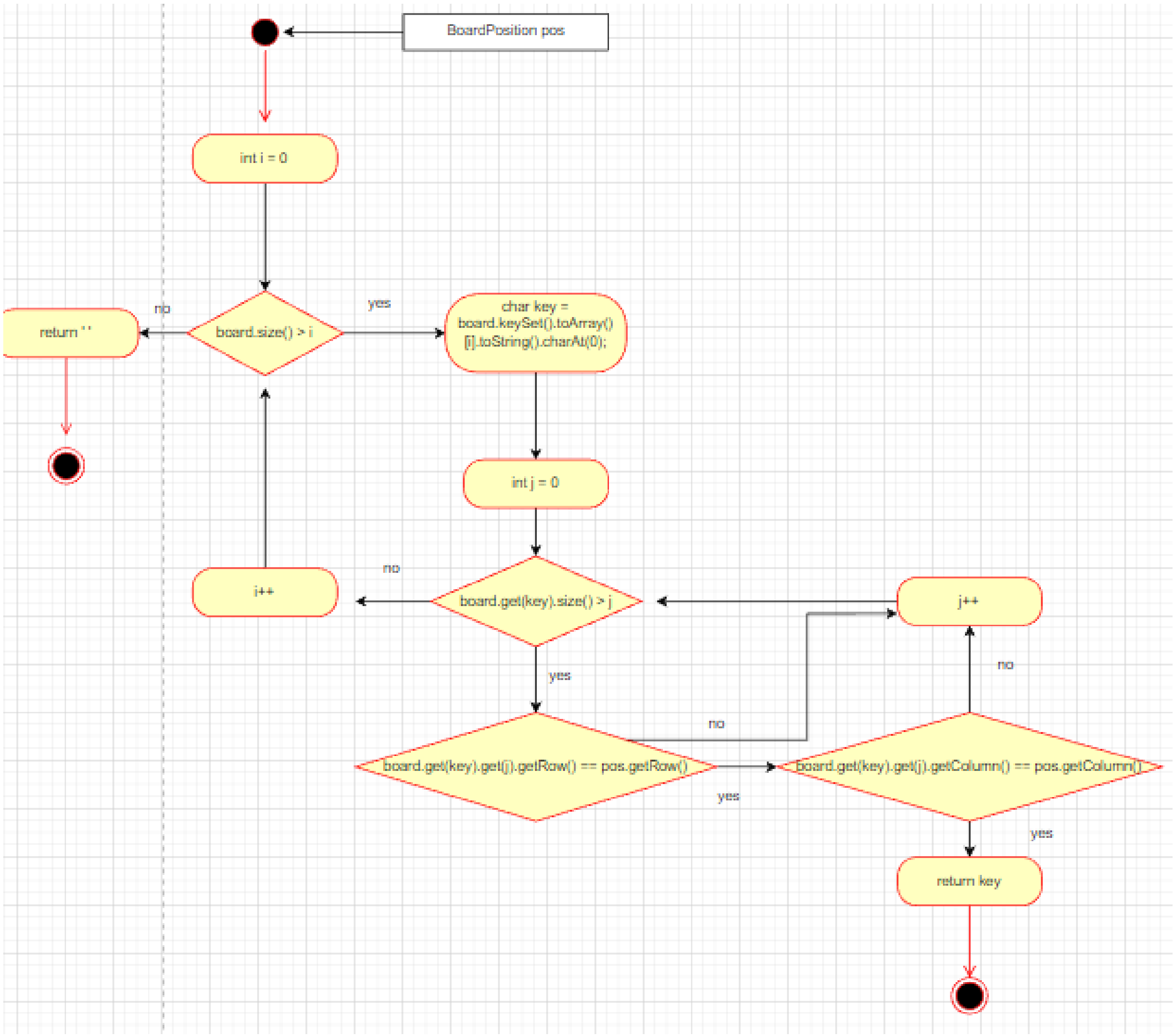


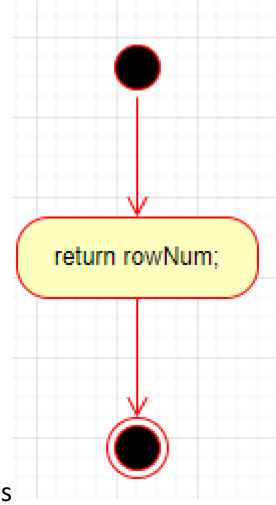
checkVerticalWin


BoardPosition pos
char player
return
(board[pos.getColumn()]
[pos.getRow()])

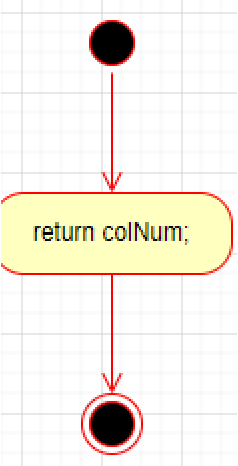
whatsAtPos

WhatsAtPos(for mem)


BoardPosition pos
int i = 0
board.size() > i
no
return ' '
yes
char key =
board.keySet().toArray()
[i].toString().charAt(0);
int j = 0
i++
no
board.get(key).size() > j
j++
yes
no
no
board.get(key).get(j).getRow() == pos.getRow()
yes
board.get(key).get(j).getColumn() == pos.getColumn()
yes
return key

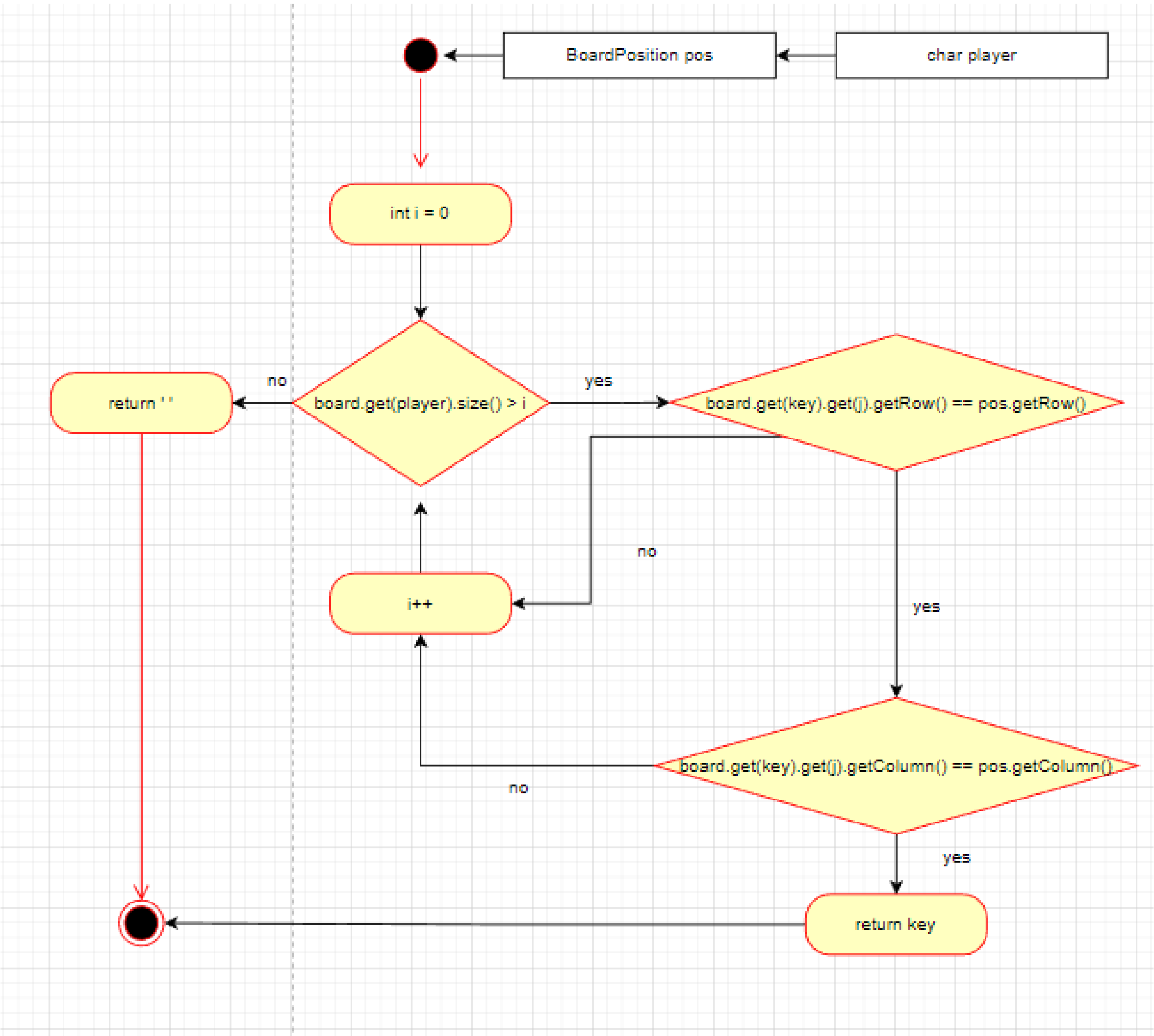


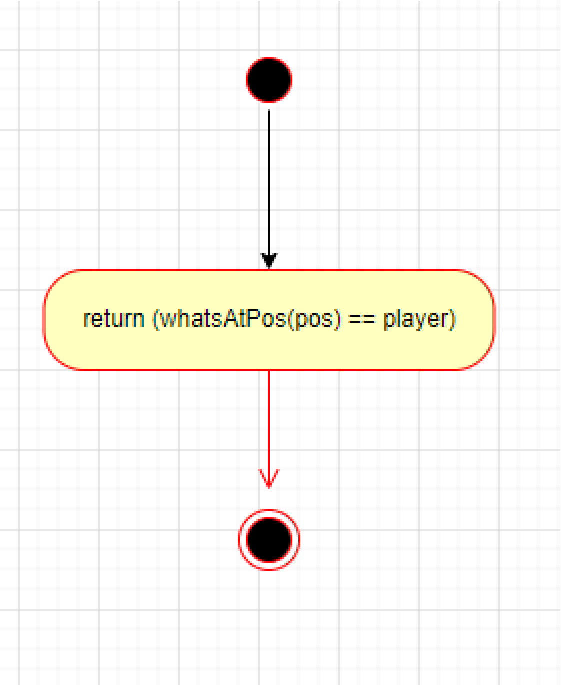
getNumRows



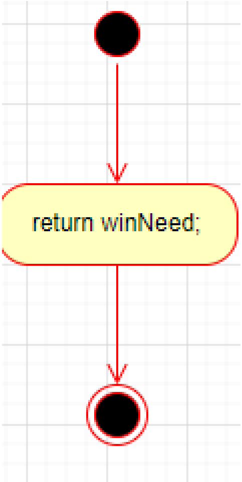
getNumColumns

isplayerAtPos (mem)


BoardPosition pos
char player
int i = 0
board.get(player).size() > i
no
yes
return ' '
board.get(key).get(j).getRow() == pos.getRow()
no
yes
i++
board.get(key).get(j).getColumn() == pos.getColumn()
no
yes
return key

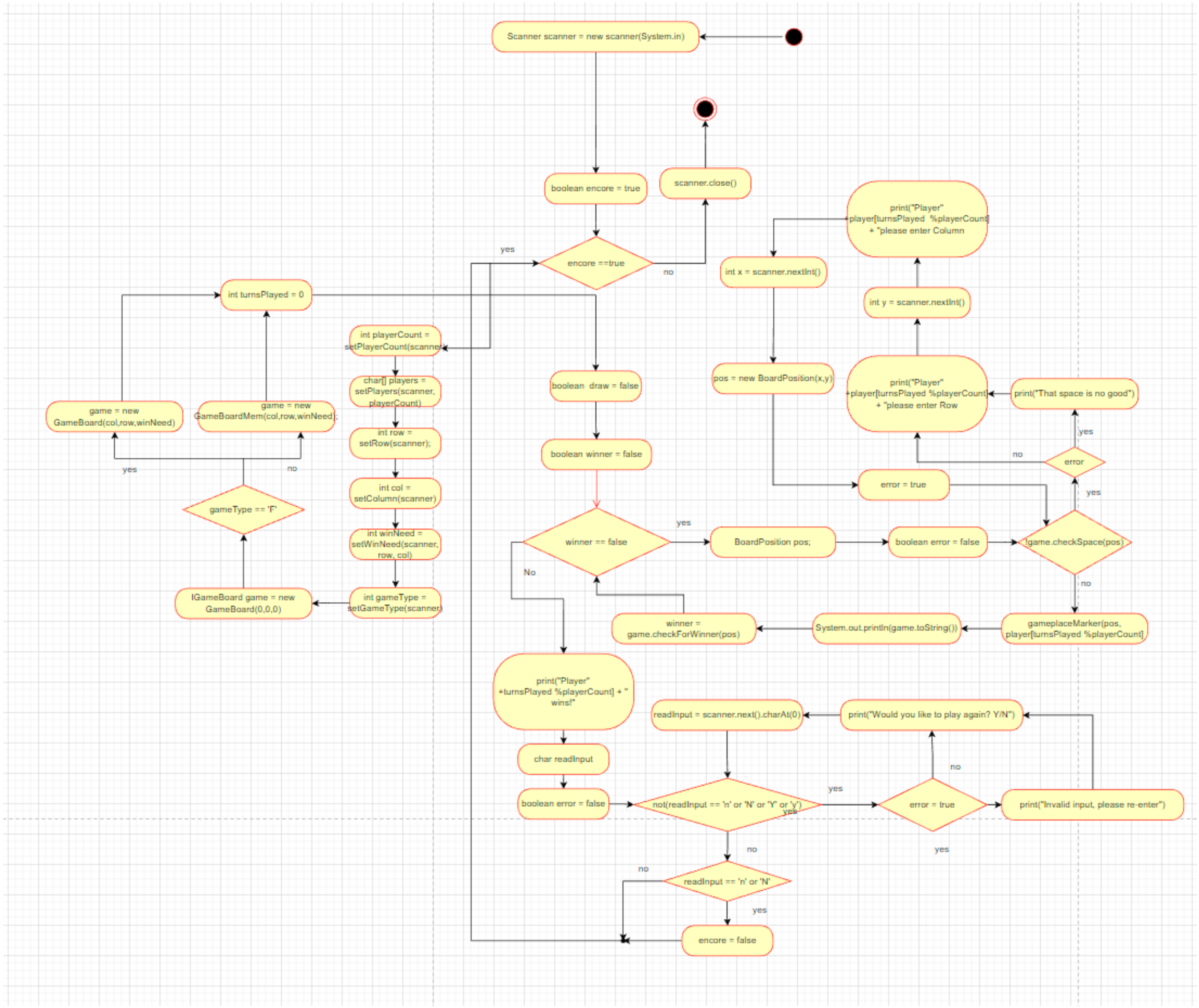


isPlayerAtPos

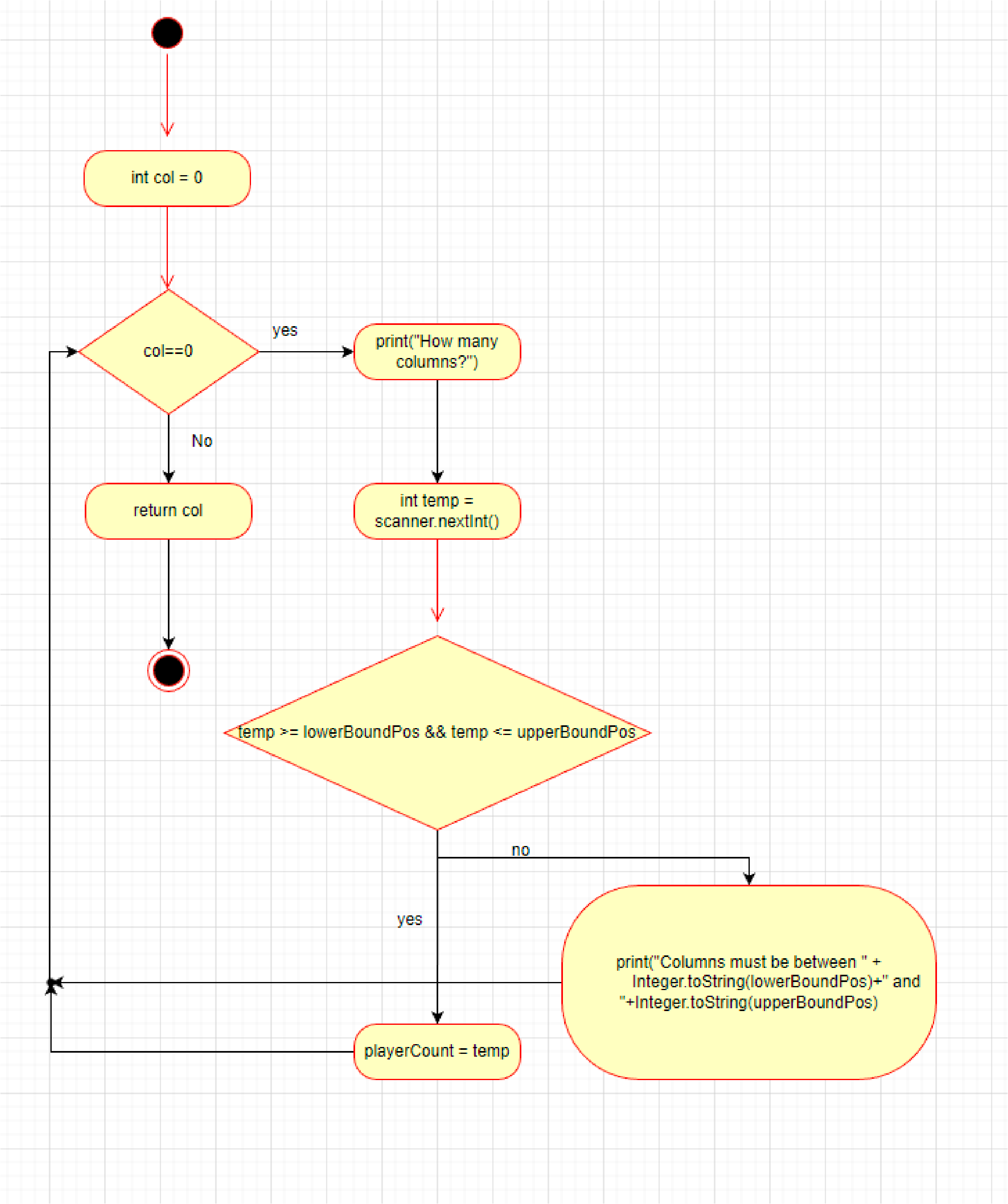


getNumToWin

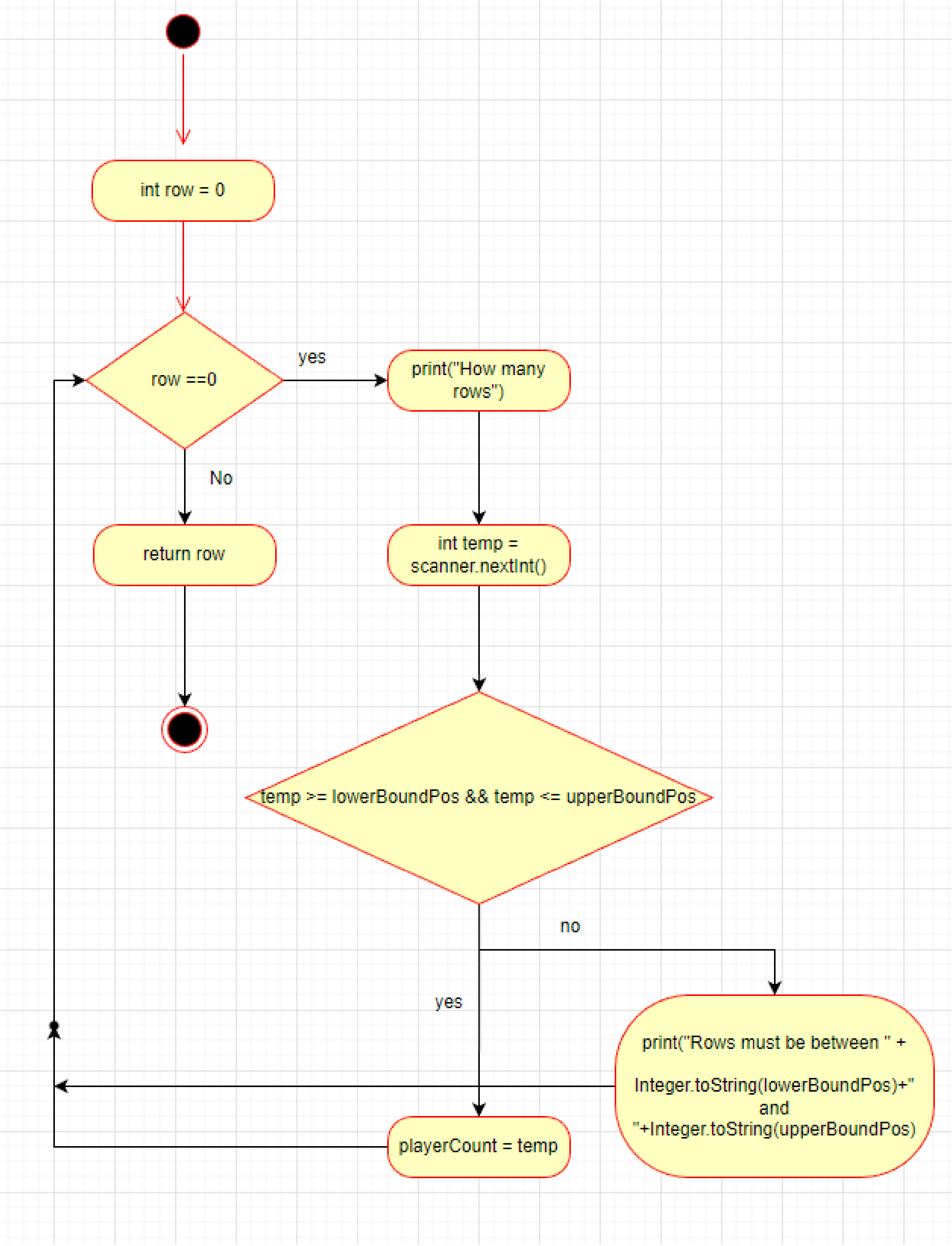
main

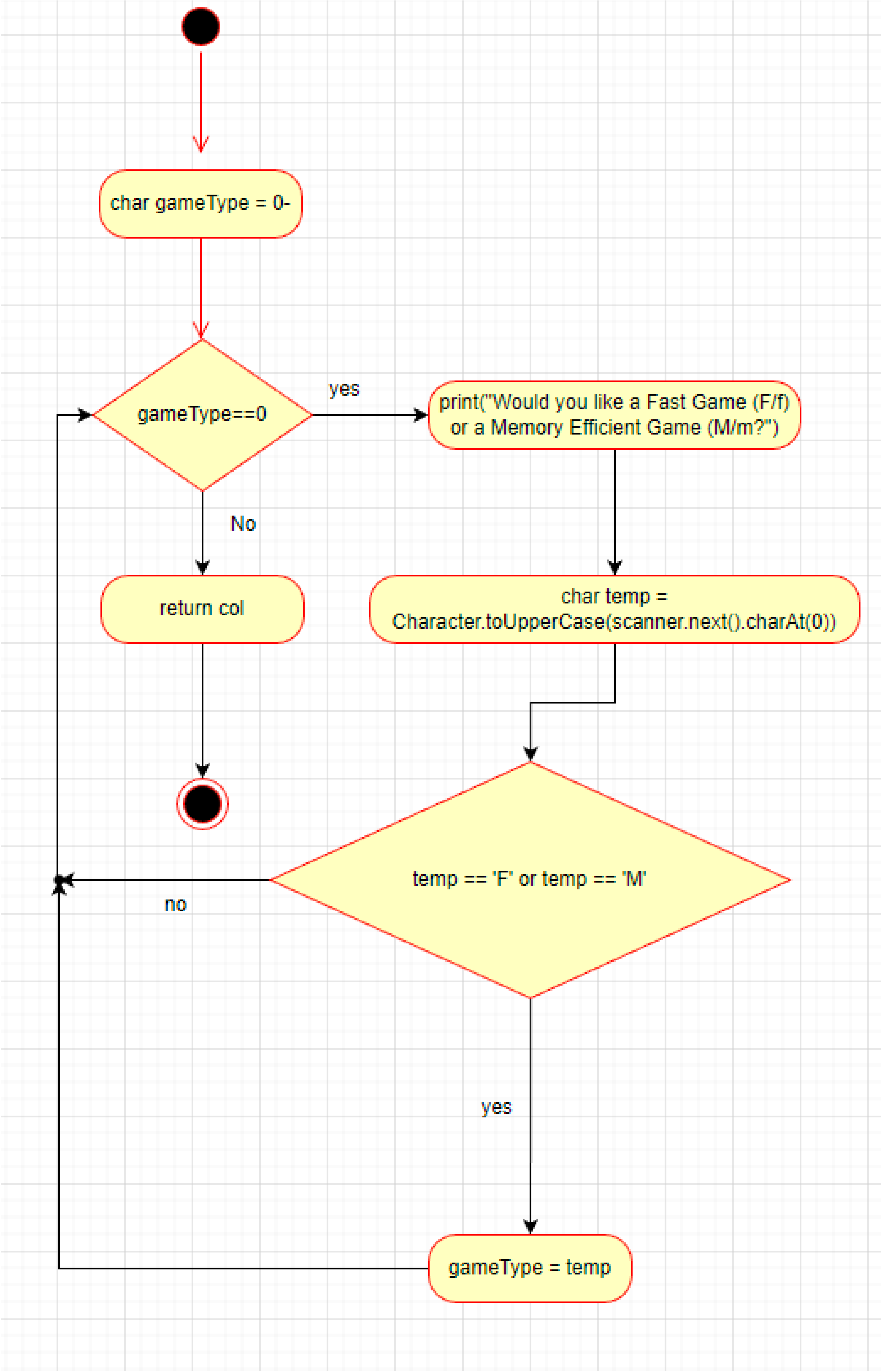

Scanner scanner = new scanner(System.in)
boolean encore = true
scanner.close()
encore ==true
yes
no
int turnsPlayed = 0
int playerCount = setPlayerCount(scanner)
char[] players = setPlayers(scanner, playerCount)
int row = setRow(scanner);
int col = setColumn(scanner)
int winNeed = setWinNeed(scanner, row, col)
int gameType = setGameType(scanner)
IGameBoard game = new GameBoard(0,0,0)
gameType == 'F'
yes
no
game = new GameBoard(col,row,winNeed)
game = new GameBoardMem(col,row,winNeed);
boolean draw = false
boolean winner = false
winner == false
yes
No
BoardPosition pos;
boolean error = false
!game.checkSpace(pos)
yes
no
error
yes
no
print("That space is no good")
print("Player" +player[turnsPlayed %playerCount] + "please enter Row
int y = scanner.nextInt()
print("Player" +player[turnsPlayed %playerCount] + "please enter Column
int x = scanner.nextInt()
pos = new BoardPosition(x,y)
error = true
gameplaceMarker(pos, player[turnsPlayed %playerCount]
System.out.println(game.toString())
winner = game.checkForWinner(pos)
print("Player" +turnsPlayed %playerCount] + " wins!"
char readInput
boolean error = false
readInput = scanner.next().charAt(0)
print("Would you like to play again? Y/N")
not(readInput == 'n' or 'N' or 'Y' or 'y')
yes
yes
error = true
no
yes
print("Invalid input, please re-enter")
no
readInput == 'n' or 'N'
no
yes
encore = false

setCol


int col = 0
col==0
yes
print("How many columns?")
No
return col
int temp = scanner.nextInt()
temp >= lowerBoundPos && temp <= upperBoundPos
no
yes
print("Columns must be between " + Integer.toString(lowerBoundPos)+" and "+Integer.toString(upperBoundPos)
playerCount = temp

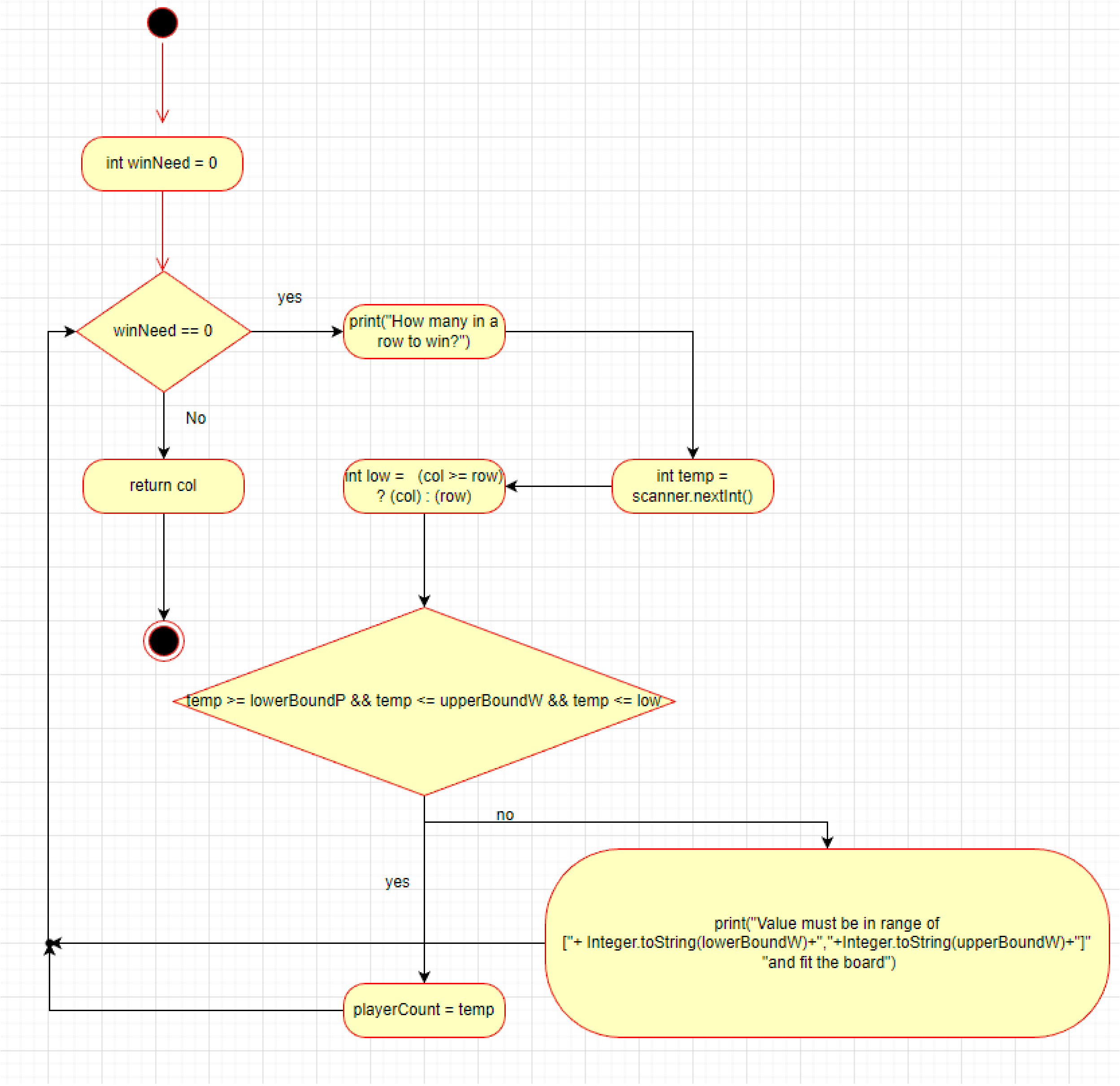
setRow


int row = 0
row ==0
yes
print("How many rows")
No
return row
int temp = scanner.nextInt()
temp >= lowerBoundPos && temp <= upperBoundPos
no
yes
print("Rows must be between " + Integer.toString(lowerBoundPos)+" and "+Integer.toString(upperBoundPos)
playerCount = temp

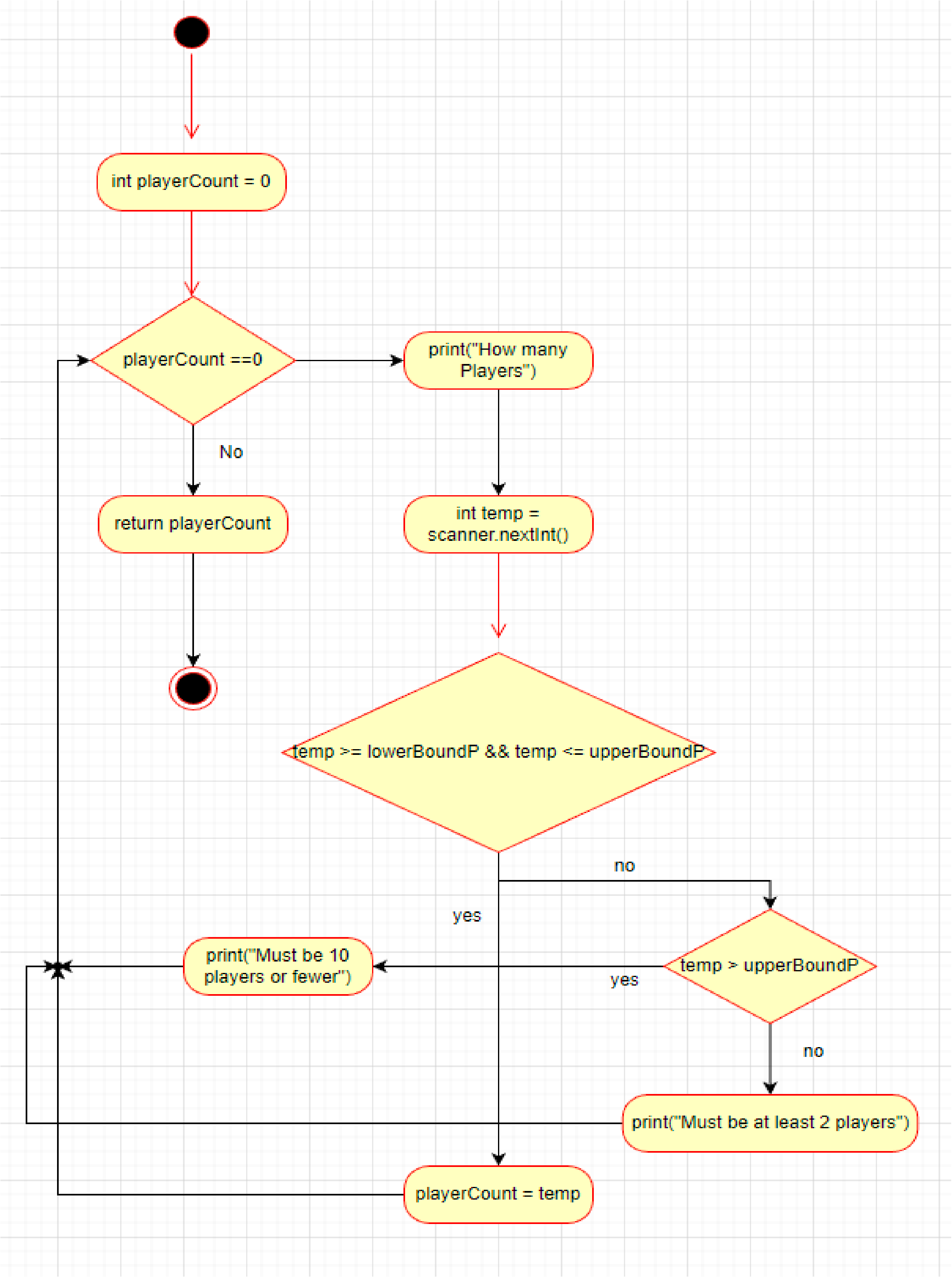

char gameType = 0-
gameType==0
yes
print("Would you like a Fast Game (F/f) or a Memory Efficient Game (M/m?")
No
return col
char temp = Character.toUpperCase(scanner.next().charAt(0))
temp == 'F' or temp == 'M'
no
yes
gameType = temp

setGameType

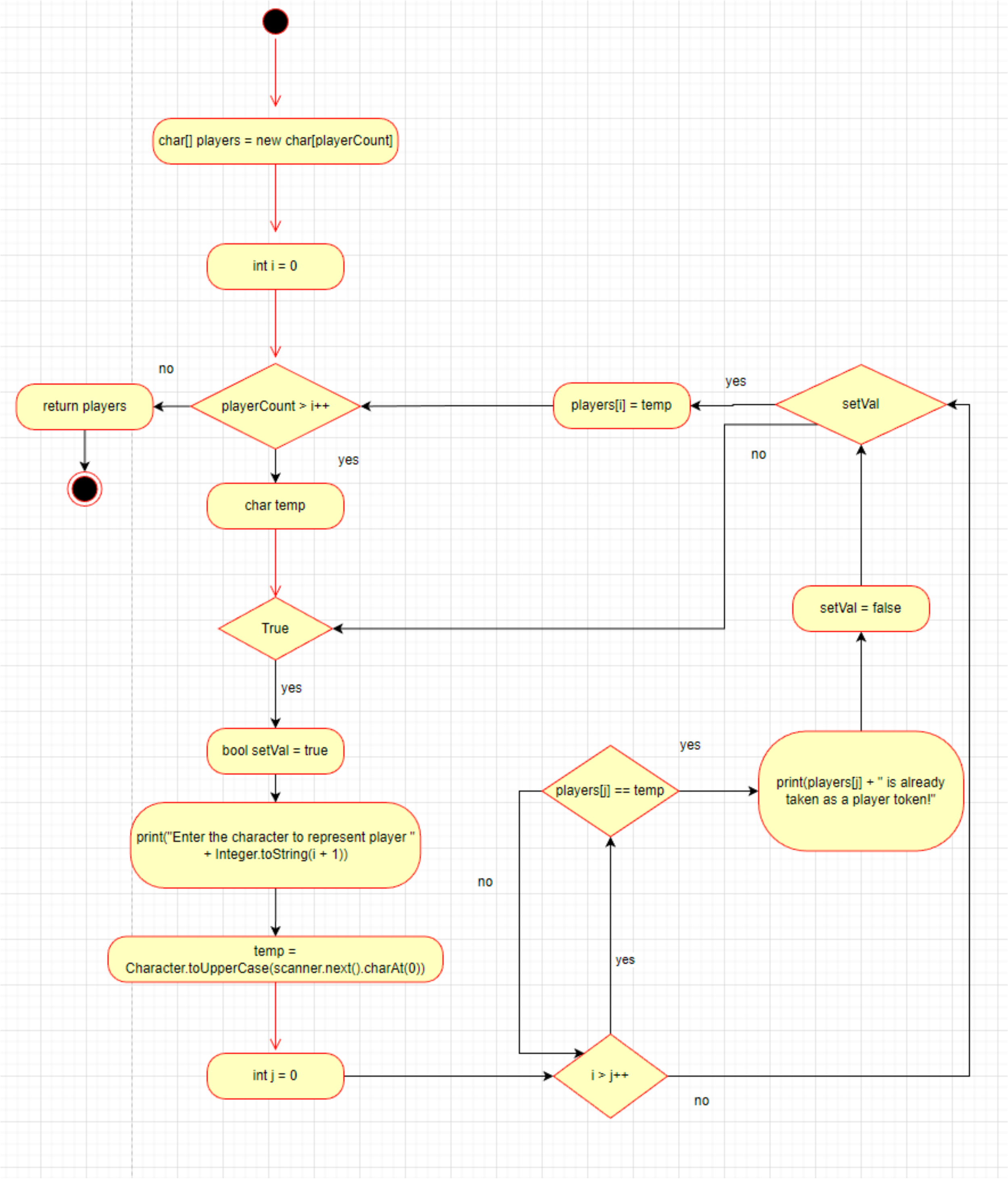
setWinNeed


int winNeed = 0
winNeed == 0
yes
print("How many in a row to win?")
No
return col
int low = (col >= row) ? (col) : (row)
int temp = scanner.nextInt()
temp >= lowerBoundP && temp <= upperBoundW && temp <= low
no
yes
print("Value must be in range of ["+ Integer.toString(lowerBoundW)+","+Integer.toString(upperBoundW)+"]" "and fit the board")
playerCount = temp

setPlayerCount

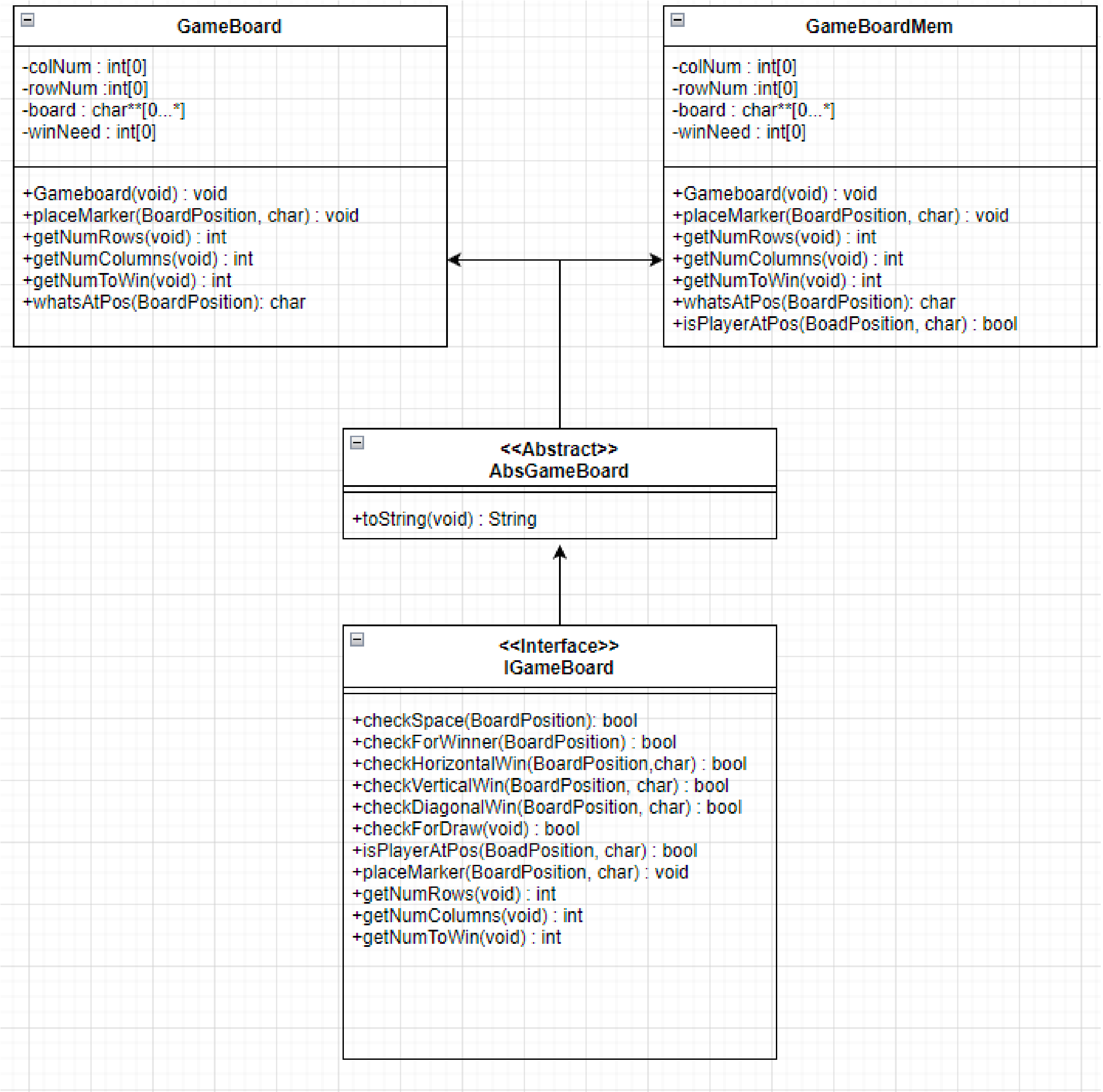

int playerCount = 0
playerCount ==0
print("How many Players")
No
return playerCount
int temp = scanner.nextInt()
temp >= lowerBoundP && temp <= upperBoundP
no
yes
print("Must be 10 players or fewer")
temp > upperBoundP
yes
no
print("Must be at least 2 players")
playerCount = temp

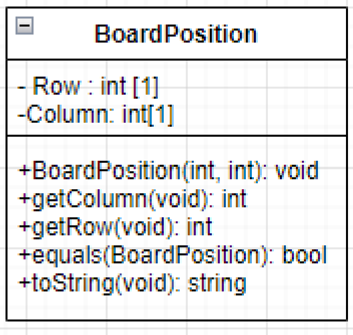
setPlayers

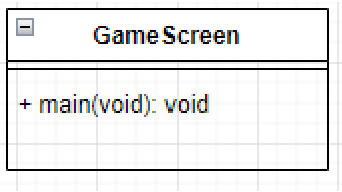


char[] players = new char[playerCount]
int i = 0
no
return players
playerCount > i++
yes
char temp
players[i] = temp
yes
setVal
no
True
setVal = false
yes
bool setVal = true
yes
players[j] == temp
print(players[j] + " is already taken as a player token!"
print("Enter the character to represent player " + Integer.toString(i + 1))
no
yes
temp = Character.toUpperCase(scanner.next().charAt(0))
int j = 0
i > j++
no

UML Class Diagrams


GameBoard
-colNum : int[0]
-rowNum :int[0]
-board : char**[0...*]
-winNeed : int[0]
+Gameboard(void) : void
+placeMarker(BoardPosition, char) : void
+getNumRows(void) : int
+getNumColumns(void) : int
+getNumToWin(void) : int
+whatsAtPos(BoardPosition): char
GameBoardMem
-colNum : int[0]
-rowNum :int[0]
-board : char**[0...*]
-winNeed : int[0]
+Gameboard(void) : void
+placeMarker(BoardPosition, char) : void
+getNumRows(void) : int
+getNumColumns(void) : int
+getNumToWin(void) : int
+whatsAtPos(BoardPosition): char
+isPlayerAtPos(BoadPosition, char) : bool
<<Abstract>>
AbsGameBoard
+toString(void) : String
<<Interface>>
IGameBoard
+checkSpace(BoardPosition): bool
+checkForWinner(BoardPosition) : bool
+checkHorizontalWin(BoardPosition,char) : bool
+checkVerticalWin(BoardPosition, char) : bool
+checkDiagonalWin(BoardPosition, char) : bool
+checkForDraw(void) : bool
+isPlayerAtPos(BoadPosition, char) : bool
+placeMarker(BoardPosition, char) : void
+getNumRows(void) : int
+getNumColumns(void) : int
+getNumToWin(void) : int





Testing

In your report, include the test case method name and description for each test case which includes your input values, your expected output, and a reason for why you chose this test case and what makes it distinct. Remember that the current state of the GameBoard is part of the input and part of the output. Follow the same format as outlined in the example test case PDF.

Constructor

Input:

numberOfCol = 3
numberOfCol = 3
numberOfCol = 3

Output:

State: (number to win = 3)

	0	1	2
0			
1			
2			

Reason:

This test case is unique and distinct because the board dimensions are the smallest possible option.

Function name:
TestConstructor_ Small_board

Input:

numberOfCol = 5
numberOfCol = 5
numberOfCol = 5

Output:

State: (number to win = 2)

	0	1	2	3
0	X			
1		X		
2				
3				

Reason:

This test case is unique and distinct because the board number to win state is set to the lowest value.

Function name:
TestConstructor_ Small_win

<table>
<tr><th>Input:</th><th>Output:</th><th>Reason:</th></tr>
<tr>
<td>numberOfCol = 100
numberOfCol = 100
numberOfCol = 20</td>
<td>State: (number to win = 20)
<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>…</th><th>100</th></tr>
<tr><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>4</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>…</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>100</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>
</td>
<td>This test case is unique and distinct because the board dimensions are the maximum value.

<b>Function name:</b>
<i>TestConstructor_large_board</i></td>
</tr>
</table>

checkSpace

<table>
<tr><th>Input:</th><th>Output:</th><th>Reason:</th></tr>
<tr>
<td>State: (number to win = 4)
<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>0</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td>x</td><td></td><td></td></tr>
<tr><td>3</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>4</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>
Pos.getRow = 2
Pos.getCol = 2</td>
<td>checkSpace = false

State of the board is unchanged</td>
<td>This test case is unique and distinct because the position 2,2 is occupied by a player and surrounded in spaces.

<b>Function name:</b>
<i>TestCheckSpace_false_surrounded_by_spaces</i></td>
</tr>
</table>

Input:

State: (number to win = 4)

	0	1	2	3	4
0					
1		x	x	x	
2		x		x	
3		x	x	x	
4					

Pos.getRow = 2
Pos.getCol = 2

Output:
checkSpace = true

State of the board is unchanged

Reason:
This test case is unique and distinct because the position 2,2 is not occupied by a player and is surrounded by players.

Function name:
TestCheckSpace_true_ surrounded_by_players

Input:

State: (number to win = 4)

	0	1	2	3	4
0					
1					
2					
3					
4					

Pos.getRow = -2
Pos.getCol = -2

Output:
checkSpace = false

State of the board is unchanged

Reason:
This test case is unique and distinct because the position (-2,-2) is outside of the board dimensions.

Function name:
TestCheckSpace_false_ outside_boundary

CheckHorizontalWin

Input:

State: (number to win = 4)

	0	1	2	3	4
0					
1					
2	x	x	x	x	X
3					
4					

Pos.getRow = 2
Pos.getCol = 2
P = ‘x’

Output:

CheckHorizontalWin = true

State of the board is unchanged

Reason:

This test case is unique and distinct because the last x was placed in the middle of the string of 4 consecuitive x’s as opposed to on the end, so the function needs to counts x’s on the right and left

Function name:

TestCheckHorizontalWin_win_last_marker_middle

Input:

State: (number to win = 4)

	0	1	2	3	4
0					
1					
2	x	x	x	o	x
3					
4					

Pos.getRow = 2
Pos.getCol = 2
P = ‘x’

Output:

CheckHorizontalWin = false

State of the board is unchanged

Reason:

This test case is unique and distinct because a o seperates the row of x’s but the row does have a number of x’s equal to the win requirement.

Function name:

TestCheckHorizontalWin_win_seperated_row

<table>
<tr>
<td><b>Input:</b>
State: (number to win = 3)
<table>
<tr><td></td><td>0</td><td>1</td><td>2</td></tr>
<tr><td>0</td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>x</td><td>x</td><td>x</td></tr>
</table>
Pos.getRow = 2
Pos.getCol = 0
P = ‘x’</td>
<td><b>Output:</b>
CheckHorizontalWin = true
State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because the last x was placed at the border and is the same size as the boards width. This is an edge case.
<b>Function name:</b>
<i>TestCheckHorizontalWin_win_size_of_board_</i></td>
</tr>
</table>

<table>
<tr>
<td><b>Input:</b>
State: (number to win = 25)
<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>…</td><td>24</td></tr>
<tr><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>…</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>24</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>
Pos.getRow = 2
Pos.getCol = 2
P = ‘x’</td>
<td><b>Output:</b>
CheckHorizontalWin = true
State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because a the board win number is set to boundry value and is equal to the size of the boards dimensions. This is an edge case.
<b>Function name:</b>
<i>TestCheckHorizontalWin_win_max_row</i></td>
</tr>
</table>

CheckVerticalWin

Input:

State: (number to win = 4)

	0	1	2	3	4
0			x		
1			x		
2	x	x	x	o	o
3		o	x	o	
4					

Pos.getRow = 2
Pos.getCol = 2
P = 'x'

Output:
CheckVerticalWin = true

State of the board is unchanged

Reason:
This test case is unique and distinct because the last x was placed in the middle of the string of 4 consecuitive x's as opposed to on the end, so the function needs to counts x's on the top and bottom

Function name:
TestCheckVerticalWin_win_last_marker_middle

Input:

State: (number to win = 4)

	0	1	2	3	4
0		x			
1		x			
2	x	x	x	o	x
3		o			
4		x			

Pos.getRow = 2
Pos.getCol = 1
P = 'x'

Output:
CheckVerticalWin = false

State of the board is unchanged

Reason:
This test case is unique and distinct because a o seperates the row of x's but the row does have a number of x's equal to the win requirement.

Function name:
TestCheckVerticalWin_no_win_seperated_row

Input:

State: (number to win = 3)

	0	1	2
0		x	
1		x	
2		x	

Pos.getRow = 2
Pos.getCol = 1
P = ‘x’

Output:
CheckVerticalWin = true

State of the board is unchanged

Reason:
This test case is unique and distinct because the last x was placed at the border and is the same size as the boards width. This is an edge case.

Function name:
*TestCheckVerticalWin_win_size_of_board_*

Input:

State: (number to win = 25)

	0	1	2	3	4	…	24
0				x			
1				x			
2				x			
3				x			
…				x			
24				x			

Pos.getRow = 2
Pos.getCol = 2
P = ‘x’

Output:
CheckVerticalWin = true

State of the board is unchanged

Reason:
This test case is unique and distinct because the board win number equals the size of the boards dimensions. This is an edge case.

Function name:
TestCheckVerticalWin_win_max_row

CheckDiagonalWin

Input:

State: (number to win = 4)

	0	1	2	3	4
0	x				
1		x			
2			x		
3				x	
4					

Pos.getRow = 0
Pos.getCol = 0
P = ‘x’

Output:

CheckDiagonalWin = true

State of the board is unchanged

Reason:

This test case is unique and distinct because there exists a row of x’s with size equaling the number to win. Further, the win condition is checked from NW to SE direction.

Function name:
TestCheckDiagonallWin_win_NW->SE

Input:

State: (number to win = 4)

	0	1	2	3	4
0	x				
1		x			
2			x		
3				x	
4					

Pos.getRow = 3
Pos.getCol = 3
P = ‘x’

Output:

CheckDiagonalWin = true

State of the board is unchanged

Reason:

This test case is unique and distinct because there exists a row of x’s with size equaling the number to win. Further, the win condition is checked from SE to NW direction.

Function name:
TestCheckDiagonallWin_win_SE->NW

<table>
<tr>
<td>

Input:

State: (number to win = 4)

<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>0</td><td>x</td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td>x</td><td></td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td>o</td><td></td><td></td></tr>
<tr><td>3</td><td></td><td></td><td></td><td>x</td><td></td></tr>
<tr><td>4</td><td></td><td></td><td></td><td></td><td>x</td></tr>
</table>

Pos.getRow = 4
Pos.getCol = 4
P = ‘x’

</td>
<td>

Output:

CheckDiagonalWin = false

State of the board is unchanged

</td>
<td>

Reason:

This test case is unique and distinct because there exists a row of x’s with size equaling the number to win but is interupted by a o. Further, the win condition is checked from SE to NW direction.

Function name:

TestCheckDiagonallWin _no_win_SE->NW_interupted

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

<table>
<tr>
<td>

Input:

State: (number to win = 4)

<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>0</td><td>o</td><td>x</td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td>x</td><td></td><td>x</td><td></td></tr>
<tr><td>2</td><td>x</td><td>x</td><td>x</td><td>o</td><td>x</td></tr>
<tr><td>3</td><td></td><td>x</td><td></td><td>x</td><td></td></tr>
<tr><td>4</td><td>x</td><td>x</td><td></td><td></td><td></td></tr>
</table>

Pos.getRow = 4
Pos.getCol = 0
P = ‘x’

</td>
<td>

Output:

CheckDiagonalWin = true

State of the board is unchanged

</td>
<td>

Reason:

This test case is unique and distinct because there exists a row of x’s with size equaling the number to win. Further, the win condition is checked from SW to NE direction.

Function name:

TestCheckDiagonallWin _win_NE->SW

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

<table>
<tr>
<td><b>Input:</b>

State: (number to win = 4)

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>0</td><td>o</td><td>x</td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td>x</td><td></td><td>x</td><td></td></tr>
<tr><td>2</td><td>x</td><td>x</td><td>x</td><td>o</td><td>x</td></tr>
<tr><td>3</td><td></td><td>x</td><td></td><td>x</td><td></td></tr>
<tr><td>4</td><td>x</td><td>x</td><td></td><td></td><td></td></tr>
</table>

Pos.getRow = 1
Pos.getCol = 3
P = ‘x’</td>
<td><b>Output:</b>
CheckDiagonalWin = true

State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because there exists a row of x’s with size equaling the number to win. Further, the win condition is checked from NE to SW direction.

<b>Function name:</b>
<i>TestCheckDiagonallWin _win_NE-&gt;SW</i></td>
</tr>
</table>

<table>
<tr>
<td><b>Input:</b>

State: (number to win = 25)

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>…</th><th>24</th></tr>
<tr><td>0</td><td>x</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>1</td><td></td><td>x</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td></td><td></td><td>x</td><td></td><td></td><td></td><td></td></tr>
<tr><td>3</td><td></td><td></td><td></td><td>x</td><td></td><td></td><td></td></tr>
<tr><td>4</td><td></td><td></td><td></td><td></td><td>x</td><td></td><td></td></tr>
<tr><td>…</td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td></tr>
<tr><td>24</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td></tr>
</table>

Pos.getRow = 3
Pos.getCol = 3
P = ‘x’</td>
<td><b>Output:</b>
CheckDiagonalWin = true

State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because there exists a row of x’s with size equaling the size of the boards dimensions and also equals the row max value. This is an edge case.

<b>Function name:</b>
<i>TestCheckDiagonallWin _win_MAX</i></td>
</tr>
</table>

<table>
<tr>
<td><b>Input:</b>
State: (number to win = 2)
<table>
<tr><td></td><td>0</td><td>1</td><td>2</td></tr>
<tr><td>0</td><td>x</td><td></td><td>o</td></tr>
<tr><td>1</td><td></td><td>x</td><td></td></tr>
<tr><td>2</td><td>o</td><td></td><td>o</td></tr>
</table>
Pos.getRow = 1
Pos.getCol = 1
P = ‘x’</td>
<td><b>Output:</b>
CheckDiagonalWin = true
State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because there exists a row of x’s with size equaling the minimum row size for a win. This is an edge case.
<b>Function name:</b>
<i>TestCheckDiagonallWin _win_MIN</i></td>
</tr>
</table>

checkForDraw

<table>
<tr>
<td><b>Input:</b>
State: (number to win = 5)
<table>
<tr><td></td><td>0</td><td>1</td><td>2</td></tr>
<tr><td>0</td><td>o</td><td>x</td><td>O</td></tr>
<tr><td>1</td><td>O</td><td>x</td><td>O</td></tr>
<tr><td>2</td><td>x</td><td>x</td><td>x</td></tr>
</table>
Pos.getRow = 1
Pos.getCol = 1
P = ‘x’</td>
<td><b>Output:</b>
checkForDraw = false
State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because the board is set to the smallest dimmensions possible.
<b>Function name:</b>
<i>TestCheckForDraw_Min_full_board</i></td>
</tr>
</table>

<table>
<tr>
<td><b>Input:</b>

State: (number to win = 5)

<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>0</td><td>o</td><td>x</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>1</td><td>O</td><td>x</td><td>x</td><td>x</td><td>O</td></tr>
<tr><td>2</td><td>x</td><td>o</td><td>x</td><td>o</td><td>x</td></tr>
<tr><td>3</td><td>O</td><td>x</td><td>O</td><td>x</td><td>o</td></tr>
<tr><td>4</td><td>x</td><td>x</td><td>o</td><td>o</td><td>o</td></tr>
</table>

Pos.getRow = 1
Pos.getCol = 1
P = ‘x’</td>
<td><b>Output:</b>
checkForDraw = true

State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because the board is entirly filled and no win condition exists.

<b>Function name:</b>
<i>TestCheckForDraw_full_board_</i></td>
</tr>
</table>

<table>
<tr>
<td><b>Input:</b>

State: (number to win = 5)

<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>0</td><td>o</td><td>x</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>1</td><td>O</td><td></td><td>x</td><td>x</td><td>O</td></tr>
<tr><td>2</td><td>x</td><td>o</td><td>x</td><td>o</td><td>x</td></tr>
<tr><td>3</td><td>O</td><td>x</td><td>O</td><td>x</td><td>o</td></tr>
<tr><td>4</td><td>x</td><td>x</td><td>o</td><td>o</td><td>o</td></tr>
</table>

Pos.getRow = 2
Pos.getCol = 1
P = ‘x’</td>
<td><b>Output:</b>
checkForDraw = false

State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because the board is one move away from being filled.

<b>Function name:</b>
<i>TestCheckForDraw_almost_full_board_</i></td>
</tr>
</table>

Input:

State: (number to win = 5)
Note: $ = [unique non-repeating character value]

	0	1	2	3	4	…	100
0	$	$	$	$	$	$	$
1	$	$	$	$	$	$	$
2	$	$	$	$	$	$	$
3	$	$	$	$	$	$	$
4	$	$	$	$	$	$	$
…	$	$	$	$	$	$	$
100	$	$	$	$	$	$	$

Pos.getRow = 1
Pos.getCol = 1
P = ‘x

Output:

checkForDraw = true

State of the board is unchanged

Reason:

This test case is unique and distinct because the board is filled for the largest possible board.

Function name:
TestCheckForDraw_full_board_Max

whatsAtPos

Input:

State: (number to win = 5)

	0	1	2	3	4
0	o	x	O	O	O
1	O		x	x	O
2	x	o	x	o	x
3	O	x	O	x	o
4	x	x	o	o	o

Pos.getRow = 1
Pos.getCol = 1

Output:

WhatsAtPos = ‘ ‘

State of the board is unchanged

Reason:

This test case is unique and distinct because the board at the given position is a space.

Function name:
TestWhatsAtPos_space_ element

Input:

State: (number to win = 5)

	0	1	2	3	4
0	o	x	O	O	O
1	O		x	5	O
2	x	o			x
3			O	x	o
4	x	x	o	o	

Pos.getRow = 1
Pos.getCol = 3

Output:
WhatsAtPos = ‘5‘

State of the board is unchanged

Reason:
This test case is unique and distinct because the board at the given position is a numeric value.

Function name:
TestWhatsAtPos _numeric_element

Input:

State: (number to win = 5)

	0	1	2	3	4
0	o	x	O	O	O
1	O		x	5	O
2	\n	o			x
3			O	x	o
4	x	x	o	o	

Pos.getRow = 2
Pos.getCol = 0

Output:
WhatsAtPos = ‘\n‘

State of the board is unchanged

Reason:
This test case is unique and distinct because the board at the given position is an endline character.

Function name:
TestWhatsAtPos
_ endline _*element*

Input:

State: (number to win = 5)

	0	1	2	3	4
0	o	x	O	O	O
1	O		x	5	O
2	\n	o			x
3			O	x	o
4	x	x	o	o	

Pos.getRow = 0
Pos.getCol = 0

Output:

WhatsAtPos = ‘o‘

State of the board is unchanged

Reason:

This test case is unique and distinct because the given position is at corner and thus it is an edge case.

Function name:

TestWhatsAtPos
_ min _*edge*

Input:

State: (number to win = 5)

	0	1	2	3	4
0	o	x	O	O	O
1	O		x	5	O
2	\n	o			x
3			O	x	o
4	x	x	o	o	P

Pos.getRow = 0
Pos.getCol = 0

Output:

WhatsAtPos = ‘P‘

State of the board is unchanged

Reason:

This test case is unique and distinct because the given position is at max corner and thus it is an edge case.

Function name:

TestWhatsAtPos
_ max _*edge*

isPlayerAtPos

<table>
<tr>
<td><b>Input:</b>

State: (number to win = 5)

<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>0</td><td>o</td><td>x</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>1</td><td>O</td><td></td><td>x</td><td></td><td>O</td></tr>
<tr><td>2</td><td>\n</td><td>o</td><td></td><td></td><td>x</td></tr>
<tr><td>3</td><td></td><td></td><td>O</td><td>x</td><td>o</td></tr>
<tr><td>4</td><td>x</td><td>x</td><td>o</td><td>o</td><td>P</td></tr>
</table>

Pos.getRow = 0
Pos.getCol = 0
P = ‘o’</td>
<td><b>Output:</b>
isPlayerAtPos = true

State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because the given position is at min corner and thus it is an edge case.

<b>Function name:</b>
<i>TestisPlayerAtPos</i>
_ min _<i>edge</i></td>
</tr>
</table>

<table>
<tr>
<td><b>Input:</b>

State: (number to win = 5)

<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>0</td><td>o</td><td>x</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td>1</td><td>O</td><td></td><td>x</td><td></td><td>O</td></tr>
<tr><td>2</td><td>\n</td><td>o</td><td></td><td></td><td>x</td></tr>
<tr><td>3</td><td></td><td></td><td>O</td><td>x</td><td>o</td></tr>
<tr><td>4</td><td>x</td><td>x</td><td>o</td><td>o</td><td>P</td></tr>
</table>

Pos.getRow = 0
Pos.getCol = 0
P = ‘P’</td>
<td><b>Output:</b>
isPlayerAtPos = true
State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because the given position is at max corner and thus it is an edge case.

<b>Function name:</b>
<i>TestisPlayerAtPos</i>
_ max _<i>edge</i></td>
</tr>
</table>

Input:

State: (number to win = 5)

	0	1	2	3	4
0	o	x	O	O	O
1	O		x		O
2		o			x
3			O	x	o
4	x	x	o	o	

Pos.getRow = 1
Pos.getCol = 3
P = ‘x’

Output:
isPlayerAtPos = false

State of the board is unchanged

Reason:
This test case is unique and distinct because the board at the given position is a space, not the given character.

Function name:
TestisPlayerAtPos
_ is_space

Input:

State: (number to win = 5)

	0	1	2	3	4
0	o	x	O	O	O
1	O		x		O
2		o			x
3			O	x	o
4	x	x	o	o	

Pos.getRow = 1
Pos.getCol = 4
P = ‘x’

Output:
isPlayerAtPos = false

State of the board is unchanged

Reason:
This test case is unique and distinct because the board at the given position is not the given character but is another player’s token.

Function name:
TestisPlayerAtPos
_ diff _*token*

<table>
<tr>
<td><b>Input:</b>

State: (number to win = 5)
<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>0</td><td>o</td><td>x</td><td>O</td><td>O</td><td></td></tr>
<tr><td>1</td><td>O</td><td></td><td>x</td><td>5</td><td>O</td></tr>
<tr><td>2</td><td></td><td>o</td><td></td><td></td><td>x</td></tr>
<tr><td>3</td><td></td><td></td><td>O</td><td>x</td><td>o</td></tr>
<tr><td>4</td><td>x</td><td>x</td><td>o</td><td>o</td><td></td></tr>
</table>

Pos.getRow = 4
Pos.getCol = 0
P = ‘x’</td>
<td><b>Output:</b>
isPlayerAtPos = true

State of the board is unchanged</td>
<td><b>Reason:</b>
This test case is unique and distinct because the board at the given position is at the corner of the board. This is an edge case.

<b>Function name:</b>
<i>TestisPlayerAtPos</i>
_ lowerLeft _<i>edge</i></td>
</tr>
</table>

PlaceMarker

<table>
<tr>
<td><b>Input:</b>

State: (number to win = 5)
<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>0</td><td></td><td>x</td><td>O</td><td>O</td><td></td></tr>
<tr><td>1</td><td>O</td><td></td><td>x</td><td>5</td><td>O</td></tr>
<tr><td>2</td><td></td><td>o</td><td></td><td></td><td>x</td></tr>
<tr><td>3</td><td></td><td></td><td>O</td><td>x</td><td>o</td></tr>
<tr><td>4</td><td></td><td>x</td><td>o</td><td>o</td><td></td></tr>
</table>

Pos.getRow = 0
Pos.getCol = 0
P = ‘x’</td>
<td><b>Output:</b>
State of the board is unchanged
Board = #board + [x @ at Pos]
<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>0</td><td>x</td><td>x</td><td>O</td><td>O</td><td></td></tr>
<tr><td>1</td><td>O</td><td></td><td>x</td><td>5</td><td>O</td></tr>
<tr><td>2</td><td></td><td>o</td><td></td><td></td><td>x</td></tr>
<tr><td>3</td><td></td><td></td><td>O</td><td>x</td><td>o</td></tr>
<tr><td>4</td><td></td><td>x</td><td>o</td><td>o</td><td></td></tr>
</table></td>
<td><b>Reason:</b>
This test case is unique and distinct because the board at the given position is at a corner of the board. This is an edge case.

<b>Function name:</b>
<i>TestisPlaceMarker</i>
_ topLeft _<i>edge</i></td>
</tr>
</table>

Input:

State: (number to win = 5)

	0	1	2	3	4
0	x	x	O	O	
1	O		x	5	O
2		o			x
3			O	x	o
4		x	o	o	

Pos.getRow = 0
Pos.getCol = 0
P = ‘x’

Output:

State of the board is unchanged
Board = #board + [x @ at Pos]

	0	1	2	3	4
0	x	x	O	O	
1	O		x	5	O
2		o			x
3			O	x	o
4	x	x	o	o	

Reason:

This test case is unique and distinct because the board at the given position is at a corner of the board. This is an edge case.

Function name:
TestisPlaceMarker
_ bottomLeft _*edge*

Input:

State: (number to win = 5)

	0	1	2	3	4
0	x	x	O	O	
1	O		x	5	O
2		o			x
3			O	x	o
4	x	x	o	o	

Pos.getRow = 4
Pos.getCol = 4
P = ‘x’

Output:

State of the board is unchanged
Board = #board + [x @ at Pos]

	0	1	2	3	4
0	x	x	O	O	
1	O		x	5	O
2		o			x
3			O	x	o
4	x	x	o	o	x

Reason:

This test case is unique and distinct because the board at the given position is at a corner of the board. This is an edge case.

Function name:
TestisPlaceMarker
_ bottomRight _*edge*

Input:

State: (number to win = 5)

	0	1	2	3	4
0	x	x	O	O	
1	O		x	5	O
2		o			x
3			O	x	o
4	x	x	o	o	

Pos.getRow = 0
Pos.getCol = 4
P = ‘x’

Output:
State of the board is unchanged
Board = #board + [x @ at Pos]

	0	1	2	3	4
0	x	x	O	O	x
1	O		x	5	O
2		o			x
3			O	x	o
4	x	x	o	o	x

Reason:
This test case is unique and distinct because the board at the given position is at a corner of the board. This is an edge case.

Function name:
TestisPlaceMarker
_ topRight _*edge*

Input:

State: (number to win = 5)

	0	1	2	3	4
0	x	x	O		x
1	O	o	o	o	O
2		o		o	
3		o	O	o	o
4	x	x	o	o	x

Pos.getRow = 2
Pos.getCol = 2
P = ‘x’

Output:
State of the board is unchanged
Board = #board + [x @ at Pos]

	0	1	2	3	4
0	x	x	O		x
1	O	o	o	o	O
2		o	x	o	
3		o	O	o	o
4	x	x	o	o	x

Reason:
This test case is in the middle of the board and surrounded by different elements.

Function name:
TestisPlaceMarker
_ surrounded _*mid*