Task 1 XY-Coordinate

**Checkboundaries**

**spec:**

the method is supposed to return true when the current position is valid and false if it is invalid.

**Design:**

It was designed to compare the current x and y values with the xSize and ySize values(these will be the same value in Tic tac toe as the board is quadratic), and if they are inside the range 0(inclusive)-size(exclusive) it is valid.

**Implementation:**

A single if statement that checks whether x and y are within the above range and returns the Boolean of the statement.

**Shift**

**Spec:**

the method is supposed to return new x and y values that are shifted by the corresponding dx and dy values.

**Design:**

the method was altered to also take a Coordinate as argument, this was to make sure that the correct coordinate is shifted.

**Implementation:**

the method takes the x and y values from the coordinate start and adds the dx and dy respectively to create a new XYCoordinate which it then returns.

Task 2 TTT-Board

**isFree**

**Spec:**

the purpose of this method is to check if the current position on the board is free.

**Design:**

Check the Board at the given position and if the value is 0 return true, else return false.

**Implementation:**

Implemented as designed.

**getPlayer**

Spec:

A method that is to return the number of the player that has placed a marker on a given position on the board, and return 0 if no player has a marker on that position

Design:

It returns the value of the board at the position it is passed.

Implementation:

Implemented as designed.

**addMove**

Spec:

This method the current players number to the position that was clicked on th board.

Design:

It takes a position and a int as arguments and places the int on the position on the board.

Implementation:

a try catch block was implemented to catch errors. The method tries to assign the currentplayer at the board position it was passed. if an error occurs it checks whether it was caused by an invalid board position using checkBoundaries or if the playernumber is invalid using the size of the board(as the size of board is created by taking the number of players and adding 1). An extra error log is added in case neither of the above are the reason for the Illegal argument error. This is done in an attempt to fail loudly as to help with debugging.

**checkFull**

Spec:

the method is supposed to check if the board is full, I.E. no valid move can be taken.

Design:

Use 2 for loops to run though each position on the board and if it encounters a board position that returns a value of 0, the method returns false.

Implementation:

starts out with a Boolean variable set equal to true, then using 2 for loops runs through each position on the board and checks if the value at each position is equal to 0. If it is the Boolean is set equal to false, and at the end of the method the variable is returned. In this case if the method just once encounters a board position that is equal to 0 the method returns false.

Checkfull is used in checkResult and here the if statement is changed to an else if statement. Before in cases where a player won on the last possible move the game would show the result screen with “this was a draw” printed.

**checkwinning**

Spec:

This method is used for checking if a player has won the game.

Design:

the Plan was to create start positions that could be used by the checkSequence method to check all rows, columns and diagonals on the board.

Implementation:

the method uses 3 different double for loops to iterate through all possible starting positions used for checkSequence. First block checks rows and columns, here the important bit is that for columns is that we don’t need start positions in the 2 right most columns, as there aren’t enough tiles for a full sequence of 3 markers. The same goes for rows but here we don’t need start positions in the bottom 2 rows.

The other 2 double for loops create start positions for checking the diagonals. These are divided into 2 groups, one for diagonals going up and one for diagonals going down. Starting positions for the first group is a combination of the ones for rows and columns. We don’t need positions in neither the bottom 2 rows or the rightmost 2 columns. For the second group we don’t need the top 2 rows and the rightmost 2 columns

The method then calls checksequence with the generated start positions and dx and dy corresponding to either rows(dx=1 ,dy=0), columns(dx= ,dy=1) etc.

if the returned value from checksequence isn’t 0 it means that a player has won and checkWinning returns the int.

**checksequence**

Spec:

The method takes 3 arguments, a Coordinate, dx, and dy, and uses this to determine if 3 tiles in a row, column or diagonal contain the same number aside from 0.

Design:

It was designed to check the board at the starting position and then shifting the position using the shift method, and then checking if this new position on the board contains the same value. it will do this twice to check a sequence of 3. if it finds 3 of the same number in a sequence then it returns that vale, else it returns 0.

Implementation:

takes 3 arguments, a Coordinate and 2 integers, dx and dy. The method creates an int variable, called checkValue, equal to the board at the Coordinate and if it’s equal to 0, returns it. Then using a for loop to run the shift method twice with the given dx and dy arguments. After each loop it checks whether the board at this new position is equal to checkValue. If the are not equal the method returns 0 and if they are equal the second loop occurs and a new check happens. Again if they aren’t equal o is returned, but if they are equal it means 3 tiles in a sequence are the same and that player has won, resulting in checkValue being returned.

Task 3 testing the Game –

**Checksequence method:**

an array out of bounds error occured in the checksequence method, this was caused by updating the start coordinate. so first run through x,y was 0,0 then 0,1 then 0,3 and should have been 0,2 in the last, but start pos was 0,1 and i shifted with 0,2 = 0,3. was discovered using log.d. fixed by deleting the start update, but could also be fixed by only shifting by 0,1 both times.

Later changed to only shifting by 1 each time and updating the starting pos.

**Error in the checkwinning method:**

There is a bug with checking when a player has won the game. at first a winner was declared by setting a marker in one of the corners in the 2-player game, and board bigger than this not all sequences of 3 in a row results in a win.

error was discovered in the checkWinning method where looking at the winner variable happened outside the for loops which resulted in only the last sequence beeing checked

The problem was fixed by placing the if statement that checked the winner variable inside the for loops so now all sequences the for loop checked was looked at, instead of only the last.

**CheckResult method:**

There was 2 if statements, one for checking for a winner and one for checking if the board was full. This resulted in the case that where a player won on the turn that the board was also full the message would state that the game was a draw(I assume it for a split second actually said “player x wins”, but that it was overwritten with the draw message). Changing the full board statement to an else if means it only gets run if there is no winner.

AI.java:

First issue was trying to get a board passed from TTTGame to use for evaluating the best move for the ai, but board in TTTGame isn’t a int[][] so from TTTGame i couldnt pass the board. Instead had the addMove method in TTTBoard call the aiAddMove method and pass the current board along.

in the early version we wanted to populate an aiBoard to be used during recursion to check more than 1 move ahead, but this was discovered to be unneeded as just the normal rules with a few exceptions results in a win or draw in all tested cases when only the current board is considered.

Value bug: having a lossValue of 80 resulted in a few cases where a winning move and a blocking move had the same value(winning move = 100 – 10 for a oppmarker, and blocking move being 80+10 for own marker).

Fixed by reducing lossValue to 79

TODO: PICTURE HERE

Human start 1,1: if the human player starts and places a token in 1,1 I.E. the middle. All other free positions where evaluated to -10 and as bestValue starts of at 0, none of the free positions returned a higher value than 0 a bestMove was never set. **Result= nullpointer on getX() in addMove();**

Fix: only in this case would the bestMoveList be empty so we made an if statement that checked for this and when true would populate the list with positions corresponding to the 4 corners.

**ConnectFour:**

Bug: nomatter where I clicked player 1 won. No mistakes found in checkwinning and checksequence. The bug was in how the marker was placed. For loop ran through Y-values from high to low(bottom to top) and the first time an empty cell was encountered placed a marker there. It was missing a “break;” statement so it ran through all y values even after placing the first marker. This lead to having 4 in a row after the first click and player 1 winning. Missing break found by always return 0 in checkwinning and seeing the entire column filling with the current players number instead of only the lowest tile.

Task 4 AI

**Specification**

the AI is supposed to be an unbeatable opponent, as in the AI will always win or draw the game, but never loose

**Design**

The AI was planned to analyze the board and give each position values based on how good likely it was that it would lead to a win or at least a draw. It was to do this by iterating through the board and for each position look at the row, column and diagonals, where applicable, and add values together and then choose the position that had the highest value. At first the idea was that recursion should be used to get the AI to look several moves ahead to but initial testing revealed that already while just looking at the next move there was only a few situations where it didn’t guarantee at least a draw.

It was only written to work for a 2 player 3x3 board but could be extended to work on both larger boards and more players, of course here we might not be able to guarantee at least a draw.

**implementation**

The Values used to analyze the board were to start with 100 for a tile that would lead to a win, 90 for a tile that would lead to a loss, and 10 and -10 for tiles containing the AI’s mark and the players mark respectively. Later the lossValue was changed to 79 to avoid edge cases where the AI could win by placing a mark in one spot but choose to instead block the player in another spot.

There are 4 methods in the AI.java, 4 checkmethods, moveEvaluate, GenerateMove and aiAddMove.

The 4 checkmethods check each row, column and if applicable one or both diagonals and returns a value based on what each tile in that sequence contained, I.E. if it had 2 AI marks it would return 100.

MoveEvaluate simply adds the 4 check methods together and returns it.

GenerateMove runs, using 2 for loops, through all 9 possible positions on the board, and if that position is free(I.E. the board a that position contains a 0) calls moveEvaluate. The int returned here is assigned to the variable currentValue, which is then checked vs the variable bestValue. If both have the same value a new XYCoordinate is created using the current x and y values in the loop. This Coordinate is then added to bestMoveList. If currentValue is greater than bestValue bestMoveList is cleared and then a new Coordinate is added. This is done so that if several possible moves have the same value all are saved in the list and a the end a random move from the list is selected.

aiAddMove simply returns the GenerateMove method, and is called from the addMove in TTTBoard.

There are three exceptions to the standard rules that the AI follow.

The first is that when the AI goes first it will always place a marker in the middle, as this is where there most possible paths that lead to a win for the AI.

The second is if the human player starts and places a marker in the middle, then all other 8 positions will return a value of -10 and therefore currentValue will never be greater than or equal to bestValue. This results in no move being added to the bestMoveList and a NullPointerException. The solution we initially chose was to set bestValue to -10 and call the generateMove method again, but this resulted in a possible loss if the AI chose to place it’s marker in one of the middle tiles. Instead we not just populate the bestMoveList with Coordinates for the 4 corners.

The last is an extension of the second exception. In the case where the human player made a move in 1,1, the AI would the mark a corner, and if the player then marked the opposite corner the result would be that the AI would force a loss on itself with the standard values (unless the player made a mistake). The solution was to implement a check that would look for this case and then alter bestMoveList to contain 2 different Coordinates.

***TODO picture here!***

added stuff for AI

added button for starting game vs ai. it sets numplayers equal to 1 instead of taking value from the numberpicker.

added button that makes the ai take it's move aswell as a different title depending on whether the AI or human starts.

added method to game.java and TTTgame.java called playerTurn. this method is used for controlling when the AI move button and the board can be pressed to make sure a button only responds when it should.

**in TTTGame.java**

TTTgame constructor has been changed to randomize starting player when playing vs the ai(could also be extended to the normal gamemode) and addmove have been altered to handle playing vs the AI. I.E. if player 1 took his turn it sets currentplayer = 7 aka the AI, and vice versa.

**in TTTBoard.java**

TTTBoard contructor was changed to create a 3x3 board when playing vs AI

addMove now, if currentplayer is 7, calls aiAddMove to find the pos at which to add a marker.

Conclusion

**TTT:**

the game works as intended, but there are certain areas where it could be improved.

the gameresultscreen would be better if it showed the final board instead of just a screen stating who won TODO fix this!

our checkWinning could be altered to, only checking for sequences from the position where the currentplayer just made a move, and not checking before it is possible for a player to win, I.E. after turn 4 in a 2player game turn 6 in a 3player game(number of turns equalling 2\*the number of players), instead of checking the entire board after every turn. in programs that require more processing power this could be of significance but in a game this simple it would make very little difference.

**AI**:

the AI works as intended, for all cases we've tested the AI either wins or draws but since there are 9!(equals 362.880) ways the board can be filled(minus some because they are the same as other cases with the board rotated 90 degrees etc.) we havent tested them all.

An update to the AI could be changing it to work with a larger board and/or more players. To get it to work with a larger board but still 2 players it would only need a change to the for loops that check through the board for start positions and values.

making it work with more players and larger board would require a few changes to how the turns are "calculated" as well as the above changes.