FollowChess

Overview

We created a chess program in which we can load and print the chessboard, apply moves, track the captured pieces of both opponents, and check if either player's king is in check or not.

The problem is mainly divided into four parts: loading the input board, applying moves alternatively, checking the check and mate situations for both kings, and determining if a particular piece can move from its current position.

The problem has a fixed structure for inputs, which is a combination of characters and integers. Columns are represented with characters, and rows are represented with integers. It is also crucial to ensure that the first move must be made by a white piece. White pieces are represented by 'r', 'n', 'b', 'q', 'k', 'p', while black pieces are represented by 'R', 'N', 'B', 'Q', 'K', 'P', and '.' (dot) is used to denote an empty cell.

Files and external data

The implementation contains two classes: A1 and FollowChess.

A1 - It is the main class as it contains the main method.
FollowChess - It contains all the logic for this assignment. There are six main methods that we have to implement as per the assignment.

loadBoard() - This function takes a buffered reader as input and creates a 2D list of character data type. The function further relies on three other functions: create2DArray, checkValid2DArray, and checkValidPieces.

- **create2DArray()** - This function creates a 2D array.
- **checkValidPieces()** - While creating the array, it ensures that the character present at each cell is valid. Valid pieces' characters must belong to black and white pieces.
- **checkValid2Darray()** - This function checks that all the rows have the same size.

printBoard() - Print the board using the board created by the loadBoard() function and output that board to the incoming output stream.

applyMoveSequence() - It is responsible for applying moves to the chessboard. It takes input as a buffered reader in which each line contains four things: start column, start row, end column, and end row. This function further uses two other functions, inCheck, and isMovePossible.

- **isMovePossible()** - Depending on the piece and player, it calls functions and returns true if the move is possible; otherwise, it returns false. Functions called from this method are canPawnMove, canKnightMove, canPieceMoveHoriVerti, and canPieceMoveDiagonally.
    - ➔ **canPawnMove()** - Checks if the pawn can move one step ahead or move diagonally to capture the opponent's piece.
    - ➔ **canKnightMove()** - Checks if the knight can move in any of the eight predefined positions.
    - ➔ **canPieceMoveHoriVerti()** - Checks if the piece, like a rook, queen, or king, can move at least one step or more in a horizontal or vertical direction.
    - ➔ **canPieceMoveDiagonally()** - Checks if the piece, like a bishop, queen, or king, can move at least one step or more diagonally in any of the four directions.

inCheck() - It ensures that the player, for which the move is attempted, doesn't put their king in check or checkmate by any of the opponent's pieces.

captureOrder() - It returns the list showing the order in which the player captured the opponent's pieces.

pieceCanMove() - It uses the same four functions as isMovePossible() but only checks if at least one move is possible for the given piece in any direction.

isWhiteorBlackPiece() - This is a general function, mostly used by every method. It only checks if the given piece is white or black. It returns true if it is a white piece; otherwise, it returns false, indicating it is a black piece.

Data structures and their relations to each other

Chess Board - I used the simplest data structure, a 2D list, to load the board. It is easy to retrieve data because it is an index-based data structure.

Capture Order - I used a list of characters to store the pieces captured by that player as it maintains the same order as we inserted.

Opponent Piece - I used a Map data structure to store the opponent's piece and its row and column while checking for check and mate for that player. I chose this because it is very convenient to store this type of data without creating our own class; instead, we can make a list of that class.

Assumptions

- When creating an object of followChess, it only contains one board at a time.
- The input stream in the loadBoard() method and the input string in pieceCanMove() have the same structure as given in the assignment.
- The number of columns must not exceed 26, and columns must be letters, while rows must be integers.
- Special cases such as pawns being able to move 2 squares as their first move, pawn promotion, en-passant captures, and castling are not considered.

Choices

- Make a copy of the board, then apply moves to that copy. If all moves are successfully performed, the temporary board is copied to the final board.
- The sequence contains 10 moves, and if it fails at the 7th move, the capture order for the preceding 7 moves will not be saved.

Key algorithms and design elements

I strive to design my four main functions as generalized as possible so that every method can use them. These functions are canPawnMove(), canKnightMove(), canPieceMoveDiagonally, and canPieceMoveHoriVerti.

Step 1: First, we select the direction in which the piece wants to move. We can calculate it using the given parameters.
Step 2: We check if, on the very first move in that direction, we go out of the chessboard; if so, we return false.
Step 3: Based on that direction, we move to the corresponding block of code using a switch statement. In that block, we use a for loop to move in that direction until we reach our destination (row and column). If, in between, we encounter any piece, either white or black, we return false.
Step 4: In the for loop, we stop one step before the end row and end column. After coming out of that switch part, we check if moving one step more in the direction will be equal to our end row and end column. If not, we return false.
Step 5: After that, we check if the piece moves one step ahead, and at that position, if a '.' (dot) is present, we return true. If not, we check if an opponent's piece is present. If yes, then it captures that piece and also records that captured piece into its respective captureOrder list.

applyMoveSequence()

Step 1: Retrieve startRow, startCol, endRow, and endCol from the current line of the input stream.

Step 2: Ensure that all four variables are valid coordinates on the chessboard. If it is the first time applyMoveSequence is called on this board, verify that the first move is made by a white piece.

Step 3: Check that the player attempting the move does not have their king in a checkmate situation. If incCheck is true, verify that if the player's chosen piece is the king, it is allowed; otherwise, return false.

Step 4: Use the isMovePossible() function to check if the move is possible.

Step 5: If step 4 returns true, repeat step 1; otherwise, return false.

Step 6: After successfully applying all moves, copy the temporary board to the permanent board, and update the captureOrder.

inCheck()

Step 1: Scan the entire board and create a map of opponent characters along with their respective rows and columns.

Step 2: Then, for each iteration through the entire map, call the appropriate function from the above four methods based on the piece type.

Step 3: Return the result obtained from that function.

pieceCanMove(): This function mainly utilizes the same four functions mentioned above, according to the piece, and returns the value provided by that method.

Limitations:

- This game does not stop or indicate the player's victory when the king is captured by any player.
- The program will not suggest any move that increases the chances of winning. Furthermore, it also does not suggest any move to remove the check and mate of the king.
- Special moves, such as the pawn's first move taking two steps, are not applicable to this program.