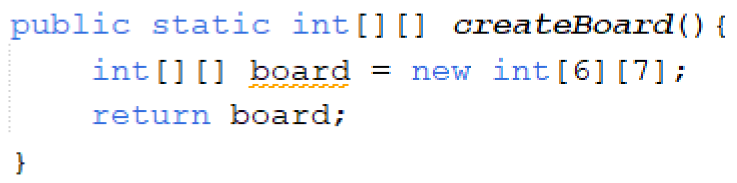
Connect 4 AI agent

I- The connect 4 game:

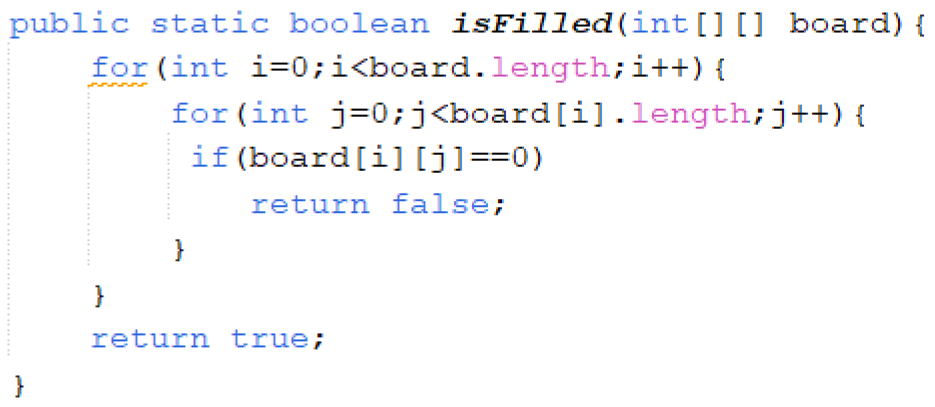
The board:

We used the method createBoard() to create a 2D array with dimensions 6x7



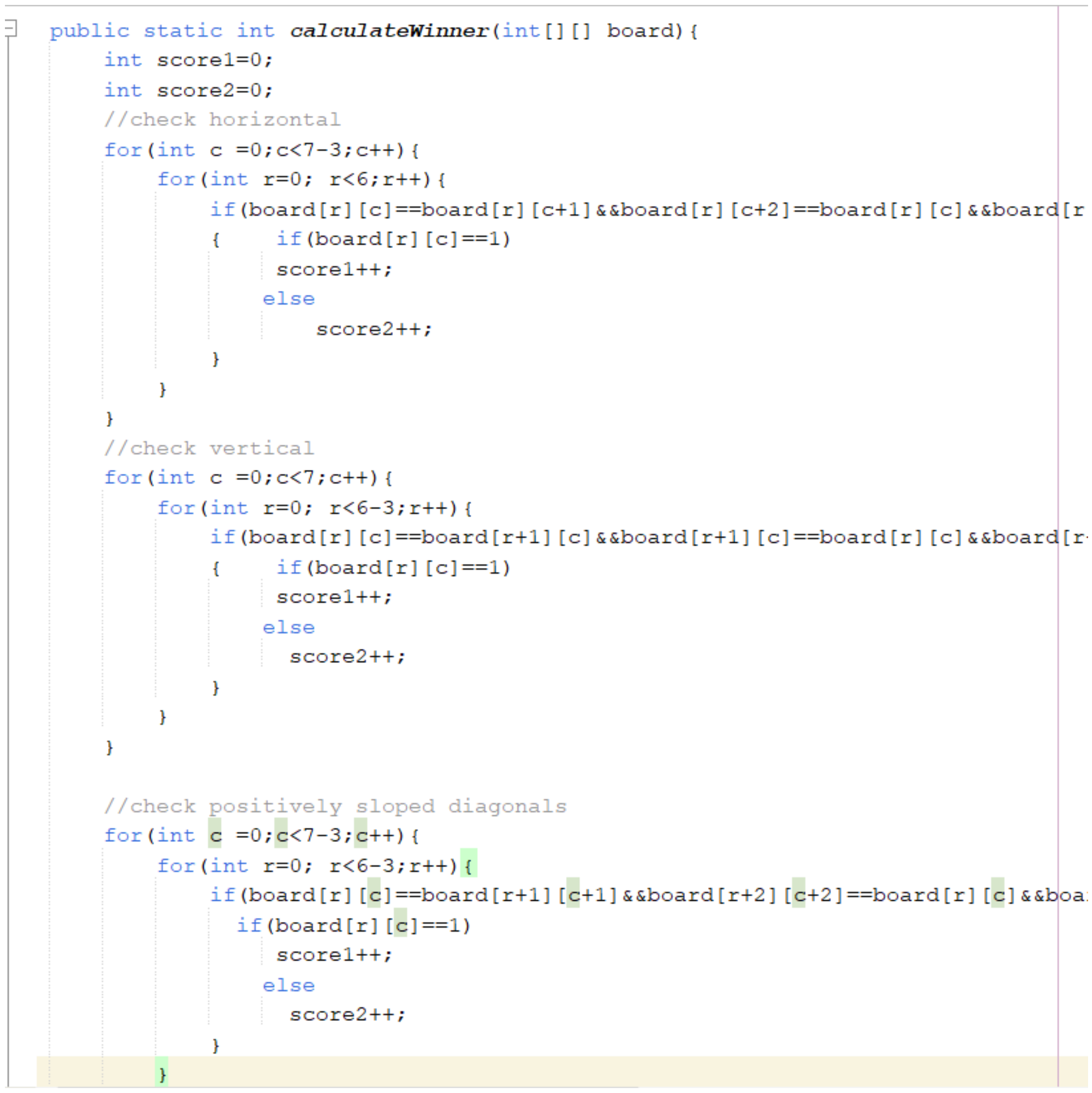
```
public static int[][] createBoard(){
    int[][] board = new int[6][7];
    return board;
}
```

To check if the board is filled hence the game will be over we used isFilled(int [][] board) method

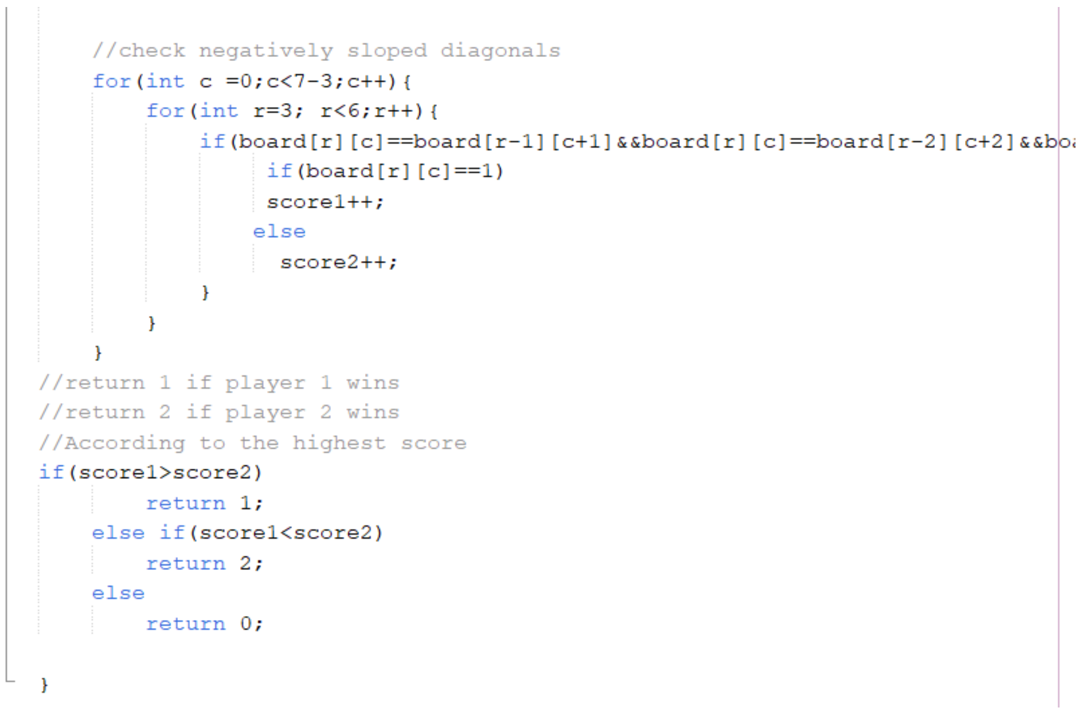


```
public static boolean isFilled(int[][] board){
    for(int i=0;i<board.length;i++){
        for(int j=0;j<board[i].length;j++){
         if(board[i][j]==0)
            return false;
        }
    }
    return true;
}
```

To calculate who the winner is we use calculateWinner(int[][] board) to count the rows, columns and diagonals of 4s for each player, it all adds up to a score and the winner is the one with the bigger score



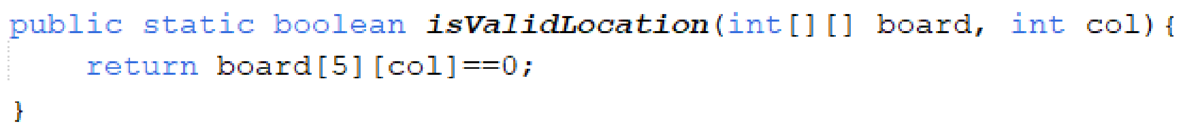
```
public static int calculateWinner(int[][] board){
    int score1=0;
    int score2=0;
    //check horizontal
    for(int c =0;c<7-3;c++){
        for(int r=0; r<6;r++){
            if(board[r][c]==board[r][c+1]&&board[r][c+2]==board[r][c]&&board[r
            {    if(board[r][c]==1)
                 score1++;
                else
                    score2++;
            }
        }
    }
    //check vertical
    for(int c =0;c<7;c++){
        for(int r=0; r<6-3;r++){
            if(board[r][c]==board[r+1][c]&&board[r+1][c]==board[r][c]&&board[r
            {    if(board[r][c]==1)
                 score1++;
                else
                  score2++;
            }
        }
    }

    //check positively sloped diagonals
    for(int c =0;c<7-3;c++){
        for(int r=0; r<6-3;r++){
            if(board[r][c]==board[r+1][c+1]&&board[r+2][c+2]==board[r][c]&&boa
              if(board[r][c]==1)
                 score1++;
                else
                  score2++;
            }
        }
```



```
    //check negatively sloped diagonals
    for(int c =0;c<7-3;c++){
        for(int r=3; r<6;r++){
            if(board[r][c]==board[r-1][c+1]&&board[r][c]==board[r-2][c+2]&&boa
                if(board[r][c]==1)
                score1++;
               else
                 score2++;
            }
        }
    }
//return 1 if player 1 wins
//return 2 if player 2 wins
//According to the highest score
if(score1>score2)
        return 1;
    else if(score1<score2)
        return 2;
    else
        return 0;

}
```

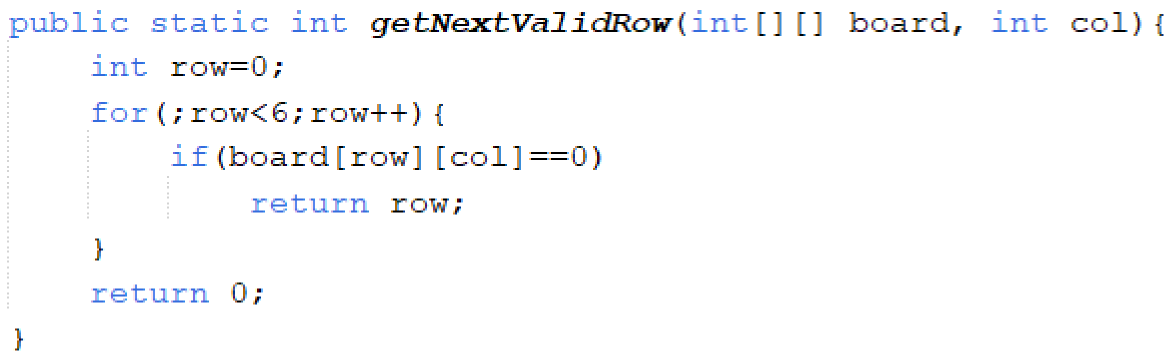
Playing:

We need to check if the column is filled or not so we used the method isValidLocation(int[][] board, int col) : if the last element in the column is not zero then the column is filled



```
public static boolean isValidLocation(int[][] board, int col){
    return board[5][col]==0;
}
```

We also need to see in which row we will put the piece so used the method getNextValidRow(int[][] board, int col)

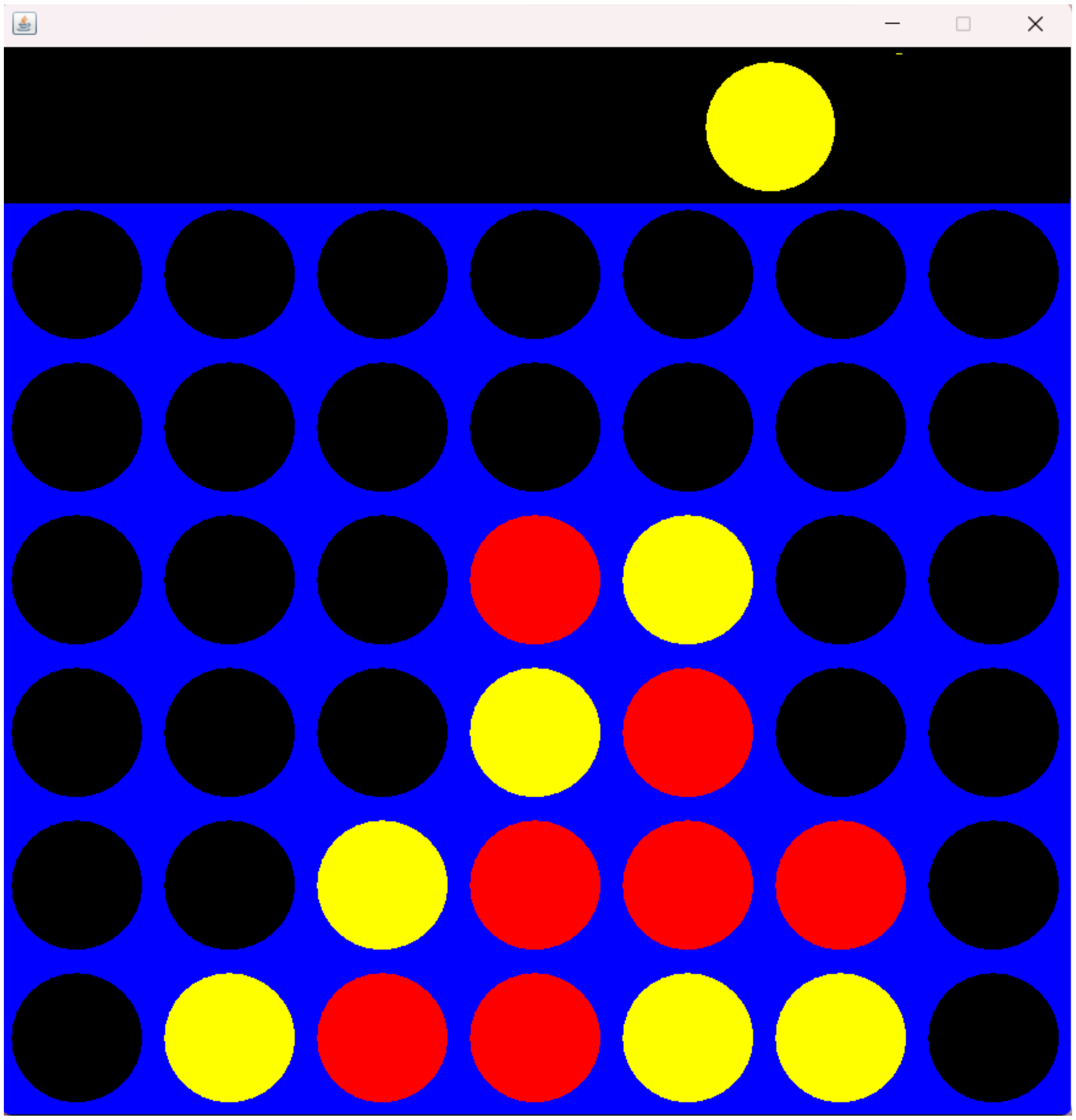


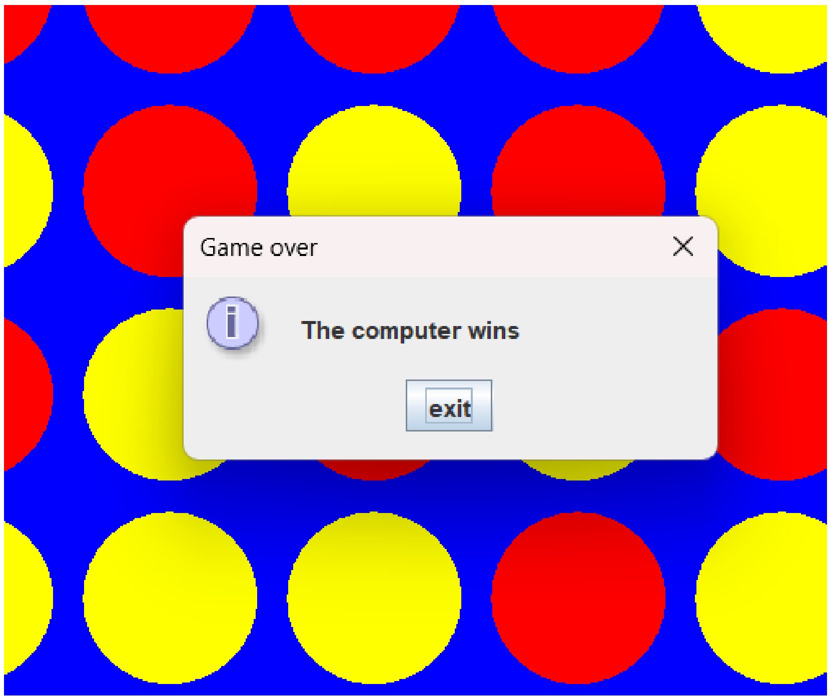
```
public static int getNextValidRow(int[][] board, int col){
    int row=0;
    for(;row<6;row++){
        if(board[row][col]==0)
            return row;
    }
    return 0;
}
```

When we have the row and column where we are going to drop the piece we actually put it using the method dropPiece(int[][] board, int row,int col,int piece)

```
//piece= either the number or the color of the player
public static void dropPiece(int[][] board, int row,int col,int piece){
    board[row][col]=piece;
}
```

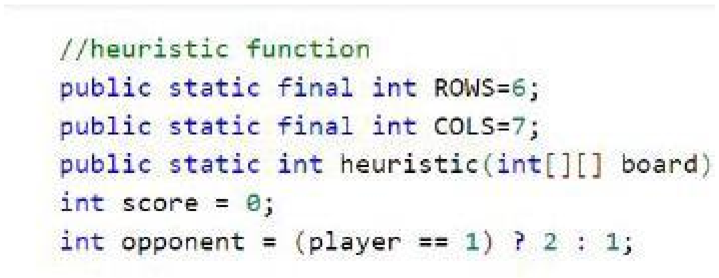
*the AI starts playing when the mouse enters the screen (so the AI plays first)
*The human plays on the click of the mouse and the AI drops its piece right after
*we used java.swing to make our GUI of the game





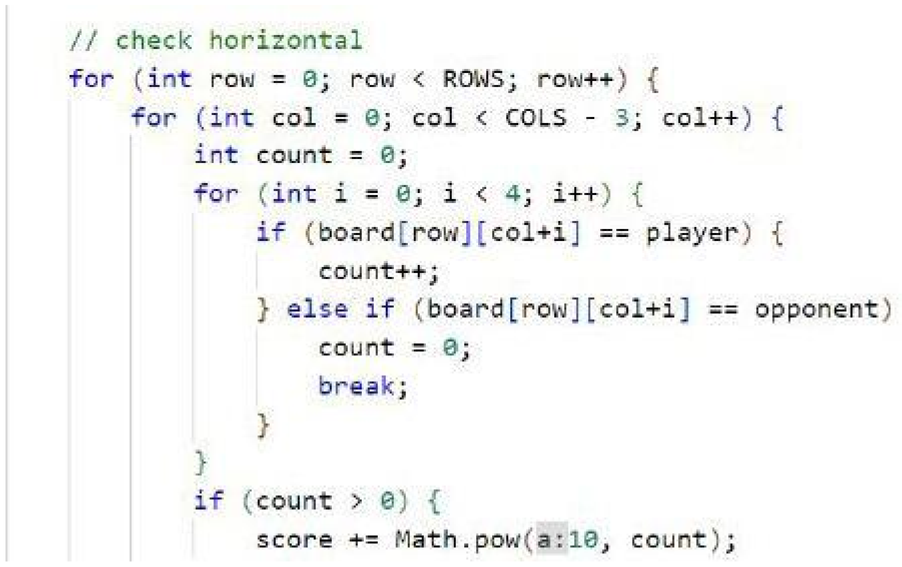
During this game the computer won by scoring 7 4s in a row against the human who only scored 6 4s in a row during the whole game

II- Heuristic:



```
//heuristic function
public static final int ROWS=6;
public static final int COLS=7;
public static int heuristic(int[][] board)
int score = 0;
int opponent = (player == 1) ? 2 : 1;
```

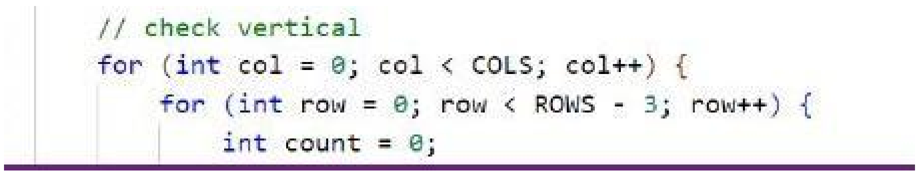
- We first initialize the rows and column
- Then we create the method **heuristic** that takes our 2D array (board)
- We initialize **score** that will be calculated based on the score of the board for each player
- If the current player is 1 then the other player is 2 and vice versa



```
// check horizontal
for (int row = 0; row < ROWS; row++) {
    for (int col = 0; col < COLS - 3; col++) {
        int count = 0;
        for (int i = 0; i < 4; i++) {
            if (board[row][col+i] == player) {
                count++;
            } else if (board[row][col+i] == opponent)
                count = 0;
                break;
            }
        }
        if (count > 0) {
            score += Math.pow(a:10, count);
```

Here we check for the 4 pieces horizontally

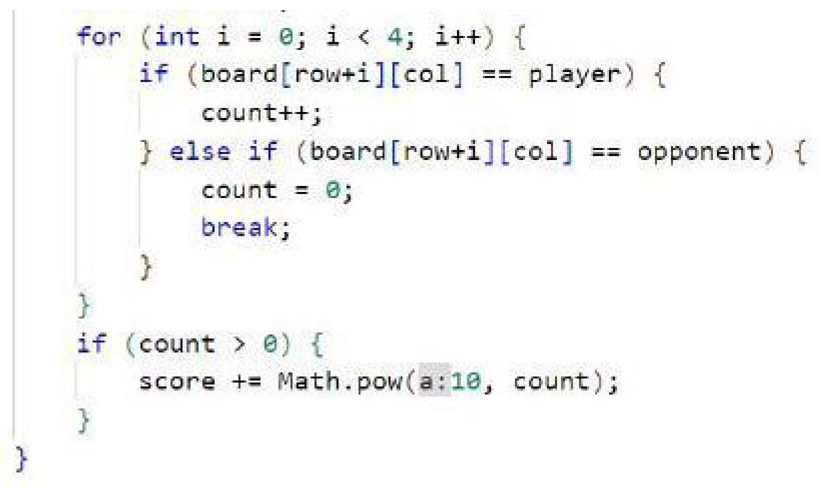
- We loop over the rows and columns ,but up to the 4th column only
- We initialize count to use it in incrementing the score
- If the board has a piece of the player we are checking ,we increment count
- If the board has a piece of the opponent we stop and get out of loop
- Then we calculate the score of the player



```
// check vertical
for (int col = 0; col < COLS; col++) {
    for (int row = 0; row < ROWS - 3; row++) {
        int count = 0;
```

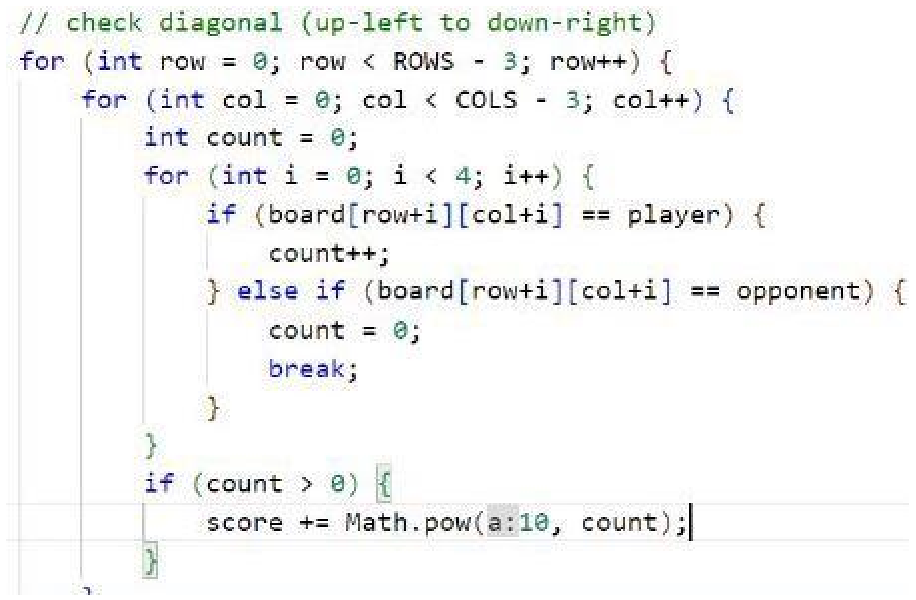
Here we check for the 4 pieces vertically

- We loop over the rows and columns ,but up to the 4th row
- only
- We initialize count to use it in incrementing the score



```
    for (int i = 0; i < 4; i++) {
        if (board[row+i][col] == player) {
            count++;
        } else if (board[row+i][col] == opponent) {
            count = 0;
            break;
        }
    }
    if (count > 0) {
        score += Math.pow(a:10, count);
    }
}
```

- If the board has a piece of the player we are checking ,we increment count
- If the board has a piece of the opponent we stop and get out of loop
- Then we calculate the score of the player

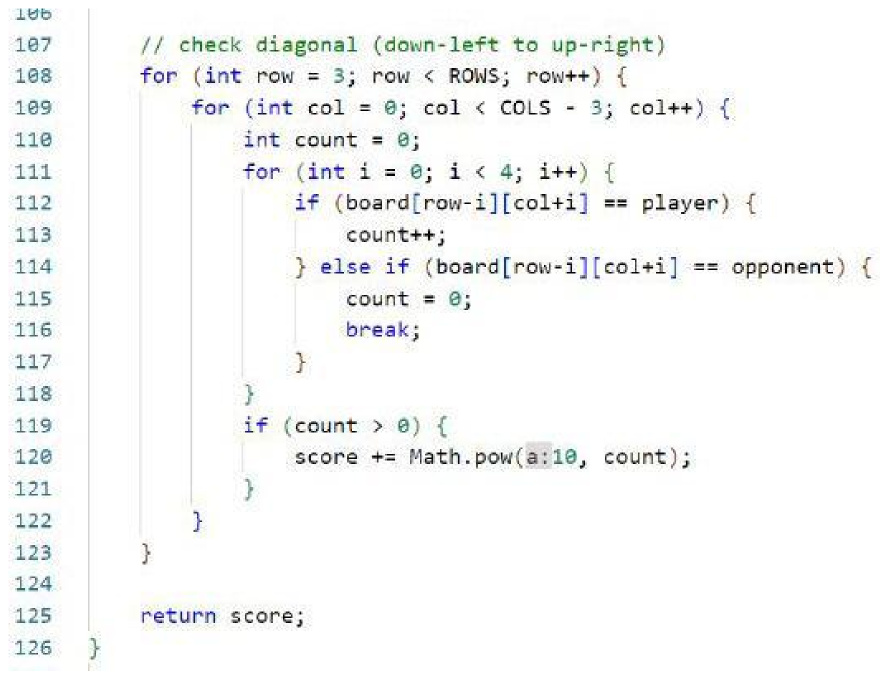


```
// check diagonal (up-left to down-right)
for (int row = 0; row < ROWS - 3; row++) {
    for (int col = 0; col < COLS - 3; col++) {
        int count = 0;
        for (int i = 0; i < 4; i++) {
            if (board[row+i][col+i] == player) {
                count++;
            } else if (board[row+i][col+i] == opponent) {
                count = 0;
                break;
            }
        }
        if (count > 0) {
            score += Math.pow(a:10, count);
        }
```

Here we check for the 4 pieces in a diagonal (up-left to down right)

- We loop over the rows and columns ,but up to the 4th row and 4th column only
- We initialize count to use it in incrementing the score
- If the board has a piece of the player we are checking ,we increment count

- If the board has a piece of the opponent we stop and get out of loop
- Then we calculate the score of the player

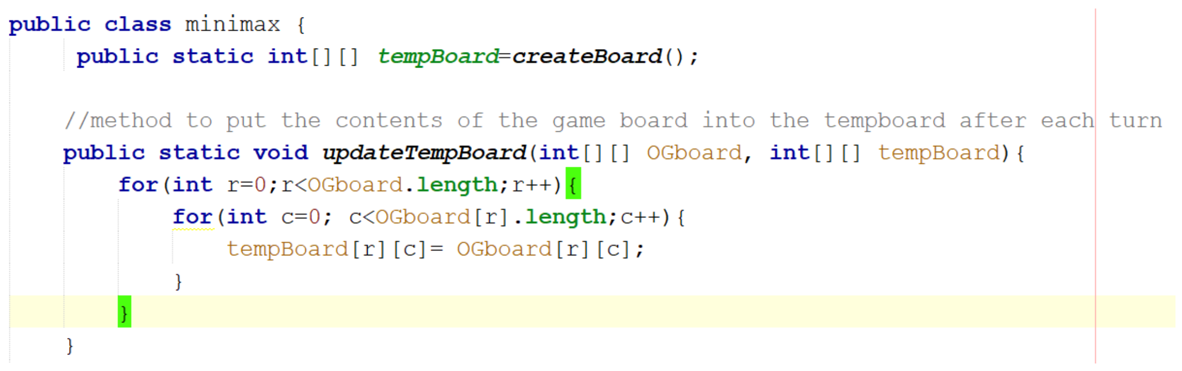


```
    // check diagonal (down-left to up-right)
    for (int row = 3; row < ROWS; row++) {
        for (int col = 0; col < COLS - 3; col++) {
            int count = 0;
            for (int i = 0; i < 4; i++) {
                if (board[row-i][col+i] == player) {
                    count++;
                } else if (board[row-i][col+i] == opponent) {
                    count = 0;
                    break;
                }
            }
            if (count > 0) {
                score += Math.pow(a:10, count);
            }
        }
    }

    return score;
}
```

Here we check for the 4 pieces in a diagonal (down-left to upright)

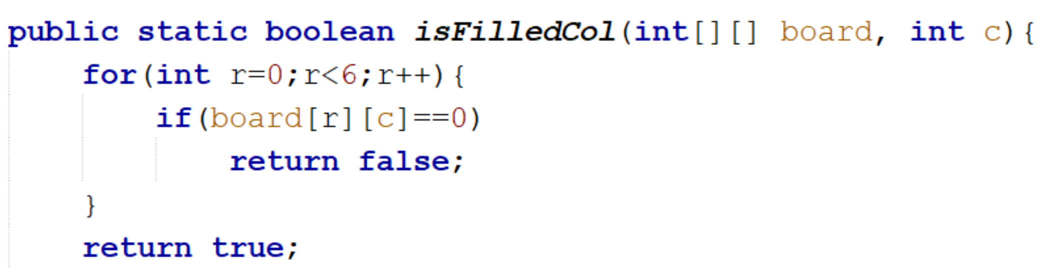
- We loop over the rows and columns ,but up to 4th column only
- We initialize count to use it in incrementing the score
- If the board has a piece of the player we are checking ,we increment count
- If the board has a piece of the opponent we stop and get out of loop
- Then we calculate the score of the player

III- Minimax:



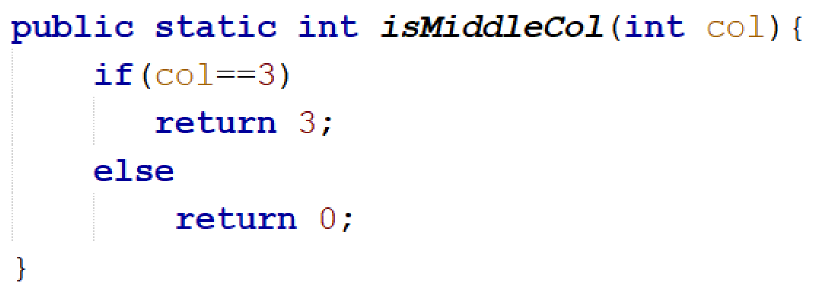
```
public class minimax {
    public static int[][] tempBoard=createBoard();

   //method to put the contents of the game board into the tempboard after each turn
   public static void updateTempBoard(int[][] OGboard, int[][] tempBoard){
       for(int r=0;r<OGboard.length;r++){
           for(int c=0; c<OGboard[r].length;c++){
               tempBoard[r][c]= OGboard[r][c];
           }
       }
   }
```

- we start by creating a temporary board by calling createBoard method
- updateTempBoard is a method that takes the original and temporary boards as parameters and updates the board state by equalizing each element in temp state with the og one



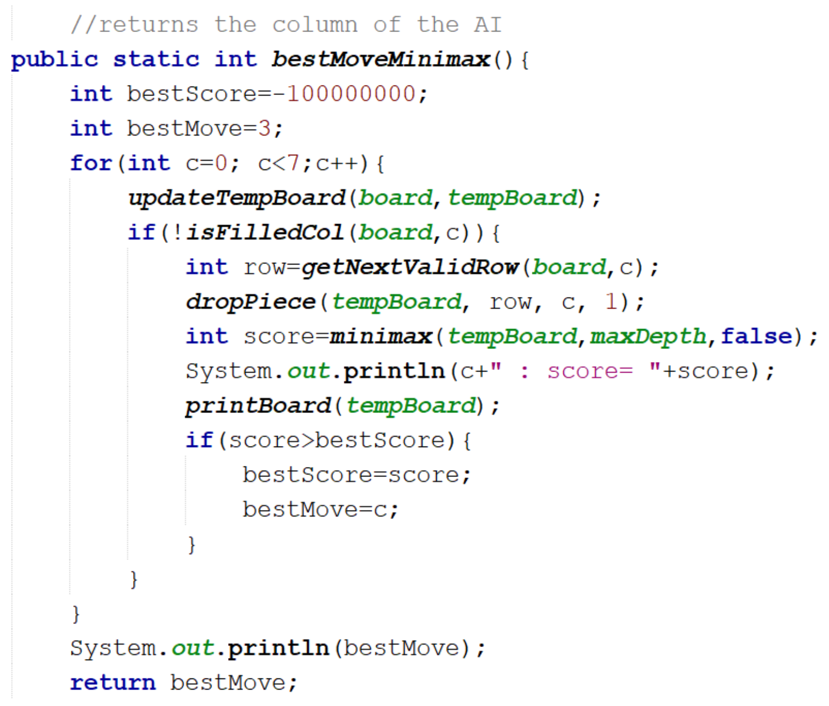
```
public static boolean isFilledCol(int[][] board, int c){
    for(int r=0;r<6;r++){
        if(board[r][c]==0)
            return false;
    }
    return true;
```

- isFilledCol method checks if a given column is filled with pieces or not
- it takes the current state and column number and returns boolean value



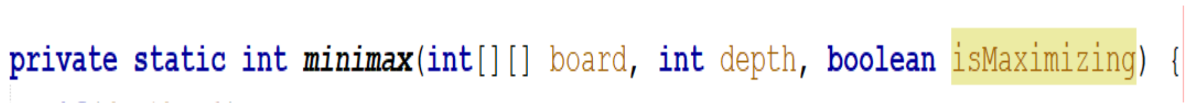
```
public static int isMiddleCol(int col){
    if(col==3)
        return 3;
    else
        return 0;
}
```

this method checks if the given column is the middle column of board or not



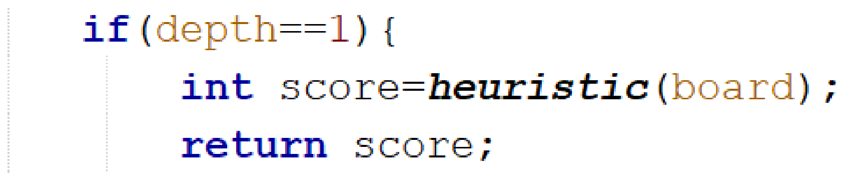
```java
    //returns the column of the AI
public static int bestMoveMinimax(){
    int bestScore=-100000000;
    int bestMove=3;
    for(int c=0; c<7;c++){
        updateTempBoard(board,tempBoard);
        if(!isFilledCol(board,c)){
            int row=getNextValidRow(board,c);
            dropPiece(tempBoard, row, c, 1);
            int score=minimax(tempBoard,maxDepth,false);
            System.out.println(c+" : score= "+score);
            printBoard(tempBoard);
            if(score>bestScore){
                bestScore=score;
                bestMove=c;
            }
        }
    }
    System.out.println(bestMove);
    return bestMove;
```

- this method is where we calculate which is the best move to go through which is a column number
- starts by initializing bestScore with -ve infinity(smallest value) and bestMove with 3(middle col)
- loop iterates through every col in board and updates the board each time and if that col is not filled it gets the valid row for the corresponding column
- then a piece is dropped in the temporary board(ai player) on that col and score equals the value that minimax method returns when called by tempBoard
- if that score is greater than bestScore, it becomes the bestScore and the bestMove becomes the column we're in in loop



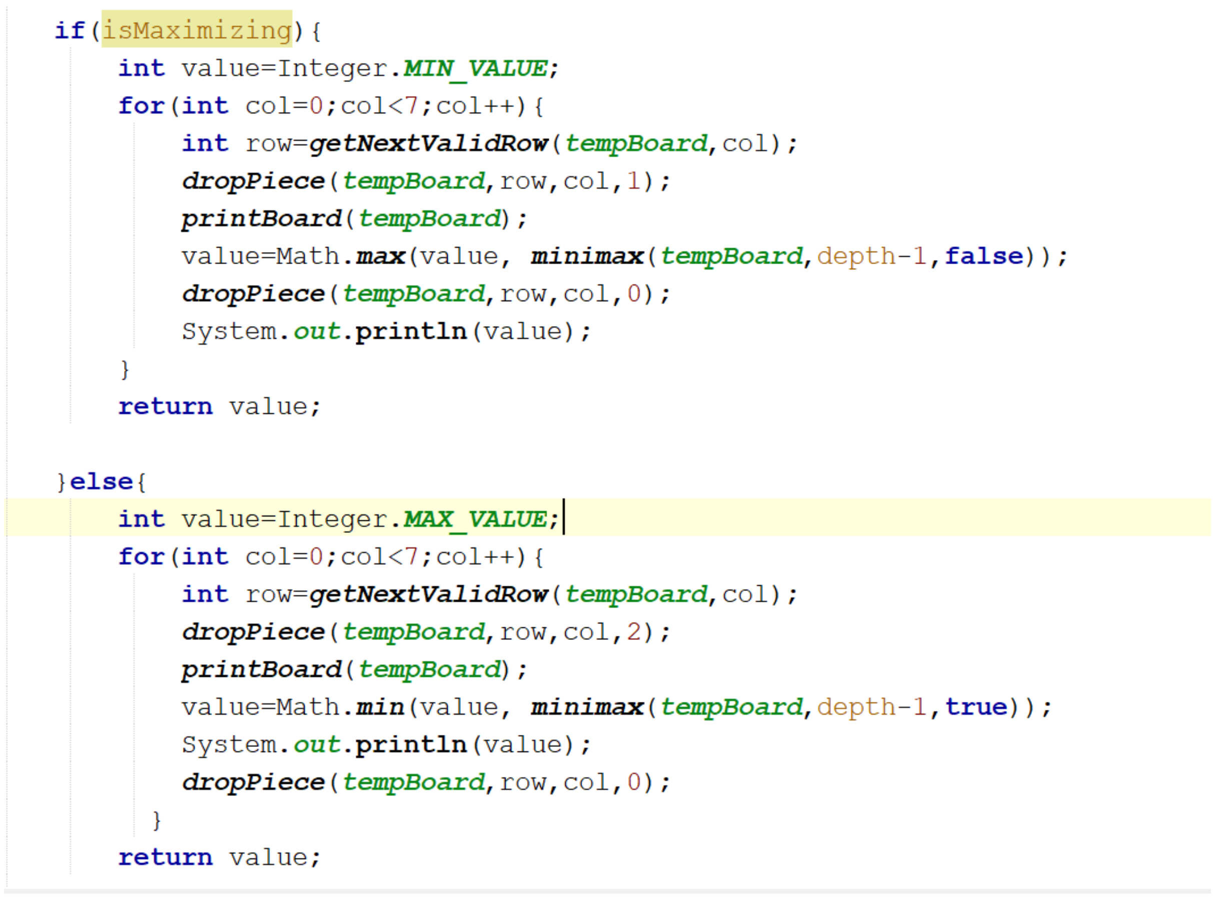
```java
private static int minimax(int[][] board, int depth, boolean isMaximizing) {
```

- this is the method that implements the minimax algorithm
- it takes the current state of game, depth, and condition (max or min) as parameters and returns value of move



```
if(depth==1){
    int score=heuristic(board);
    return score;
```

starts by an if conditional that equalizes score by return value of heuristic method if depth = 1

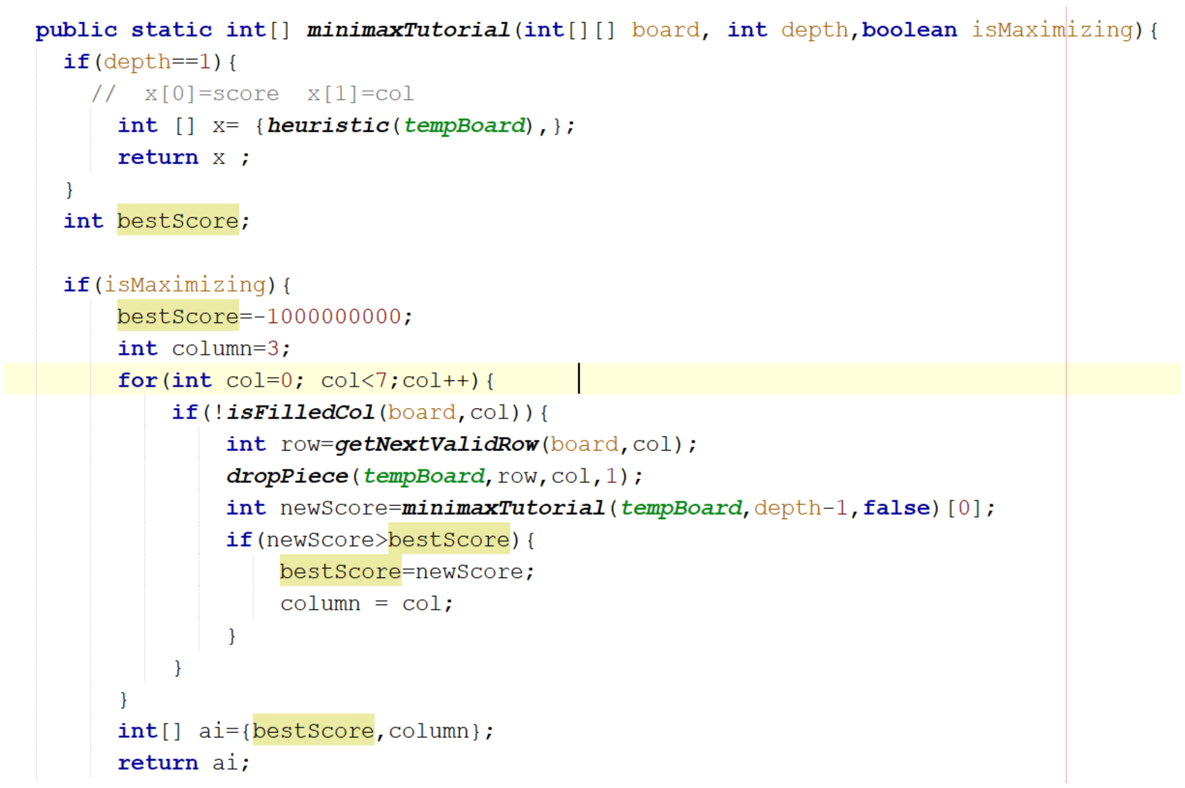


```
if(isMaximizing){
    int value=Integer.MIN_VALUE;
    for(int col=0;col<7;col++){
        int row=getNextValidRow(tempBoard,col);
        dropPiece(tempBoard,row,col,1);
        printBoard(tempBoard);
        value=Math.max(value, minimax(tempBoard,depth-1,false));
        dropPiece(tempBoard,row,col,0);
        System.out.println(value);
    }
    return value;

}else{
    int value=Integer.MAX_VALUE;
    for(int col=0;col<7;col++){
        int row=getNextValidRow(tempBoard,col);
        dropPiece(tempBoard,row,col,2);
        printBoard(tempBoard);
        value=Math.min(value, minimax(tempBoard,depth-1,true));
        System.out.println(value);
        dropPiece(tempBoard,row,col,0);
     }
    return value;
```

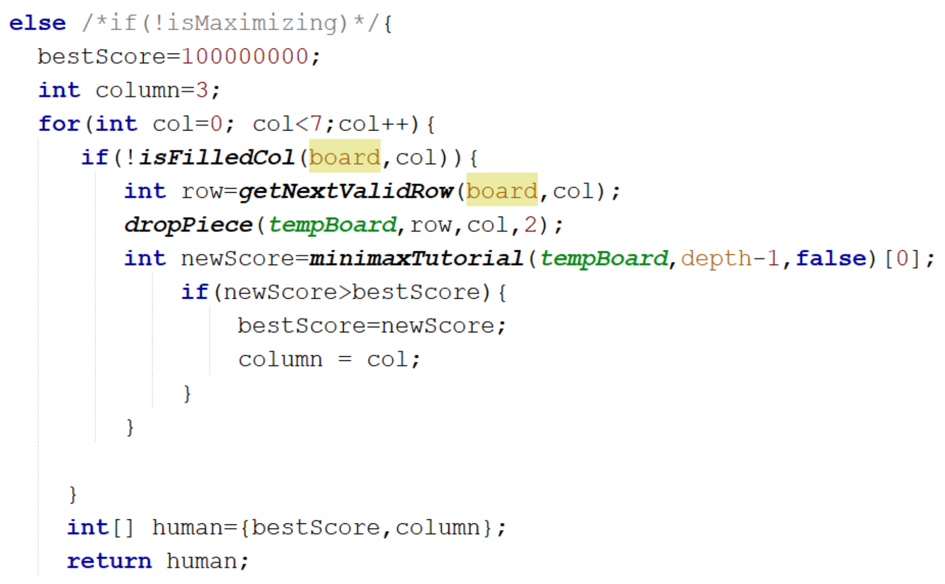
- there's another if conditional such that if max parameter is true (player is maximizing), it implements its body
- begins with initializing value with -ve infinity as the lowest value possible
- then there's a for loop that iterates through all columns on the board
- getNextValidRow method is called to find the next available row for the given column
- then dropPiece method is called to drop a piece on the temporary board at the calculated row and column for the current player

- the next line calculates the maximum value by calling the minimax method recursively with the temporary board, decreasing the depth by 1, and switching the player

- when dropPiece method is called again, it removes the dropped piece from the temporary board to undo the previous move
- lastly returns the maximum value found during the loop
- if max parameter is false (player is minimizing), it implements the opposite
- begins with initializing value with +ve infinity as the highest value possible
- then it does the same as the previous if condition body, finally returns the minimum value found during the loop



```
public static int[] minimaxTutorial(int[][] board, int depth,boolean isMaximizing){
  if(depth==1){
  //  x[0]=score  x[1]=col
    int [] x= {heuristic(tempBoard),};
    return x ;
  }
  int bestScore;

  if(isMaximizing){
    bestScore=-1000000000;
    int column=3;
    for(int col=0; col<7;col++){
        if(!isFilledCol(board,col)){
            int row=getNextValidRow(board,col);
            dropPiece(tempBoard,row,col,1);
            int newScore=minimaxTutorial(tempBoard,depth-1,false)[0];
            if(newScore>bestScore){
                bestScore=newScore;
                column = col;
            }
        }
    }
    int[] ai={bestScore,column};
    return ai;
```



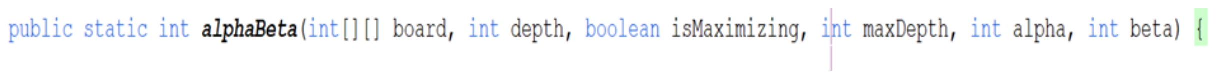
```
else /*if(!isMaximizing)*/{
  bestScore=100000000;
  int column=3;
  for(int col=0; col<7;col++){
      if(!isFilledCol(board,col)){
          int row=getNextValidRow(board,col);
          dropPiece(tempBoard,row,col,2);
          int newScore=minimaxTutorial(tempBoard,depth-1,false)[0];
              if(newScore>bestScore){
                  bestScore=newScore;
                  column = col;
              }
          }

    }
    int[] human={bestScore,column};
    return human;
```

- this method checks if newScore(of less depth and new col) is greater than current score and makes it the current score and its col the current column
- incase of maximizing, it returns bestScore of ai player and its col
- incase of minimizing, it returns bestScore of human player and its column

Different depth comparison

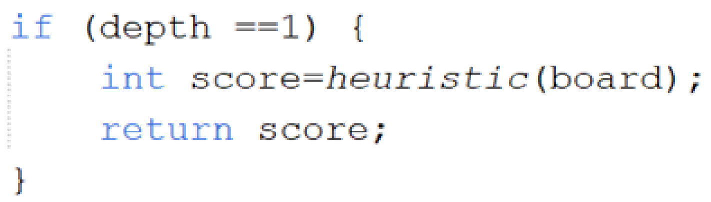
At depth 1~3
The computer plays fairly quickly
At depth 4
The computer starts to slow down but no more than 1~3 seconds
At depth 5 and higher
The computer significantly slows down due to the generation of so many states

IV- Alpha-Beta pruning



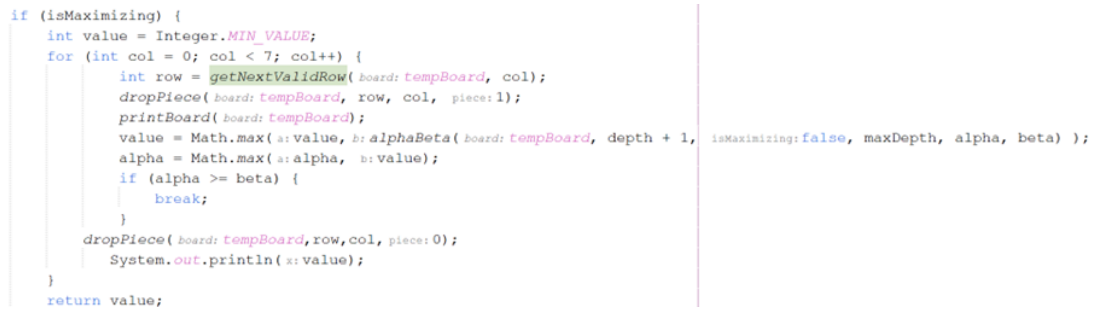
```
public static int alphaBeta(int[][] board, int depth, boolean isMaximizing, int maxDepth, int alpha, int beta) {
```

- This is the method signature for the alpha-beta pruning function. It takes in the current game state as a 2D integer array board, the current depth of the search depth, a boolean value isMaximizing indicating if the current player is maximizing or not, the maximum depth to search maxDepth, and the alpha and beta values for pruning.



```
if (depth ==1) {
    int score=heuristic(board);
    return score;
}
```

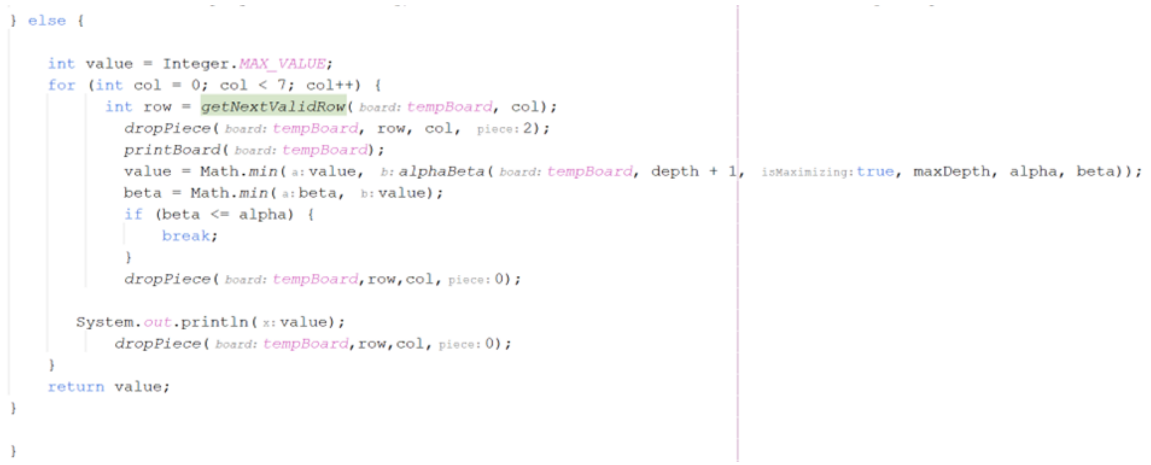
- This checks if the maximum search depth has been reached. If it has, it evaluates the heuristic value of the current game state using the heuristic function and returns it.



```
if (isMaximizing) {
    int value = Integer.MIN_VALUE;
    for (int col = 0; col < 7; col++) {
            int row = getNextValidRow( board: tempBoard, col);
            dropPiece( board: tempBoard, row, col,  piece: 1);
            printBoard( board: tempBoard);
            value = Math.max( a: value, b: alphaBeta( board: tempBoard, depth + 1,  isMaximizing: false, maxDepth, alpha, beta) );
            alpha = Math.max( a: alpha,  b: value);
            if (alpha >= beta) {
                break;
            }
        dropPiece( board: tempBoard,row,col, piece: 0);
          System.out.println( x: value);
    }
    return value;
```

- If the current player is maximizing, the function initializes the best score to the lowest possible value using Integer.MIN_VALUE.
- It then loops through each column of the board and drops a piece in the first available row in that column using getNextValidRow and dropPiece.
- The function then calls itself recursively with the updated game state, increasing the depth by 1 and changing the isMaximizing parameter to false.
- It updates the best score by taking the maximum of the current best score and the value returned by the recursive call. It also updates the alpha value and checks if it's greater than or equal to the beta value to perform pruning.

- Finally, it removes the dropped piece and prints the value before moving to the next column.



```
} else {

    int value = Integer.MAX_VALUE;
    for (int col = 0; col < 7; col++) {
          int row = getNextValidRow( board: tempBoard, col);
           dropPiece( board: tempBoard, row, col,  piece: 2);
           printBoard( board: tempBoard);
           value = Math.min( a: value,  b: alphaBeta( board: tempBoard, depth + 1,  isMaximizing: true, maxDepth, alpha, beta));
           beta = Math.min( a: beta,  b: value);
           if (beta <= alpha) {
               break;
           }
           dropPiece( board: tempBoard,row,col, piece: 0);

       System.out.println( x: value);
          dropPiece( board: tempBoard,row,col, piece: 0);
    }
    return value;
}

}
```

- If the current player is minimizing, the function initializes the best score to the highest possible value using Integer.MAX_VALUE.
- It then loops through each column of the board and drops a piece in the first available row in that column using getNextValidRow and dropPiece.
- The function then calls itself recursively with the updated game state, increasing the depth by 1 and changing the isMaximizing parameter to true.
- It updates the best score by taking the minimum of the current best score and the value returned by the recursive call.
- It also updates the beta value and checks if it's less than or equal to the alpha value to perform pruning.
- Finally, it removes the dropped piece and prints the value before moving to the next column.

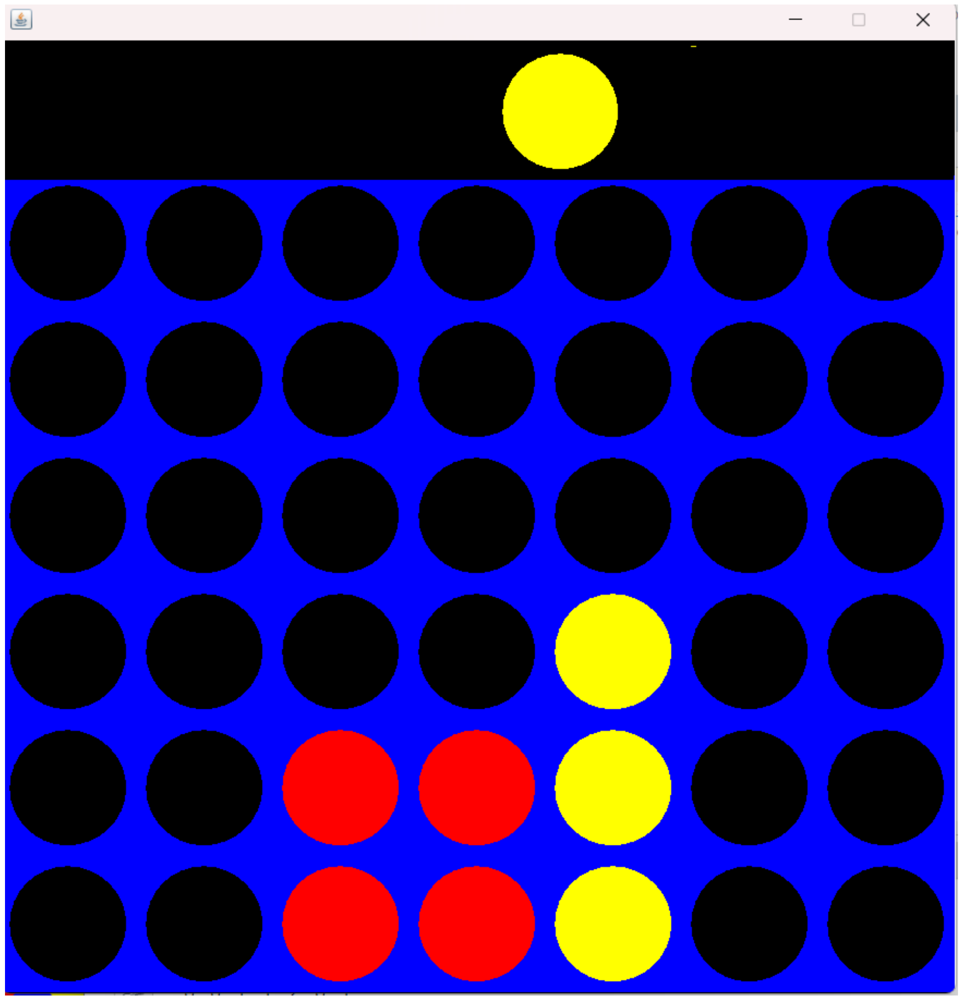
The difference between the minimax algorithm with and without the alpha-beta pruning

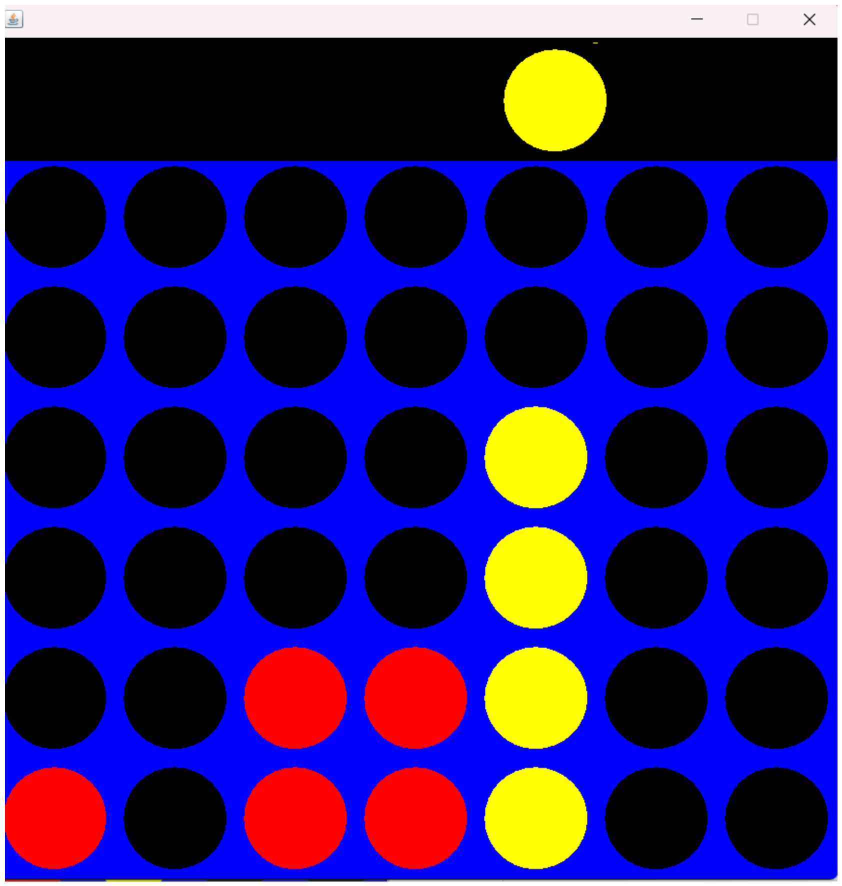
The speed difference between the two algorithms is obvious at depth 5: where the minimax **without the alpha beta pruning** slows down takes around ***15.3 seconds*** to drop the piece
Whereas the minimax **with alpha beta pruning** with the same sequence of move takes around ***1.8 seconds*** which is a huge speed difference between the two

V- Sample runs:

At depth 1:

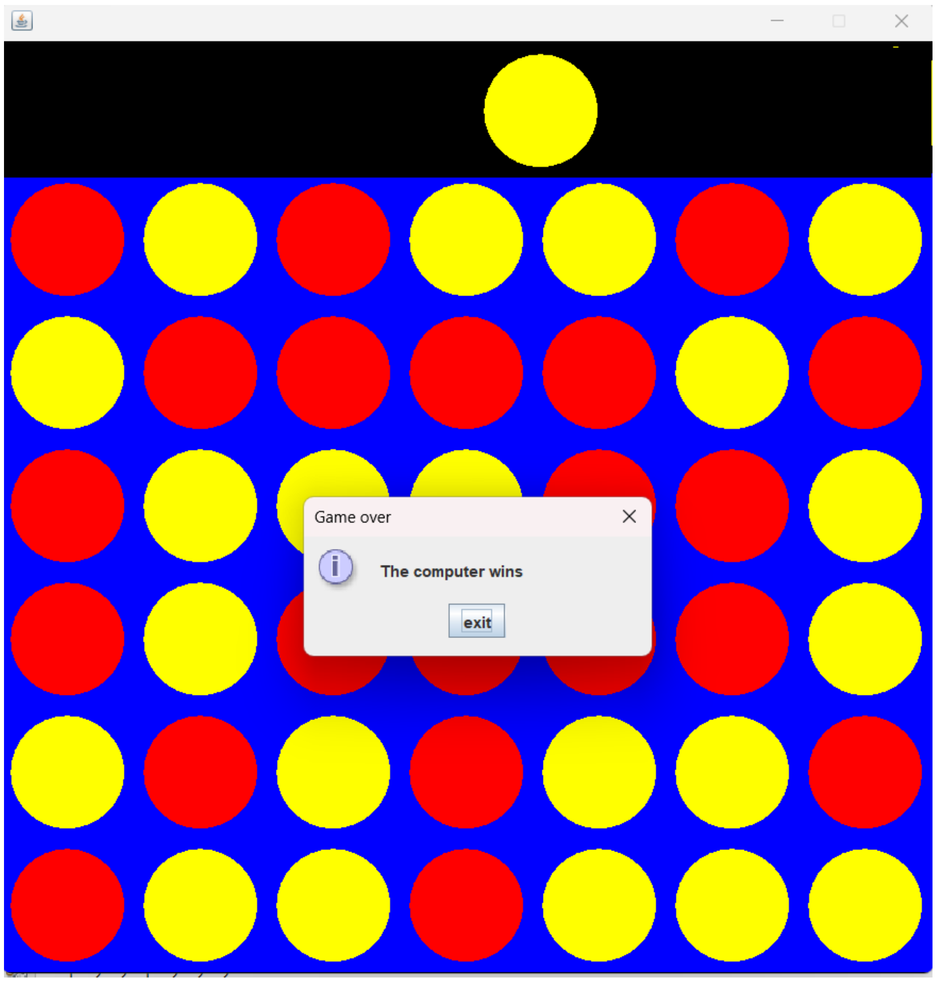
The computer plays fairly smart and can win me but it plays to win (meaning that sometimes it will let the human connect 4 in a row)





At depth 2:

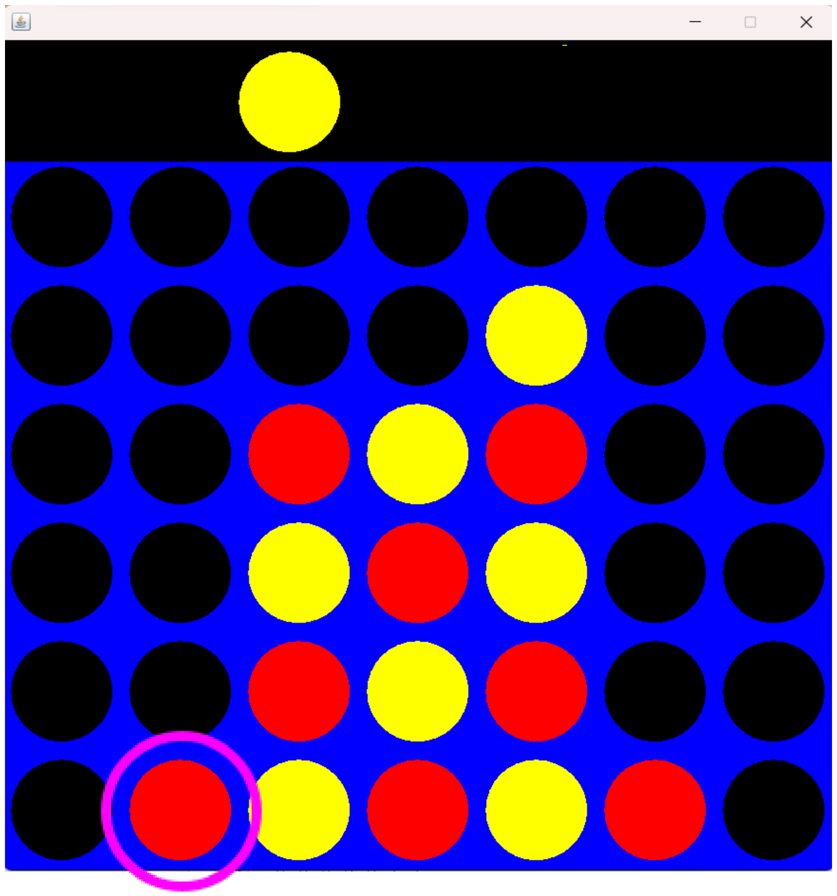
This is the depth where the computer is unbeatable it never lets me win and plays extremely hard


Game over

The computer wins

At depth 4:

At 3 the difference between and 2 isn't clear it's still playing quite optimally but at 4 due to the abundance of the states the computer starts making suboptimal decisions and sometimes the human can win the whole game



Here the computer put its piece in the second column - which let him score 4 in a row - but allowed me to score a diagonal (so it's a suboptimal decision)