Kenny Nguyen

CPSC 2150

Project 4

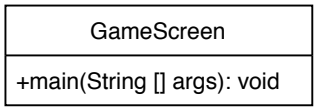
Functional Requirements:

- As a player, I can choose what column I want to place my marker in.
- As a player, I can correct what column I want to place my marker in if the column is already full.
- As a player, I can correct what column I want to place my marker in if the column is less than COLMIN.
- As a player, I can correct what column I want to place my marker in if the column is greater than COLMAX.
- As a player, I can choose to play again if I want to play another ConnectX game after a victory.
- As a player, I can exit the game when I do not want to play any more ConnectX games.
- As a player, I can visually view the Connect X board before my turn.
- As a player, I can visually view the Connect X board after my turn.
- As a player, I can view the completed board after a win condition occurs.
- As either player, I can place the “NumToWin” horizontal consecutive token in a row in order to win the game.
- As either player, I can place the “NumToWin” vertical consecutive token in a row to win the game.
- As either player, I can place the “NumToWin” diagonal consecutive token in a row to win the game.
- As either player, I can restart the game after a tie occurs.
- As a player, I can choose the number of players in the game.
- As a player, I can choose the number of players again if my previous input was invalid.
- As a player, I can enter the character to represent each player in the game.
- As a player, I can enter the character to represent each player in the game again if the character is used by a previous character.
- As a player, I can enter the character to represent each player in the game again if I accidently entered a blank space.
- As a player, I can enter the amount of rows to be on the game board.
- As a player, I can enter the amount of rows again if the previous input was invalid.
- As a player, I can enter the amount of columns to be on the game board.
- As a player, I can enter the amount of columns again if the previous input was invalid.
- As a player, I can enter the amount of tokens in a row needed to win.
- As a player I can enter the amount of tokens needed in a row to win again if I entered an invalid input.
- As a player, I can choose to play a fast game.
- As a player, I can choose to play a memory efficient game.
- As a player, I can correct my input at this stage if I chose neither a fast or memory efficient game.

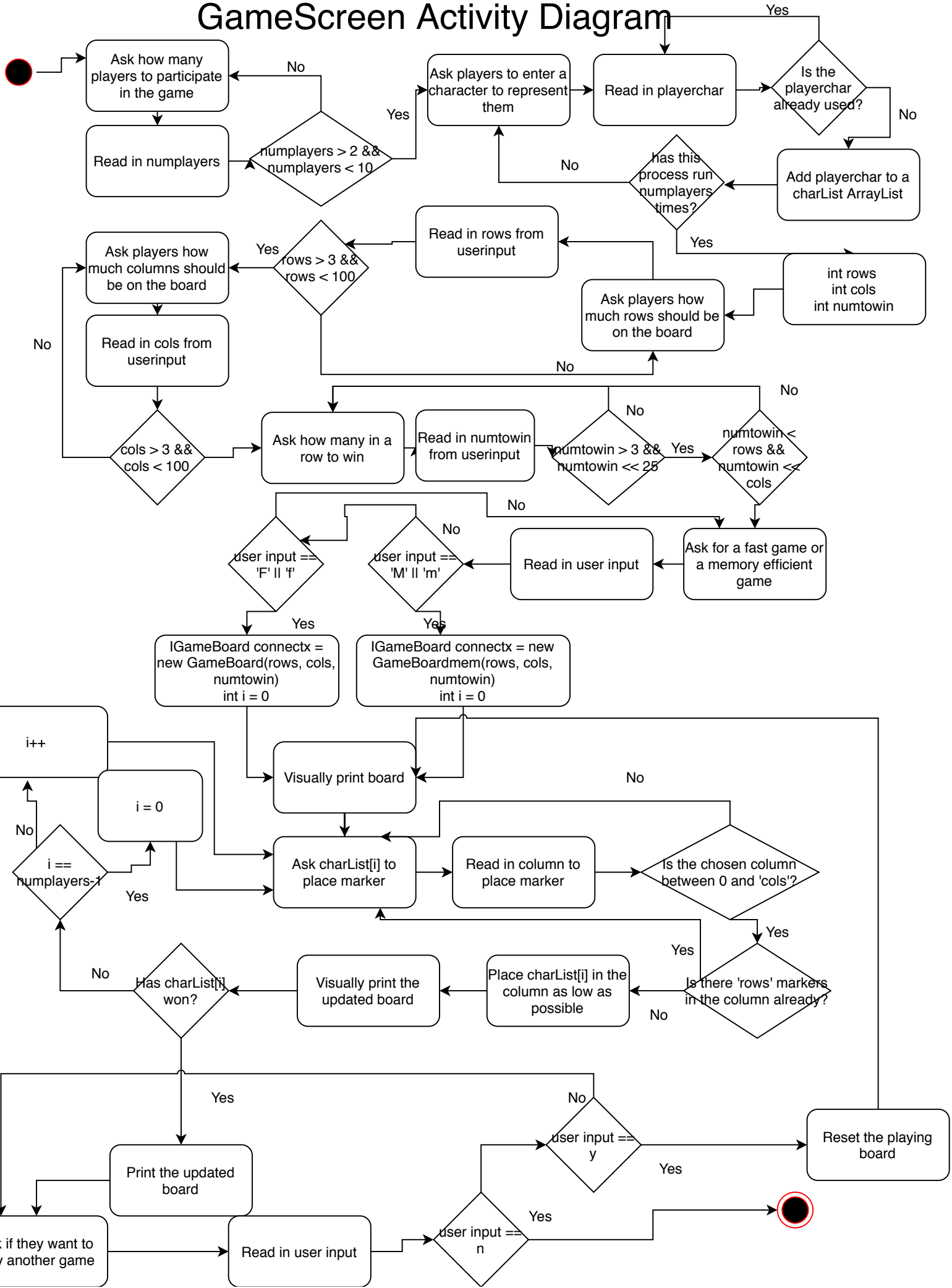
Non-Functional Requirements:

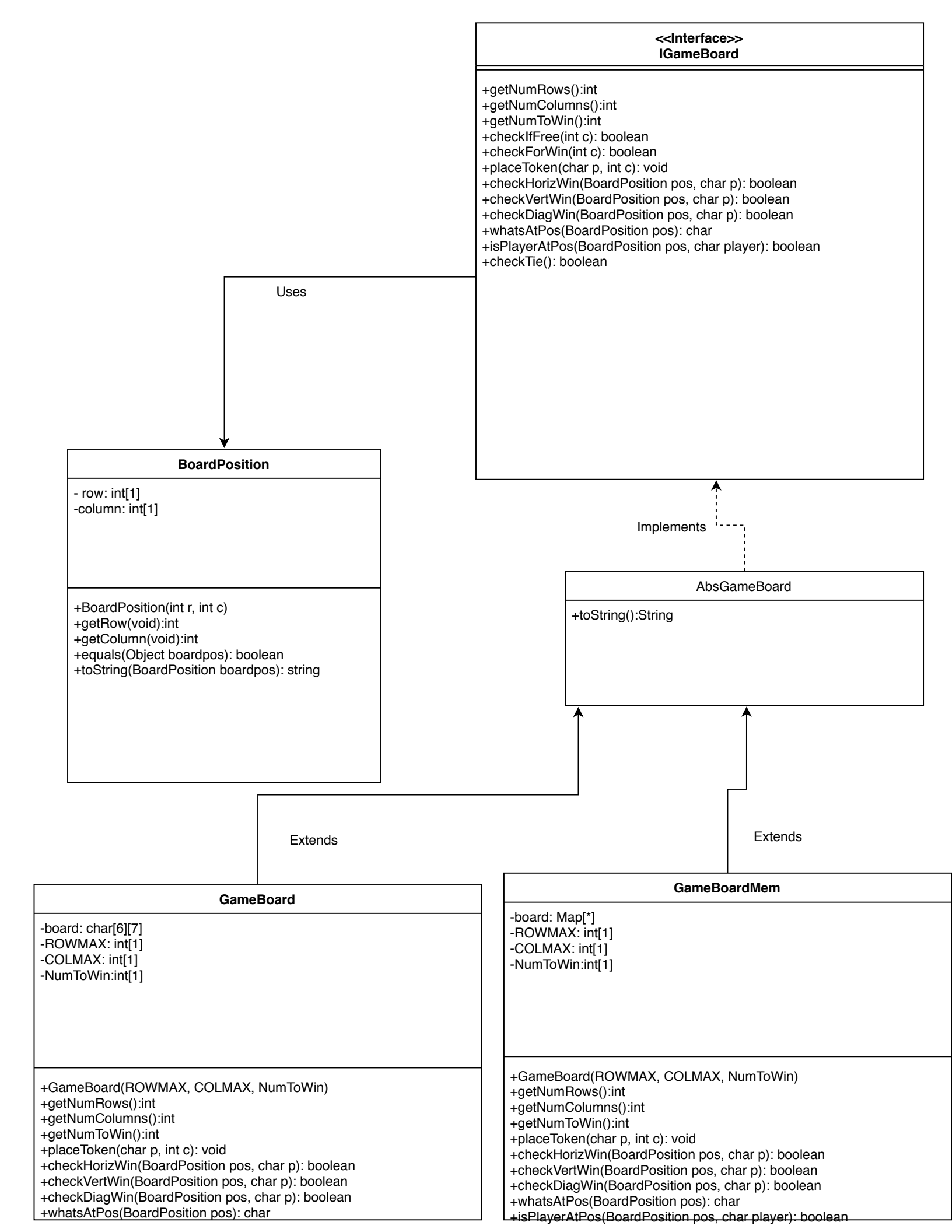
- Program must be able to run on all Clemson University Linux machines.
- The response a user inputs should take no longer than 50 milliseconds when placing markers.
- The final program should be smaller than 1 megabyte.
- The program must work with keyboard input.
- Should be able to infinitely play more games without crashing.
- Should be able to infinitely play more games without any bugs appearing.
- Should include a license stating that this project is not affiliated with *Connect Four* released in 1974 and any resemblance is purely coincidence and does not infringe on the *Connect Four* intellectual property.
- The Operating System should be able to do other tasks normally without any slowdown while simultaneously playing the game.
- The printed board should scale correctly with any resolution above 640x480.
- The program should be portable and run on any system after being unzipped.
- The learning time and usability of the program should be able to be learned after just reading the instructions in the documentation and program.
- The machine running this program must have the latest Java installed.

Game Screen Class Diagram

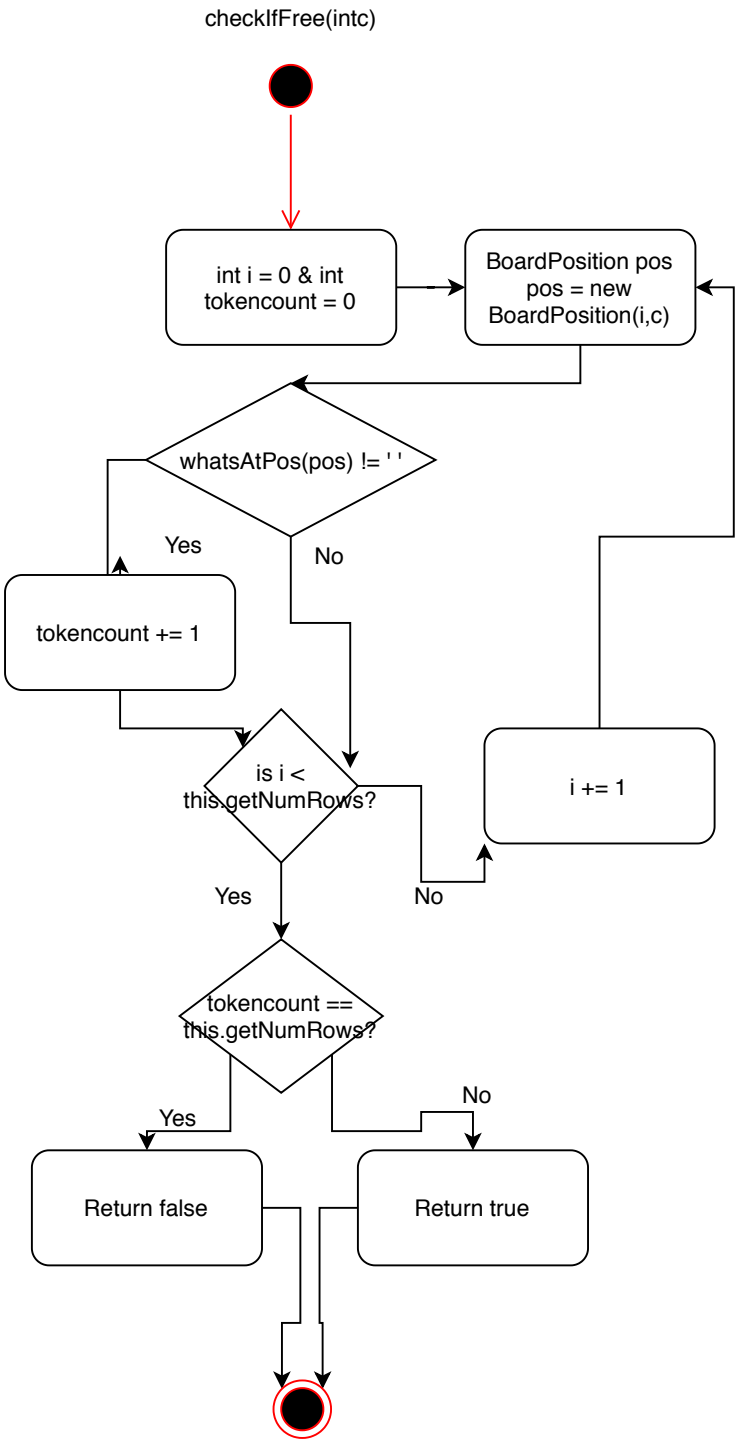


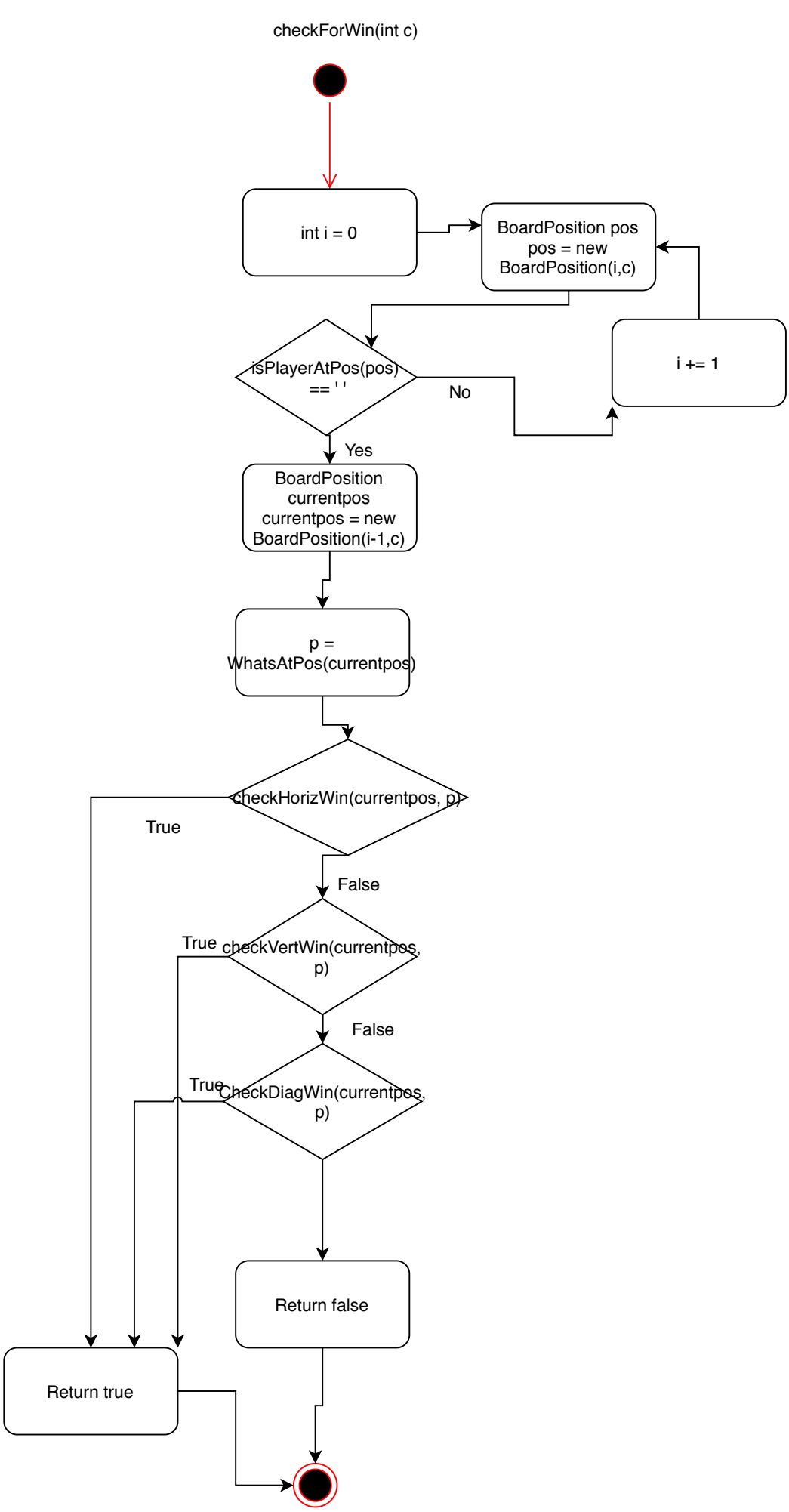
GameScreen Activity Diagram



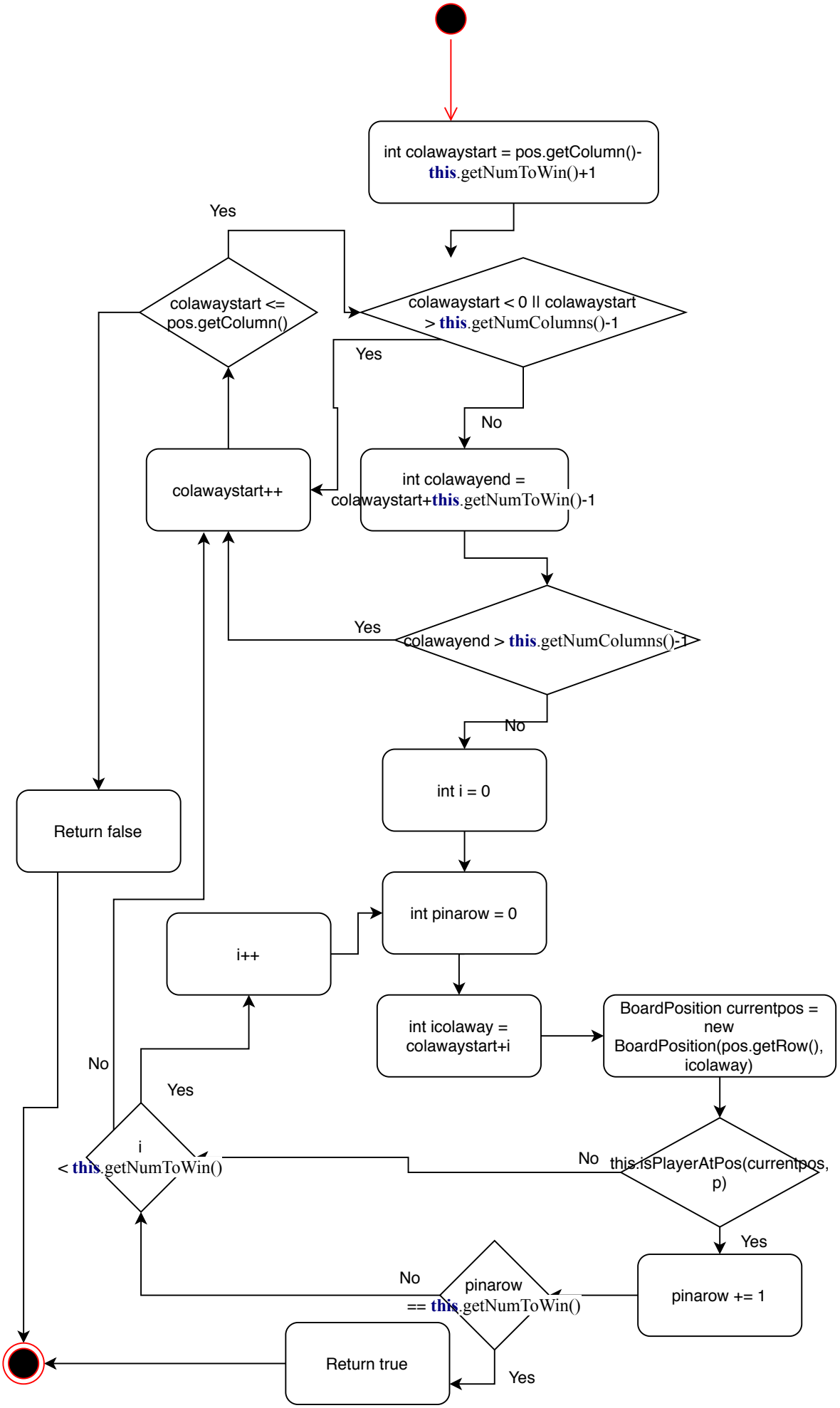

<<Interface>>
IGameBoard
+getNumRows():int
+getNumColumns():int
+getNumToWin():int
+checkIfFree(int c): boolean
+checkForWin(int c): boolean
+placeToken(char p, int c): void
+checkHorizWin(BoardPosition pos, char p): boolean
+checkVertWin(BoardPosition pos, char p): boolean
+checkDiagWin(BoardPosition pos, char p): boolean
+whatsAtPos(BoardPosition pos): char
+isPlayerAtPos(BoardPosition pos, char player): boolean
+checkTie(): boolean
Uses
BoardPosition
- row: int[1]
-column: int[1]
+BoardPosition(int r, int c)
+getRow(void):int
+getColumn(void):int
+equals(Object boardpos): boolean
+toString(BoardPosition boardpos): string
Implements
AbsGameBoard
+toString():String
Extends
Extends
GameBoard
-board: char[6][7]
-ROWMAX: int[1]
-COLMAX: int[1]
-NumToWin:int[1]
+GameBoard(ROWMAX, COLMAX, NumToWin)
+getNumRows():int
+getNumColumns():int
+getNumToWin():int
+placeToken(char p, int c): void
+checkHorizWin(BoardPosition pos, char p): boolean
+checkVertWin(BoardPosition pos, char p): boolean
+checkDiagWin(BoardPosition pos, char p): boolean
+whatsAtPos(BoardPosition pos): char
GameBoardMem
-board: Map[*]
-ROWMAX: int[1]
-COLMAX: int[1]
-NumToWin:int[1]
+GameBoard(ROWMAX, COLMAX, NumToWin)
+getNumRows():int
+getNumColumns():int
+getNumToWin():int
+placeToken(char p, int c): void
+checkHorizWin(BoardPosition pos, char p): boolean
+checkVertWin(BoardPosition pos, char p): boolean
+checkDiagWin(BoardPosition pos, char p): boolean
+whatsAtPos(BoardPosition pos): char
+isPlayerAtPos(BoardPosition pos, char player): boolean

Secondary Default IGameBoard Methods

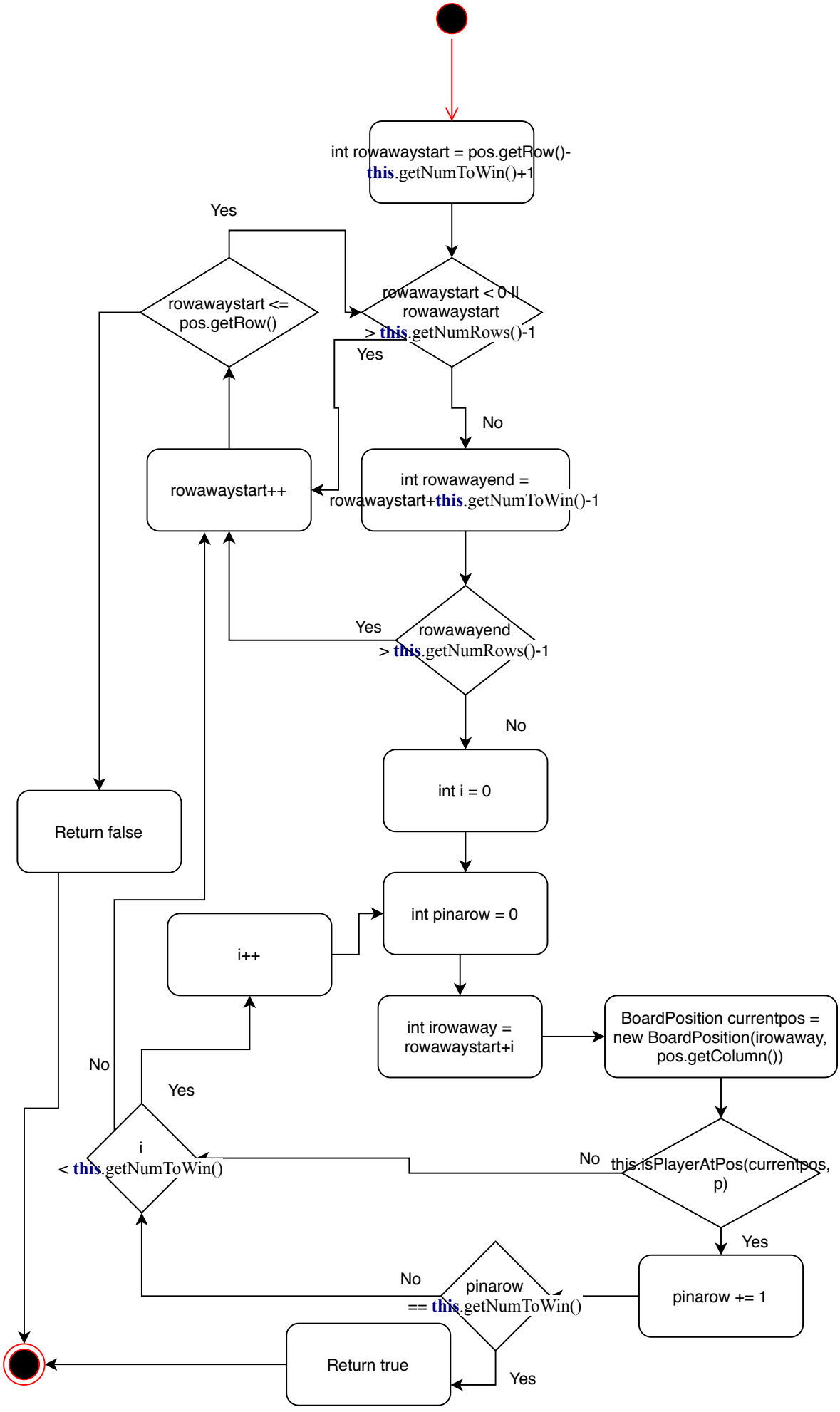

checkIfFree(intc)
int i = 0 & int tokencount = 0
BoardPosition pos pos = new BoardPosition(i,c)
whatsAtPos(pos) != ' '
Yes
No
tokencount += 1
is i < this.getNumRows?
i += 1
Yes
No
tokencount == this.getNumRows?
Yes
No
Return false
Return true

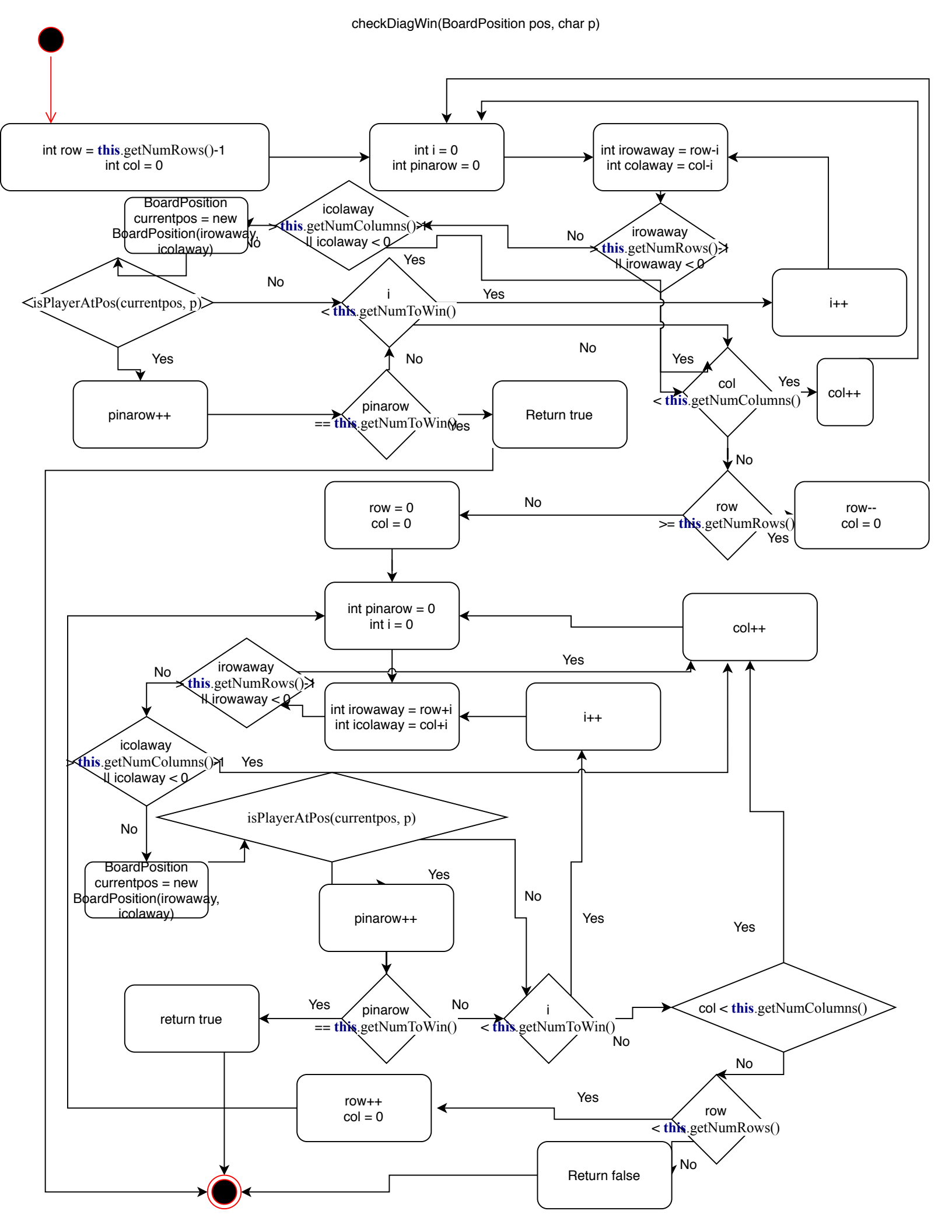

checkForWin(int c)
int i = 0
BoardPosition pos pos = new BoardPosition(i,c)
i += 1
isPlayerAtPos(pos) == ' '
No
Yes
BoardPosition currentpos currentpos = new BoardPosition(i-1,c)
p = WhatsAtPos(currentpos)
checkHorizWin(currentpos, p)
True
False
checkVertWin(currentpos, p)
True
False
CheckDiagWin(currentpos, p)
True
Return false
Return true

checkHorizWin(BoardPosition pos, char p)

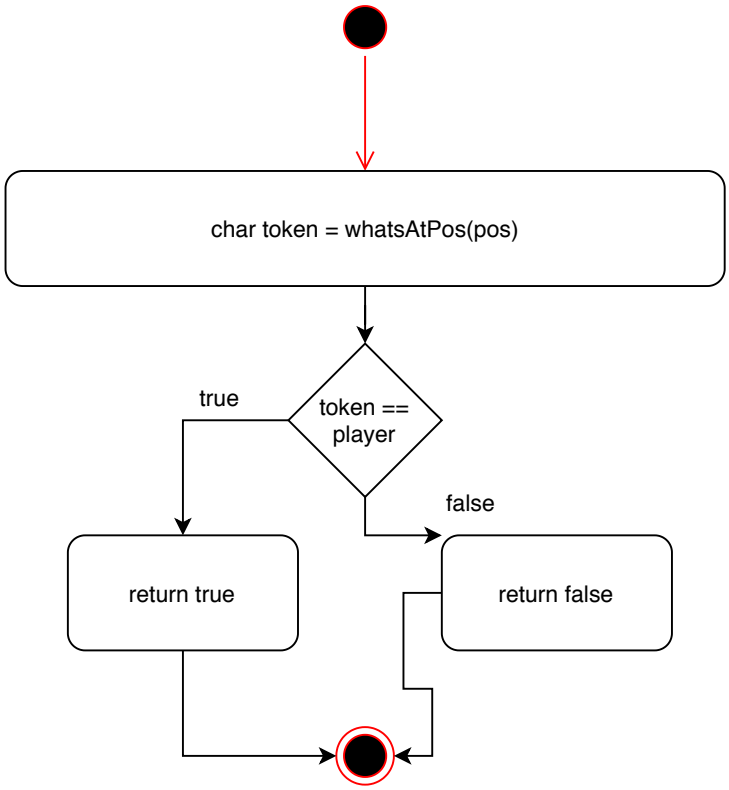


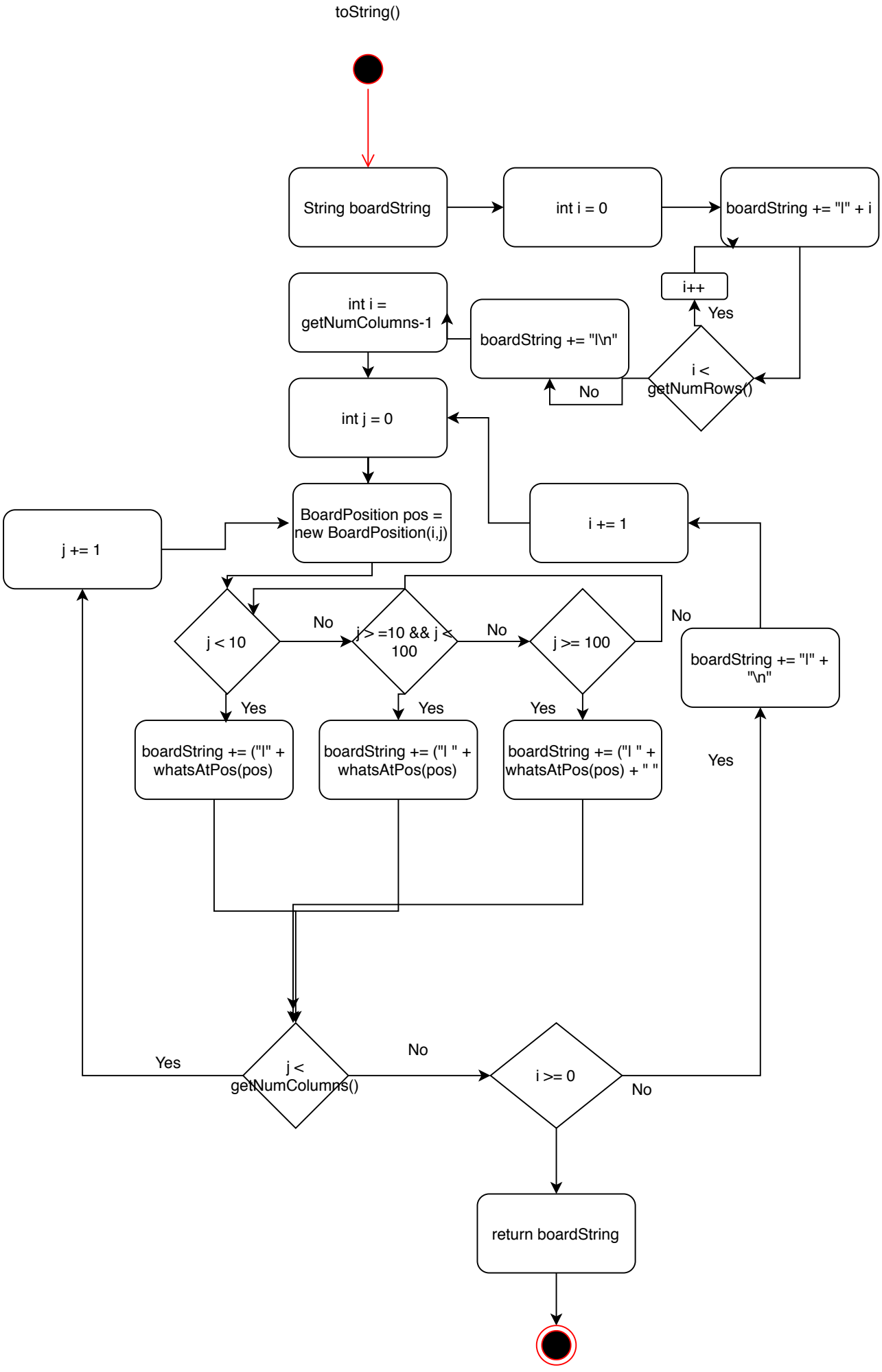
checkVertWin(BoardPosition pos, char p)

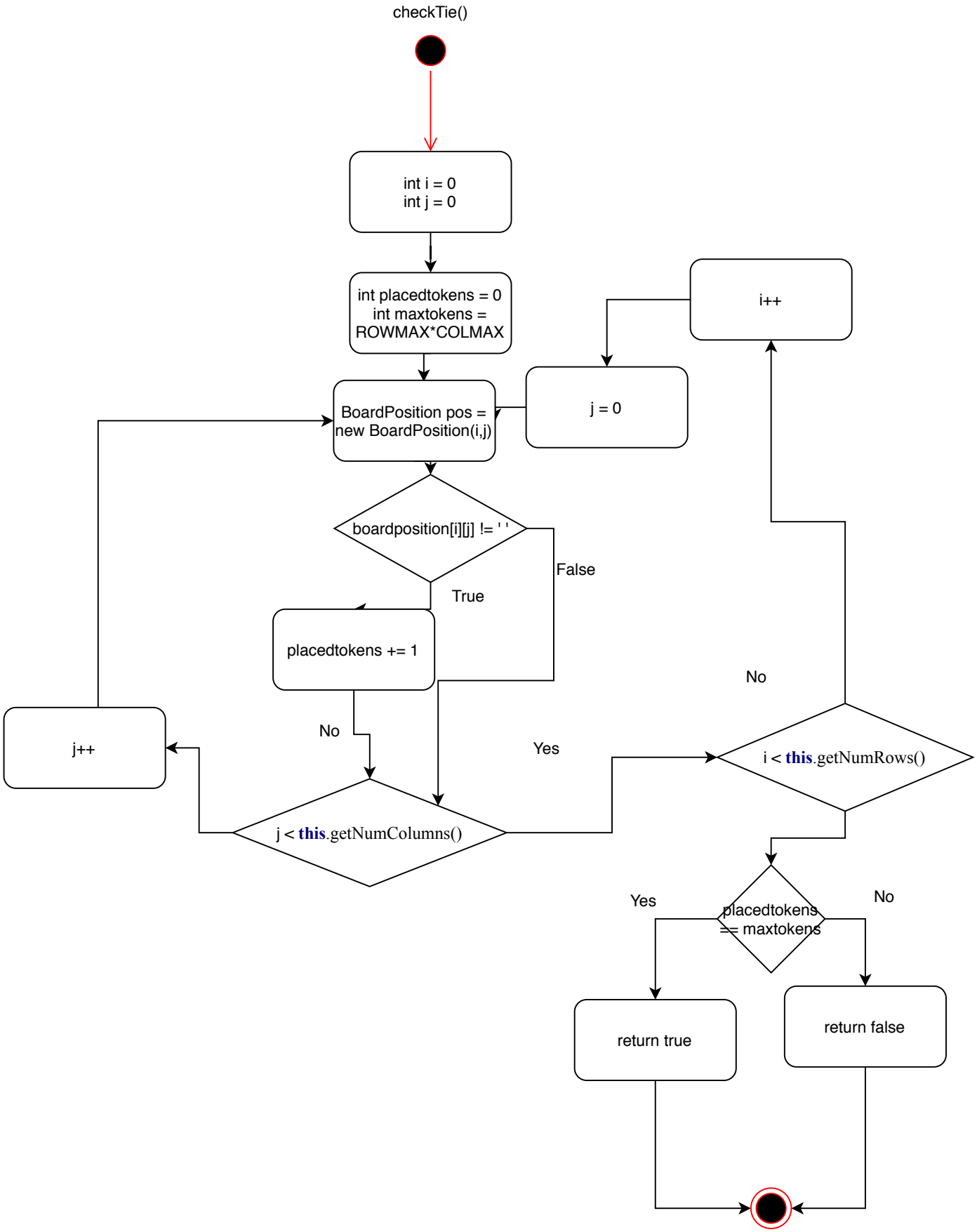



checkDiagWin(BoardPosition pos, char p)
int row = this.getNumRows()-1
int col = 0
int i = 0
int pinarow = 0
int irowaway = row-i
int colaway = col-i
BoardPosition currentpos = new BoardPosition(irowaway, icolaway)
icolaway this.getNumColumns()>I || icolaway < 0
irowaway this.getNumRows()-1 || irowaway < 0
isPlayerAtPos(currentpos, p)
i < this.getNumToWin()
i++
pinarow++
pinarow == this.getNumToWin()
Return true
col < this.getNumColumns()
col++
row >= this.getNumRows()
row--
col = 0
row = 0
col = 0
int pinarow = 0
int i = 0
col++
irowaway this.getNumRows()-1 || irowaway < 0
int irowaway = row+i
int icolaway = col+i
i++
icolaway this.getNumColumns()>I || icolaway < 0
isPlayerAtPos(currentpos, p)
BoardPosition currentpos = new BoardPosition(irowaway, icolaway)
pinarow++
return true
pinarow == this.getNumToWin()
i < this.getNumToWin()
col < this.getNumColumns()
row++
col = 0
row < this.getNumRows()
Return false
Yes
No

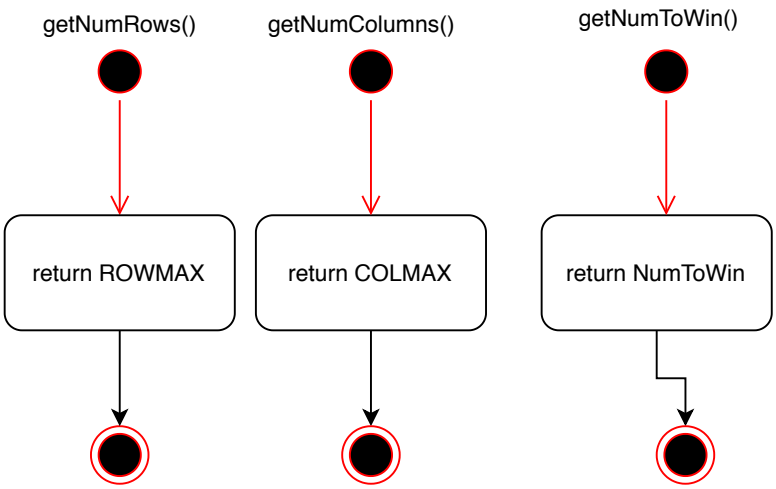
isPlayerAtPos(BoardPosition pos, char player)



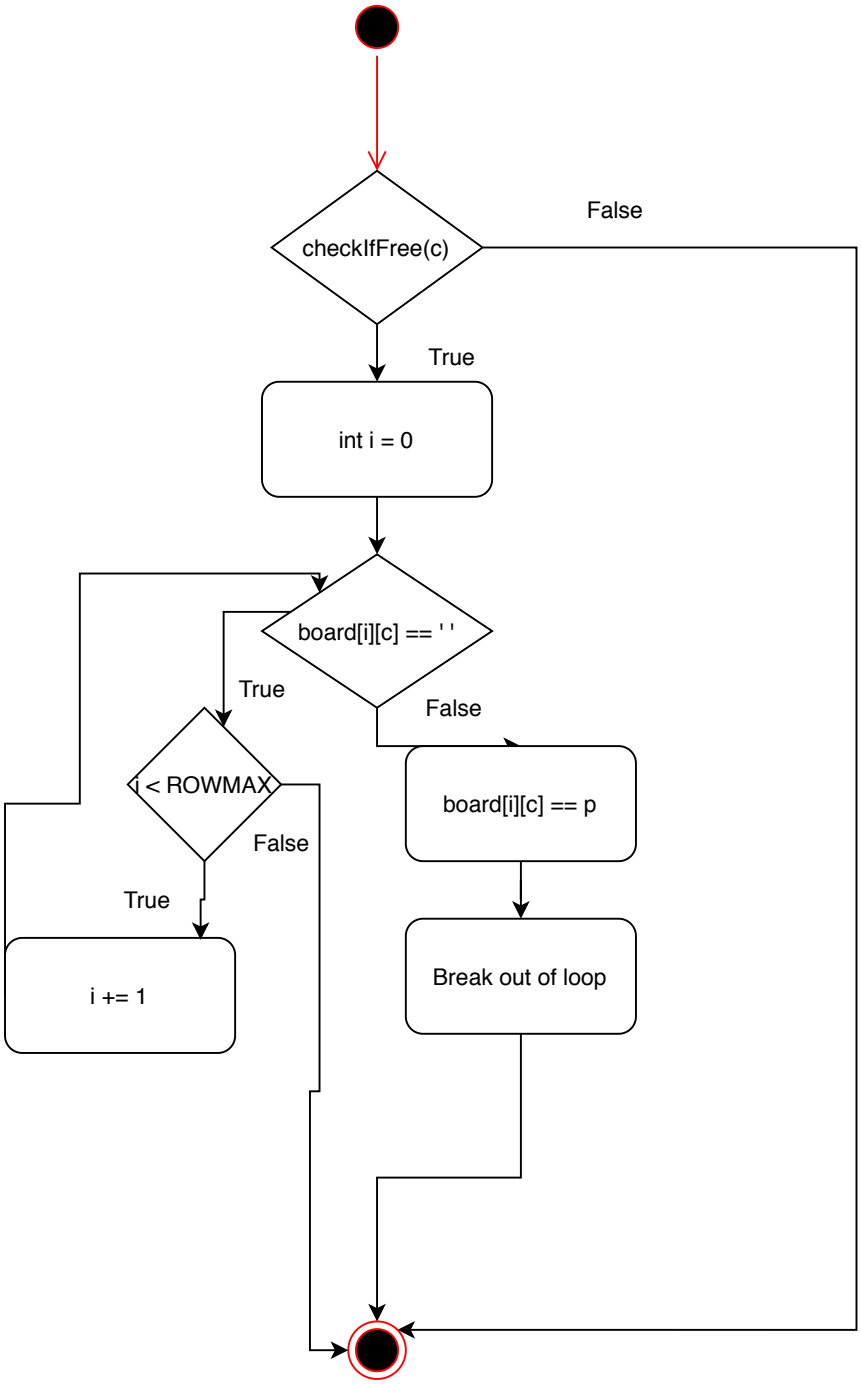

toString()
String boardString
int i = 0
boardString += "I" + i
i++
Yes
i < getNumRows()
No
boardString += "I\n"
int i = getNumColumns-1
int j = 0
BoardPosition pos = new BoardPosition(i,j)
i += 1
j += 1
j < 10
No
j >=10 && j < 100
No
j >= 100
No
boardString += "I" + "\n"
Yes
Yes
Yes
boardString += ("I" + whatsAtPos(pos)
boardString += ("I " + whatsAtPos(pos)
boardString += ("I " + whatsAtPos(pos) + " "
Yes
Yes
j < getNumColumns()
No
i >= 0
No
return boardString


checkTie()
int i = 0
int j = 0
int placedtokens = 0
int maxtokens =
ROWMAX*COLMAX
BoardPosition pos =
new BoardPosition(i,j)
j = 0
i++
boardposition[i][j] != ' '
False
True
placedtokens += 1
No
Yes
j++
j < this.getNumColumns()
i < this.getNumRows()
No
placedtokens
== maxtokens
Yes
No
return true
return false

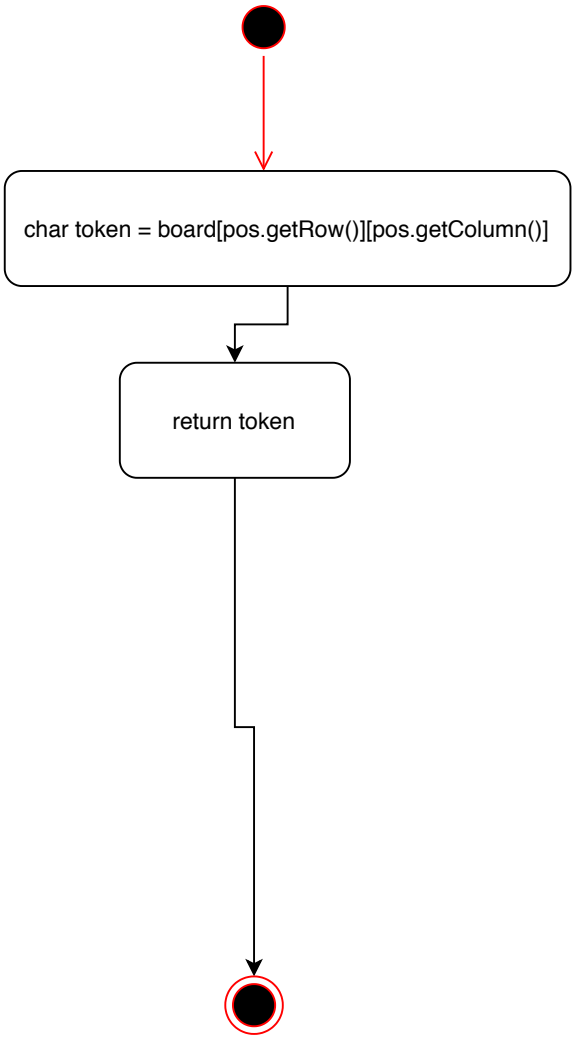
GameBoard Diagrams Next Page


getNumRows()
getNumColumns()
getNumToWin()
return ROWMAX
return COLMAX
return NumToWin

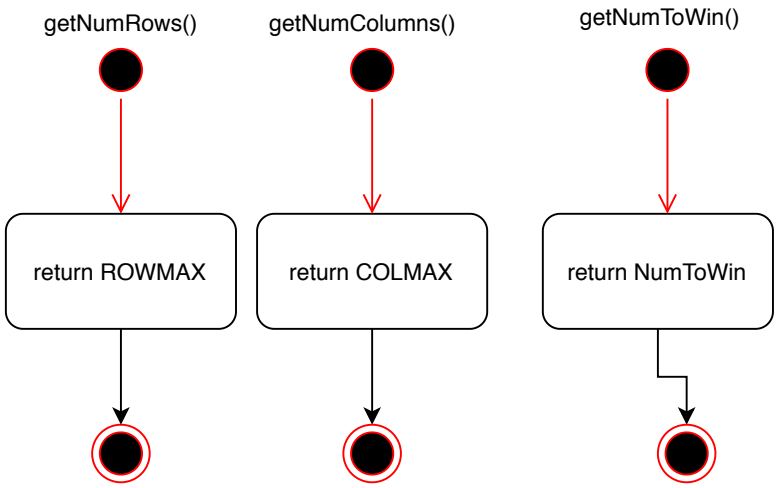
placeToken(char p, int c)

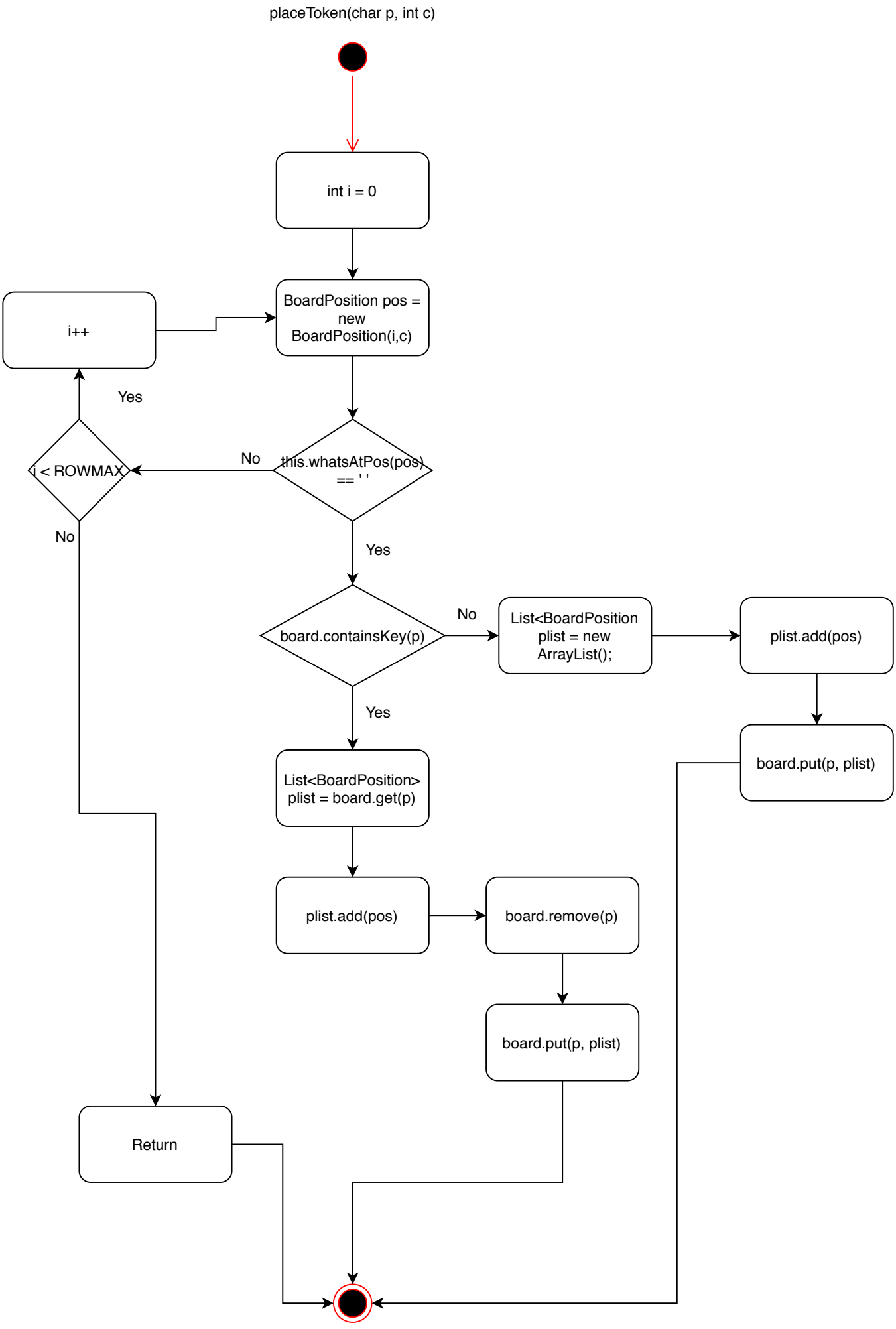


whatsAtPos(BoardPosition pos, char player)

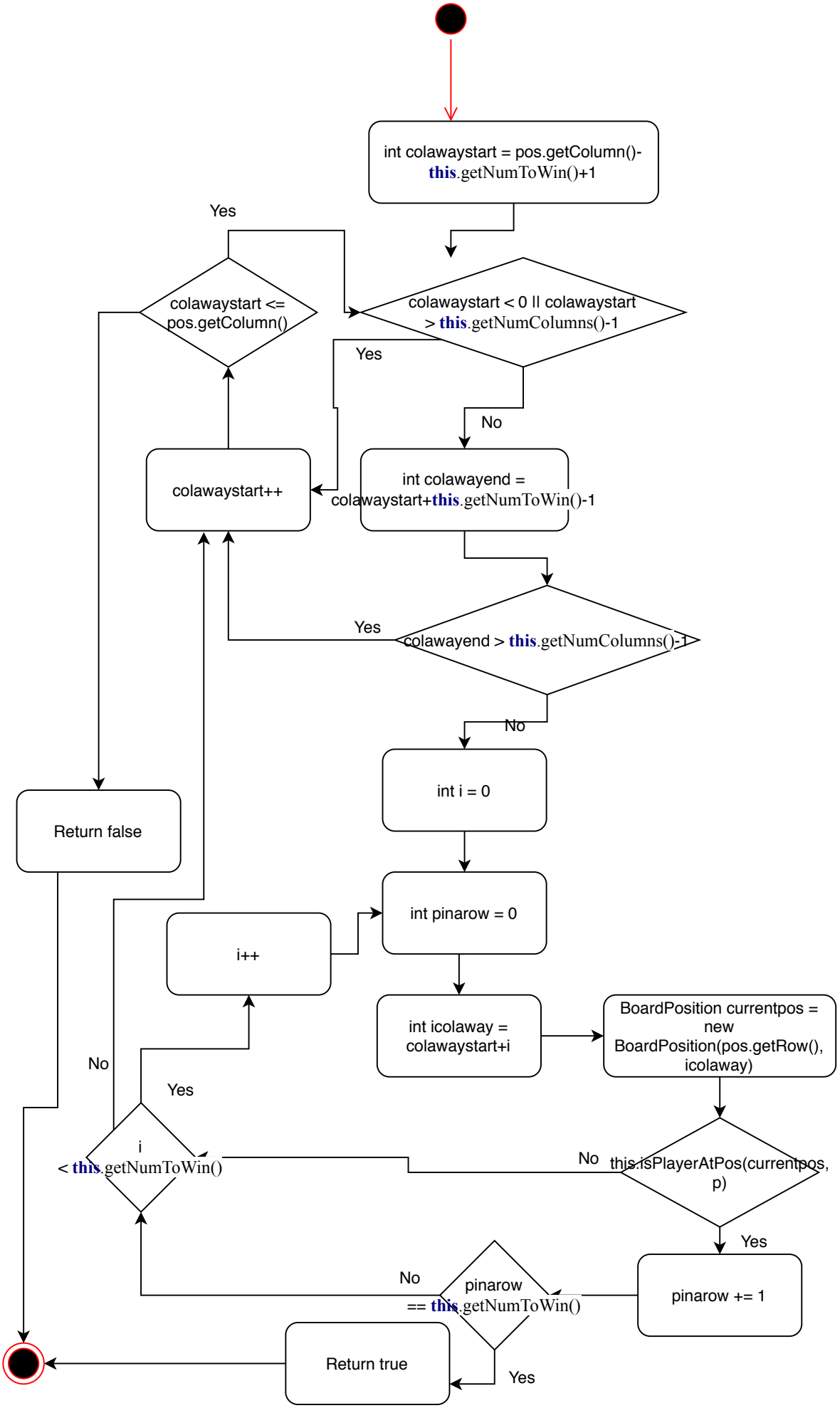


GameBoardMem Diagrams Next Page

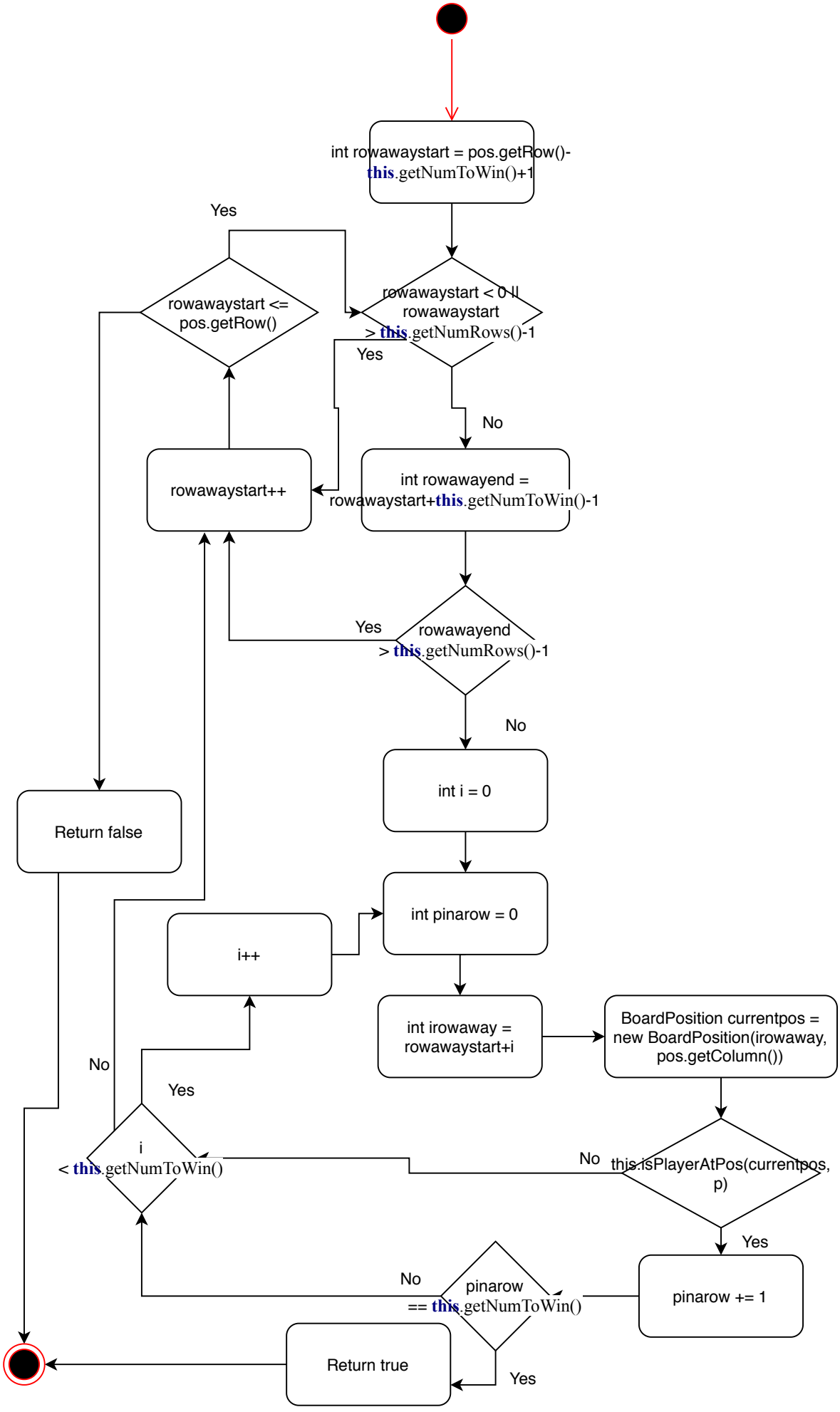

getNumRows()
getNumColumns()
getNumToWin()
return ROWMAX
return COLMAX
return NumToWin

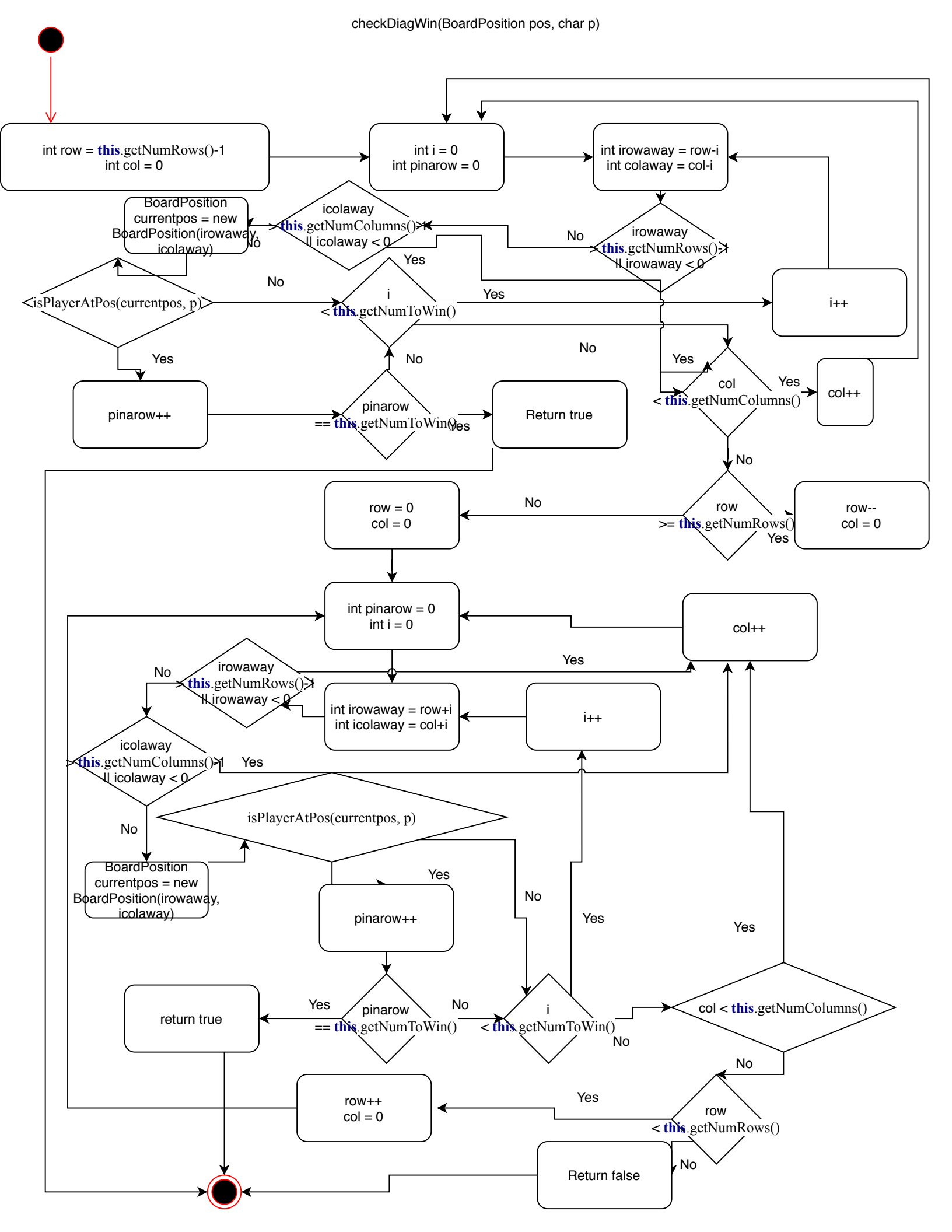

placeToken(char p, int c)
int i = 0
BoardPosition pos = new BoardPosition(i,c)
this.whatsAtPos(pos) == ' '
No
i < ROWMAX
Yes
i++
No
Yes
board.containsKey(p)
No
List<BoardPosition plist = new ArrayList();
plist.add(pos)
board.put(p, plist)
Yes
List<BoardPosition> plist = board.get(p)
plist.add(pos)
board.remove(p)
board.put(p, plist)
Return

checkHorizWin(BoardPosition pos, char p)

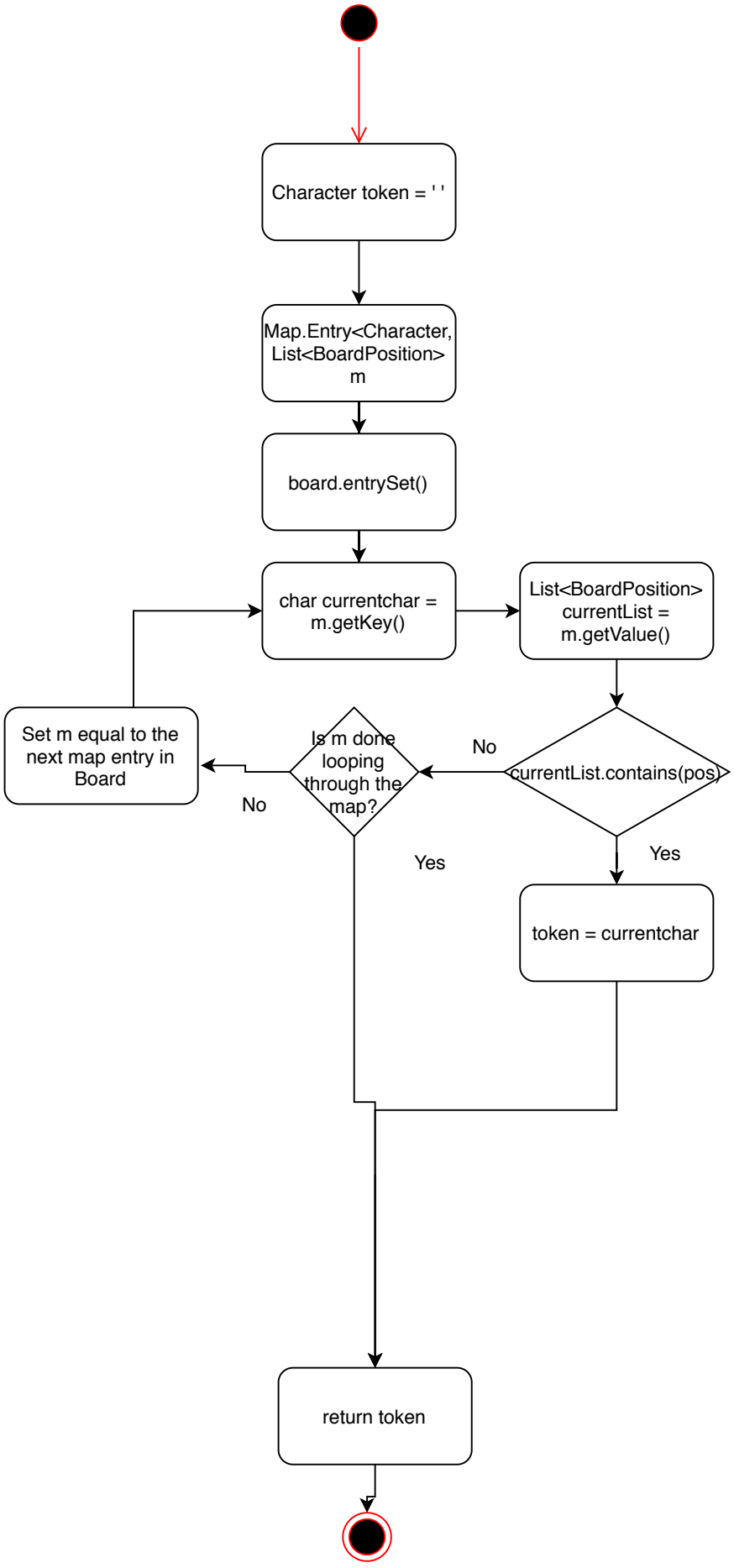


checkVertWin(BoardPosition pos, char p)



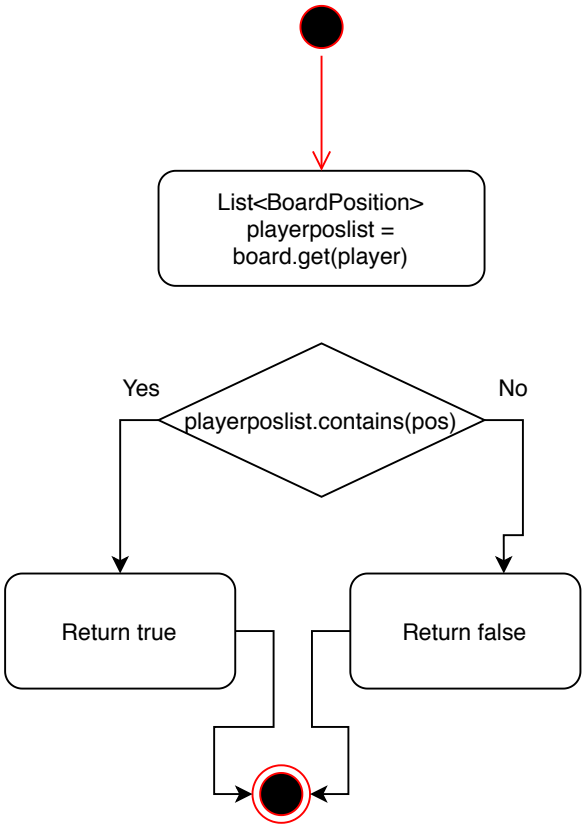

checkDiagWin(BoardPosition pos, char p)
int row = this.getNumRows()-1
int col = 0
int i = 0
int pinarow = 0
int irowaway = row-i
int colaway = col-i
BoardPosition currentpos = new BoardPosition(irowaway, icolaway)
icolaway this.getNumColumns()>I || icolaway < 0
irowaway this.getNumRows()-1 || irowaway < 0
isPlayerAtPos(currentpos, p)
i < this.getNumToWin()
i++
pinarow++
pinarow == this.getNumToWin()
Return true
col < this.getNumColumns()
col++
row >= this.getNumRows()
row--
col = 0
row = 0
col = 0
int pinarow = 0
int i = 0
col++
irowaway this.getNumRows()-1 || irowaway < 0
int irowaway = row+i
int icolaway = col+i
i++
icolaway this.getNumColumns()>I || icolaway < 0
isPlayerAtPos(currentpos, p)
BoardPosition currentpos = new BoardPosition(irowaway, icolaway)
pinarow++
return true
pinarow == this.getNumToWin()
i < this.getNumToWin()
col < this.getNumColumns()
row++
col = 0
row < this.getNumRows()
Return false
Yes
No

whatsAtPos(BoardPosition pos)



override

isPlayerAtPos(BoardPosition pos, char player)



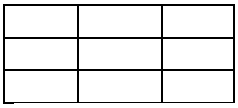
Testing Cases:

Constructor(3):

```
public void testsmallestBoard_constructor()
```

Input:
State:
Rows = 3
Cols = 3
NumToWin = 3

Output:
State



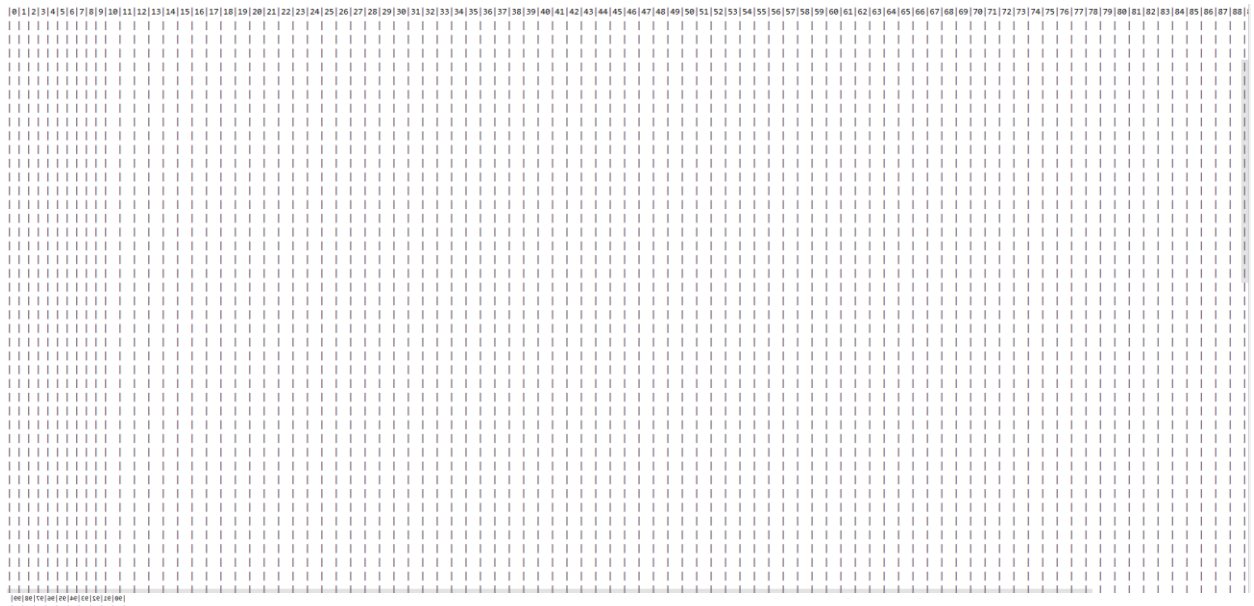
Connectx.getNumRows() = 3
Connectx.getNumColumns() = 3
Connectx.getNumToWin() = 3

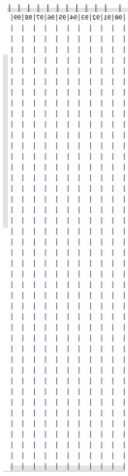
Reason:
This test case is unique and distinct because it is the smallest board the constructor can make.

```
public void testbiggestBoard_constructor()
```

Input:
State:
Rows = 100
Cols = 100
NumToWin = 25

Output:
State





Connectx.getNumRows() = 100
Connectx.getNumColumns() = 100
Connectx.getNumToWin() = 25

Reason:
This test case is unique and distinct because it is the biggest board the constructor can make.

```
public void testnormalBoard_constructor()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

Output:
State

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
```

Connectx.getNumRows() = 6
Connectx.getNumColumns() = 7
Connectx.getNumToWin() = 4

Reason:
This test case is unique and distinct because it is the standard board that Connect 4 is played on.

CheckIfFree(3):

```
public void testemptyBoard_checkIfFree()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
```

Output:
connectx.checkIfFree(0) == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests column 0 if the empty board has free space.

```
public void testfullBoard_checkIfFree()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
```

Output:
connectx.checkIfFree(0) == false

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests column 0 if the full board has free space, which should return false instead of true.

```
public void testhalffullBoard_checkIfFree()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
```

Output:
connectx.checkIfFree(0) == true

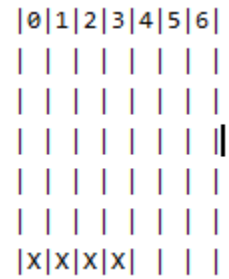
State of the board is unchanged

Reason:
This test case is unique and distinct because it tests column 0 if the half full board has free space, which should return true unlike the previous one.

CheckHorizWin(4):

```
public void testrow0leftmost_checkHorizWin()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4



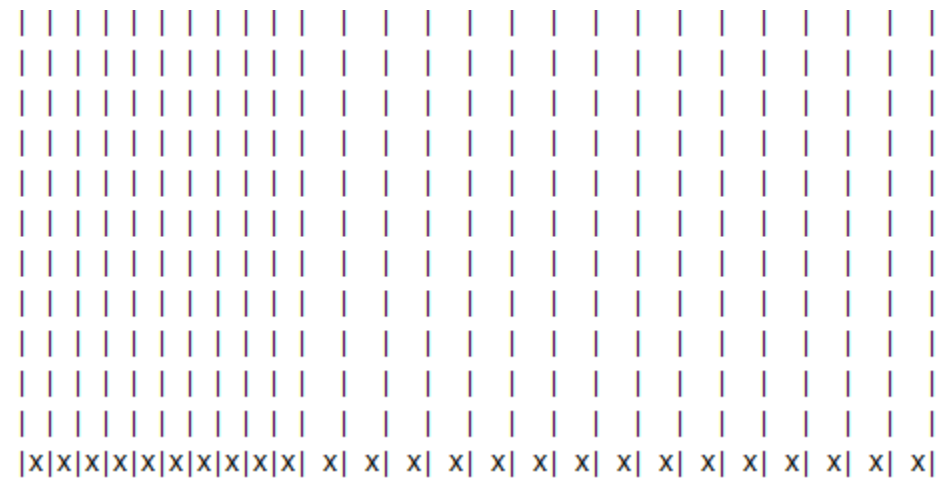
Pos(row = 0, col = 2)
checkHorizWin(pos, ‘X’)
Output:
checkHorizWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a bottom left win if the last token was placed in the third column.

```
public void testmaxNumToWin_checkHorizWin()
```

Input:
State:
Rows = 100
Cols = 100
NumToWin = 25



Pos(row = 0, col = 0)
checkHorizWin(pos, ‘X’)
Output:
checkHorizWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a bottom left Horizontal win if NumToWin was set to the max (25) on the largest board.

```
public void testlowestNumToWin_checkHorizWin()
```

Input:
State:
Rows = 3
Cols = 3
NumToWin = 3

```
|0|1|2|
| | | |
| | | |
|X|X|X|
```

Pos(row = 0, col = 0)
checkHorizWin(pos, ‘X’)
Output:
checkHorizWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a bottom left Horizontal win if NumToWin was set to the lowest (3) on the smallest board.

```
public void testtoprightmost_checkHorizWin()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | |O|O|O|O|
| | | |X|X|X|X|
| | | |X|X|X|X|
| | | |X|X|X|X|
| | | |X|X|X|X|
```

Pos(row = 4, col = 6)
checkHorizWin(pos, 'O')
Output:
checkHorizWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a top right horizontal win from the fourth token, and also tests if a token other than X can win.

CheckVertWin(4):

```
public void testbottomleft_checkVertWin()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
```

Pos(row = 3, col = 0)
checkVertWin(pos, 'X')
Output:
checkVertWin == true

State of the board is unchanged

Reason:

This test case is unique and distinct because it tests a bottom left vertical win from the fourth placed token.

```
public void testbottomright_checkVertWin()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | |X|
| | | | | | |X|
| | | | | | |X|
| | | | | | |X|
```

Pos(row = 0, col = 6)
checkVertWin(pos, ‘X’)
Output:
checkVertWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a bottom right vertical win from the first placed token.

```
public void testmiddlemiddle_checkVertWin()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 5

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | |X| | | |
| | | |X| | | |
| | | |X| | | |
| | | |X| | | |
| | | |X| | | |
```

Pos(row = 1, col = 3)
checkVertWin(pos, ‘X’)
Output:

checkVertWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a middle right vertical win with 5 needed to win from the second placed token.

```
public void testtopright_checkVertWin()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4
```

```
|0|1|2|3|4|5|6|
| | | | | | |X|
| | | | | | |X|
| | | | | | |X|
| | | | | | |X|
| | | | | | |O|
| | | | | | |O|
```

Pos(row = 2, col = 6)
checkVertWin(pos, 'X')
Output:
checkVertWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a top right vertical win if X is not at the bottom.

CheckDiagWin(7):

```
public void testbottomleftAscending_checkDiagWin()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4
```

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | |X| | | |
| | |X|X| | | |
| |X|X|X| | | |
|X|X|X|X| | | |
```

Pos(row = 2, col = 2)
checkDiagWin(pos, ‘X’)
Output:
checkDiagWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a bottom left ascending diagonal win if the token is placed between the second and fourth tokens.

```
public void testtopleftDescending_checkDiagWin()
```
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
|X| | | | | | |
|X|X| | | | | |
|X|X|X| | | | |
|X|X|X|X| | | |
```

Pos(row = 3, col = 0)
checkDiagWin(pos, ‘X’)
Output:
checkDiagWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a top left descending diagonal win if the token is placed from the fourth consecutive token spot.

```
public void testbottomrightDescending_checkDiagWin()
```
Input:
State:

Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | |X| | | |
| | | |X|X| | |
| | | |X|X|X| |
| | | |X|X|X|X|
```

Pos(row = 2, col = 4)
checkDiagWin(pos, 'X')
Output:
checkDiagWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a bottom right descending diagonal win if the token is placed from the second consecutive token spot.

```
public void testbottommiddleAscending_checkDiagWin()
```
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | |X|
| | | | | |X|X|
| | | | |X|X|X|
| | | |X|X|X|X|
```

Pos(row = 0, col = 3)
checkDiagWin(pos, 'X')
Output:
checkDiagWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a bottom middle ascending win if the first consecutive token is placed.

```
public void testmiddlemiddleDescending_checkDiagWin()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

|0|1|2|3|4|5|6|
|X| | | | | | |
|X|X| | | | | |
|X|X|X| | | | |
|X|X|X|X| | | |
|X|X|X|X| | | |
|X|X|X|X| | | |

Pos(row = 0, col = 3)
checkDiagWin(pos, 'X')
Output:
checkDiagWin == true

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a middle middle ascending
diagonal win if the token is placed from the fourth consecutive token spot.


public void testmiddlemiddleAscending_checkDiagWin()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

|0|1|2|3|4|5|6|
| | | | | | |K|
| | | | | |K|K|
| | | | |K|K|K|
| | | |K|K|K|K|
| | | |K|K|K|K|
| | | |K|K|K|K|

Pos(row = 2, col = 3)
checkDiagWin(pos, 'K')
Output:
checkDiagWin == true
```

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a middle middle ascending diagonal win if the token is placed from the first consecutive token spot, with the win not involving a token on the bottom of the board, and tokens other than ‘X’ and ‘O’ being able to win.

```
public void testbottommiddleAscending_checkDiagWinFALSE()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | |O| | | |
| | |X|X| | | |
| |X|X|X| | | |
|X|X|X|X| | | |

Pos(row = 0, col = 0)
checkDiagWin(pos, ‘X’)
Output:
checkDiagWin == false
```

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests a bottom middle ascending case if a diagonal win has not occurred after placing the first consecutive token.

CheckTie(4):

```
public void testnormalboard_checkTie()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4
```

```
|0|1|2|3|4|5|6|
|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|X|X|X|
|X|X|X|X|X|X|X|
|X|X|X|X|X|X|X|
```

checkTie()
Output:
checkTie() == True

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests whether a tie successfully occurs if the board is full.

```
public void testemptyboard_checkTie()
```
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
```

checkTie()
Output:
checkTie() == False

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests whether a tie successfully occurs if the board is empty, which should be false.

```
public void testlargestboard_checkTie()
```
Input:
State:
Rows = 100
Cols = 100
NumToWin = 25

checkTie()
Output:
checkTie() == True

State of the board is unchanged

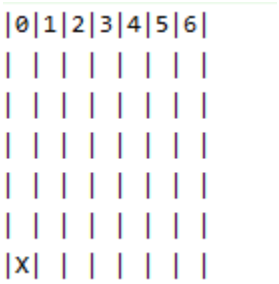
Reason:
This test case is unique and distinct because it tests whether a tie successfully occurs if the largest board is full, which should return true.

```
public void testhalffullboard_checkTie()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
|X|X|X|X|X|X|X|
|X|X|X|X|X|X|X|
|X|X|X|X|X|X|X|

checkTie()
Output:
checkTie() == False

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests whether a tie successfully occurs if the board is half full, which should return false.
```

WhatsAtPos(5):

```
public void testbottomleftx_WhatsAtPos()
Input:
State:
```

Rows = 6
Cols = 7
NumToWin = 4



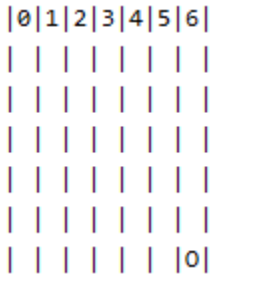
Pos(row = 0, col = 0)

WhatsAtPos(pos)
Output:
WhatsAtPos() == O

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests WhatsAtPos() can detect an X on the bottom left of the board.

```
public void testbottomrighto_WhatsAtPos()
```
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4



Pos(row = 0, col = 6)

WhatsAtPos(pos)
Output:
WhatsAtPos() == O

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests WhatsAtPos() can detect an O on the bottom right of the board.

```
public void testbottomleftblankspace_WhatsAtPos()
```
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
```

Pos(row = 0, col = 0)

WhatsAtPos(pos)
Output:
WhatsAtPos() == ‘ ‘

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests WhatsAtPos() can return a blank space if the bottom left is chosen.

```
public void testbottommiddlex_WhatsAtPos()
```
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |X| | |
```

Pos(row = 0, col = 4)

WhatsAtPos(pos)
Output:
WhatsAtPos() == X

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests WhatsAtPos() can return a X if a case not on the boundaries is chosen.

```
public void testtopmiddlex_WhatsAtPos()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4
```

```
|0|1|2|3|4|5|6|
| | | | |X| | |
| | | | |X| | |
| | | | |X| | |
| | | | |X| | |
| | | | |X| | |
| | | | |X| | |
```

```
Pos(row = 6, col = 4)

WhatsAtPos(pos)
Output:
WhatsAtPos() == X

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests WhatsAtPos() can return a X if a case not on the base level is chosen.
```

IsPlayerAtPos(5):

```
public void testbottomleftx_IsPlayerAtPos()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4
```

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|X| | | | | | |
```

```
Pos(row = 0, col = 0)
```

isPlayerAtPos(pos, ‘X’)
Output:
isPlayerAtPos == True

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests if IsPlayerAtPos finds X if it is in the bottom left.

```
public void testbottomleftFalse_IsPlayerAtPos()
```
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|O|X| | | | | |
```

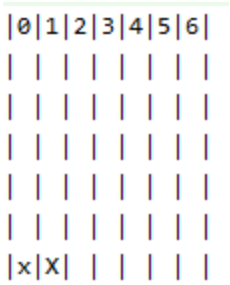
Pos(row = 0, col = 0)

isPlayerAtPos(pos, ‘O’)
Output:
isPlayerAtPos == False

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests if IsPlayerAtPos returns false if X is in the position instead of O. The X is placed on the board also since the precondition of IsPlayerAtPos states the player must already exist on the board to call isPlayerAtPos.

```
public void testbottomleftlowercasex_IsPlayerAtPos()
```
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4



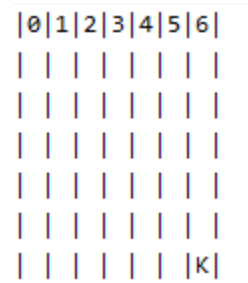
Pos(row = 0, col = 0)

isPlayerAtPos(pos, ‘X’)
Output:
isPlayerAtPos == False

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests if IsPlayerAtPos returns false if lowercase x is in the position instead of uppercase X. The X is placed on the board also since the precondition of IsPlayerAtPos states the player must already exist on the board to call isPlayerAtPos.

```
public void testbottomrightxK_IsPlayerAtPos()
```
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4



Pos(row = 0, col = 6)

isPlayerAtPos(pos, ‘K’)
Output:
isPlayerAtPos == True

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests if IsPlayerAtPos returns true if uppercase K is in the rightmost position, meaning symbols other than X and O can be detected.

```
public void testtopleftx_IsPlayerAtPos()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

|0|1|2|3|4|5|6|
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
|X| | | | | | |
Pos(row = 5, col = 0)

isPlayerAtPos(pos, 'X')
Output:
isPlayerAtPos == True

State of the board is unchanged

Reason:
This test case is unique and distinct because it tests if IsPlayerAtPos returns true
if X is in the top right most of the board.
```

PlaceToken(5):

```
public void testbottomleftx_PlaceToken()
Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

Output:
```

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|X| | | | | | |
```

Reason:
This test case is unique and distinct because it represents the first move a player can take, where a token will be at the bottom of the board.

```
public void testbottomrightK_PlaceToken()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
```

Output:

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | |K|
```

Reason:
This test case is unique and distinct because it represents the first move a player can take, where a token will be at the bottom of the board, and to see whether a token other than X and O can be placed.

```
public void almostfullcolumn_PlaceToken()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | |O|
| | | | | | |O|
| | | | | | |O|
| | | | | | |O|
| | | | | | |O|
```

Output:

```
|0|1|2|3|4|5|6|
| | | | | | |X|
| | | | | | |O|
| | | | | | |O|
| | | | | | |O|
| | | | | | |O|
| | | | | | |O|
```

Reason:
This test case is unique and distinct because it represents the token being placed in a column that is 1 away from being full.

```
public void leftfullboard_PlaceToken()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
|O| | | | | | |
|O| | | | | | |
|O| | | | | | |
|O| | | | | | |
|O| | | | | | |
|O| | | | | | |
```

Output:

```
|0|1|2|3|4|5|6|
|O| | | | | | |
|O| | | | | | |
|O| | | | | | |
|O| | | | | | |
|O| | | | | | |
|O|X| | | | | |
```

Reason:
This test case is unique and distinct because it represents the token being placed in a column that is 1 away a column that is already full.

```
public void middlehalffullboard_PlaceToken()
```

Input:
State:
Rows = 6
Cols = 7
NumToWin = 4

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | | | |O| | |
| | | | |O| | |
```

Output:

```
|0|1|2|3|4|5|6|
| | | | | | | |
| | | | | | | |
| | | | |K| | |
| | | | |O| | |
| | | | |O| | |
| | | | |O| | |
```

Reason:
This test case is unique and distinct because it represents the token being placed in a column that is only half full, with placing a token with a character not ‘X’ or ‘O’.

Instructions on how to Play:

1. Open a terminal to the source directory (there should be a “makefile” file).
2. Type “make” into the terminal and press enter in order to compile the game.
3. Type “make run” into the terminal and press enter in order to run the game (during first time use you must do step 2 before step 3 or else the game will not run).
4. Type “make clean” in order to delete the class files, but note that in order to play the game after this you must start with step 2 again instead of step 3.

Instructions on how to use test runs:

1. Open a terminal to the source directory (there should be a “makefile” file).
2. Type “test” to compile the tests.
1. Type “testGB” to test the GameBoard implementation of ConnectX.
2. Type “testGBMem” to test the GameBoard Memory implementation of ConnectX.