import java.util.ArrayList;

import java.util.Arrays;

public abstract class Piece{ // ENTIRE class coded by Aman Modi

private int row;

private int col;

private String color;

private String pieceType;

public Piece[][] array = Board.board1;

private ArrayList<String> capturedWhite ;

private ArrayList<String> capturedBlack;

private boolean enPassant;

private boolean moved;

private int p1Score;

private int p2Score;

private Multiplayer multi = new Multiplayer();;

public Piece(int r, int c, String s, String pt){

pieceType = pt;

row = r;

col = c;

color = s;

enPassant = false;

moved = false;

p1Score = 0;

p2Score = 0;

capturedWhite = new ArrayList<String>();

capturedBlack = new ArrayList<String>();

}

public boolean getMoved(){

return moved;

}

public void setMoved(boolean m){

moved = m;

}

public boolean getEnPassant(){

return enPassant;

}

public void setEnPassant(boolean ep){

enPassant = ep;

}

public int getRow(){

return row;

}

public void setRow(int r){

row = r;

}

public int getCol(){

return col;

}

public void setCol(int c){

col = c;

}

public String getColor(){

return color;

}

public void setColor(String c){

color = c;

}

public String getPieceType(){

return pieceType;

}

public void setPieceType(String pt){

pieceType = pt;

}

public void changePosition(int r, int c)

{

int newr = getRow();

int newc = getCol();

array[r][c] = array[getRow()][getCol()];

if(newr%2==0 && newc%2==0 || newr%2 != 0 && newc%2!=0)

array[newr][newc] = new DefaultWhite(newr,newc,"#");

else

array[newr][newc] = new DefaultBlack(newr,newc,"/");

setRow(r);

setCol(c);

}

public boolean capturePiece(int r, int c) //Start of Ayaan

{

if(!(array[r][c].getPieceType().equals("#") || array[r][c].getPieceType().equals("/")) && !array[r][c].getColor().equals(getColor())){

score(array[r][c]);

if(array[r][c].getColor().equals("w") || array[r][c].getColor().equals("W"))

capturedBlack.add(array[r][c].getPieceType());

else if(array[r][c].getColor().equals("b") || array[r][c].getColor().equals("B"))

capturedWhite.add(array[r][c].getPieceType());

return true;

}

else

return false;

} //End of Ayaan

public boolean checkBounds(int r, int c)

{

if((r<8 && r>=0) && (c<8 && c>=0))

return true;

else

return false;

}

public void printCaptured()

{

String output = "";

String whitePlayer = "";

String blackPlayer = "";

if(Multiplayer.player1Color.equals("white")){

whitePlayer = Multiplayer.player1 ;

blackPlayer = Multiplayer.player2;

}else{

whitePlayer = Multiplayer.player1;

blackPlayer = Multiplayer.player2;

}

output += "\n\nBlack Pieces captured by the White player " + whitePlayer + " :\n";

for(int i = 0; i < capturedBlack.size(); i++)//adds white pieces

{

output += capturedBlack.get(i) + " ";

}

System.out.println(output);

output = "\n\n";

output += "\n\nWhite Pieces captured by the Black player " + blackPlayer + " :\n";

for(int i = 0; i < capturedWhite.size(); i++)//adds black pieces

{

output += capturedWhite.get(i) + " ";

}

System.out.println(output);

}

public boolean checkOpen(int r, int c)

{

if(array[r][c].getPieceType().equals("#") || array[r][c].getPieceType().equals("/"))

return true;

else

return false;

}

public abstract boolean movePiece(int r, int c, String co);

// The following three methods checks for check.

//The following three methods are used similarly in stalemate and checkmate

// ---------------------------------------------------------------------------------------------------------------------------

// METHODS for Check

public boolean checkForCheck(int r, int c, String co){

int origRow = getRow();

int origCol = getCol();

String color1 = array[r][c].getColor();

String pt1 = array[r][c].getPieceType();

boolean moved1 = array[r][c].getMoved();

boolean status = false;

changePosition(r, c);

status = inCheck();

array[origRow][origCol] = array[r][c];

setRow(origRow);

setCol(origCol);

switch(pt1)//sets everything back to the way it was

{

case "/": array[r][c] = new DefaultBlack(r,c,"/");break;

case "#": array[r][c] = new DefaultWhite(r,c,"#");break;

case "b": array[r][c] = new Bishop(r,c,color1,pt1);break;

case "B": array[r][c] = new Bishop(r,c,color1,pt1);break;

case "P": array[r][c] = new Pawn(r,c,color1,pt1);break;

case "p": array[r][c] = new Pawn(r,c,color1,pt1);break;

case "Q": array[r][c] = new Queen(r,c,color1,pt1);break;

case "q": array[r][c] = new Queen(r,c,color1,pt1);break;

case "R": array[r][c] = new Rook(r,c,color1,pt1);break;

case "r": array[r][c] = new Rook(r,c,color1,pt1);break;

case "n": array[r][c] = new Knight(r,c,color1,pt1);break;

case "N": array[r][c] = new Knight(r,c,color1,pt1);break;

case "K": array[r][c] = new King(r,c,color1,pt1);break;

case "k": array[r][c] = new King(r,c,color1,pt1);break;

}

array[r][c].setMoved(moved1);

return status;

}

public boolean inCheck(){

//This method gets three parameters that indicate the place the user wants to move

// To test this temporary variables have been created

// (For CS1 coders ArrayList is basically an array except the fact that you can modify the components in the array)

// In this case it will store all the boolean values of test cases

ArrayList<Boolean> status = new ArrayList<Boolean>();

// Loops through the entire board

// Using this algorithm this function will check threats from the opponent color

for(int i = 0; i < array.length; i++){

for(int j = 0; j < array[i].length; j++){

//If the chess piece is not an empty spot and does not equal the same color (so we are checking opponent colors)

if(!array[i][j].getColor().equals(getColor())

&& !(array[i][j].getPieceType().equals("#"))

&& !(array[i][j].getPieceType().equals("/"))){

// Add the value returned from find position to the array

//(Follow the algorithm to findPos)

status.add(findPos(array[i][j]));

}

}

}

for(boolean x : status)

if(x == true){

return true;

}

return false;

}

public boolean findPos(Piece piece){

// This method will check if the piece attacks the opponent king

boolean checkPos = false;

ArrayList<Boolean> areas = new ArrayList<Boolean>();

String type = piece.getPieceType().toLowerCase();

if(type.equals("q")){ // If it s a queen

// Calls find king and if it attacks the opponent king it adds true

// If it doesn't add false

areas.add(findKing(piece, piece.getRow(), piece.getCol(), -1, 0)); //Checks if the queen attacks king upward

areas.add(findKing(piece, piece.getRow(), piece.getCol(), 1, 0)); //Downward

areas.add(findKing(piece, piece.getRow(), piece.getCol(), 0, -1)); //Left

areas.add(findKing(piece, piece.getRow(), piece.getCol(), 0, 1)); //Right

areas.add(findKing(piece, piece.getRow(), piece.getCol(), -1, 1)); //The following check diagnols

areas.add(findKing(piece, piece.getRow(), piece.getCol(), 1, -1));

areas.add(findKing(piece, piece.getRow(), piece.getCol(), -1, -1));

areas.add(findKing(piece, piece.getRow(), piece.getCol(), 1, 1));

} else if(type.equals("r")){

//The first four conditionals of queen (up/down, left/right)

areas.add(findKing(piece, piece.getRow(), piece.getCol(), -1, 0));

areas.add(findKing(piece, piece.getRow(), piece.getCol(), 1, 0));

areas.add(findKing(piece, piece.getRow(), piece.getCol(), 0, -1));

areas.add(findKing(piece, piece.getRow(), piece.getCol(), 0, 1));

} else if(type.equals("b")){

//The last four conditionals of queen (diagnol)

areas.add(findKing(piece, piece.getRow(), piece.getCol(), -1, 1));

areas.add(findKing(piece, piece.getRow(), piece.getCol(), 1, -1));

areas.add(findKing(piece, piece.getRow(), piece.getCol(), -1, -1));

areas.add(findKing(piece, piece.getRow(), piece.getCol(), 1, 1));

} else if(type.equals("n")){

// If it is a knight

for(int i = 0; i < 8; i++){

int currentRow = piece.getRow();

int currentCol = piece.getCol();

switch(i){ //Changes the areas knights could attack by adding rows and cols

case(0) : currentRow -= 1; currentCol += 2; break;

case(1) : currentRow -= 2; currentCol += 1; break;

case(2) : currentRow += 2; currentCol -= 1; break;

case(3) : currentRow += 1; currentCol -= 2; break;

case(4) : currentRow -= 1; currentCol -= 2; break;

case(5) : currentRow -= 2; currentCol -= 1; break;

case(6) : currentRow += 2; currentCol += 1; break;

case(7) : currentRow += 1; currentCol += 2; break;

}

// Check if the knight can really attack the position given

if(checkBounds(currentRow, currentCol))

//Check if it can attack the opponent king

if(array[currentRow][currentCol].getPieceType().equals("k") && !(array[currentRow][currentCol].getColor().equals(getColor())))

areas.add(true); // If it does, add true to the array

}

} else if(type.equals("p")){

// Checks the movement for the pawn, and if it attacks the king

for(int i = 0; i < 2; i++){

int currentRow = piece.getRow();

int currentCol = piece.getCol();

if(piece.getColor().equals("b")){

switch(i){

case(0) : currentRow += 1; currentCol -= 1; break;

case(1) : currentRow += 1; currentCol += 1; break;

}

}else{

switch(i){

case(0) : currentRow -= 1; currentCol -= 1; break;

case(1) : currentRow -= 1; currentCol += 1; break;

}

}

if(checkBounds(currentRow, currentCol))

if(array[currentRow][currentCol].getPieceType().equals("k") && !(array[currentRow][currentCol].getColor().equals(getColor())))

areas.add(true);

}

} else if(type.equals("k")){

// Checks if the opponental king attacks the king

for(int i = 0; i < 7; i++){

int currentRow = piece.getRow();

int currentCol = piece.getCol();

switch(i){

case(0) : currentRow += 1; break;

case(1) : currentCol += 1; break;

case(2) : currentRow -= 1; break;

case(3) : currentCol -= 1; break;

case(4) : currentRow -= 1; currentCol -= 1; break;

case(5) : currentRow += 1; currentCol -= 1; break;

case(6) : currentRow -= 1; currentCol += 1; break;

case(7) : currentRow += 1; currentCol += 1; break;

}

if(checkBounds(currentRow, currentCol))

if(array[currentRow][currentCol].getPieceType().equals("k") && !(array[currentRow][currentCol].getColor().equals(getColor())))

areas.add(true);

}

}

// In the arraylist it checks if the piece passed to the method attacks the king.

// If it does, it returns true and says the king is in check

for(boolean x : areas)

if(x == true)

return true;

// If none of them attack the king return false

return false;

}

public boolean findKing( Piece piece, int row, int col, int expectR, int expectC){

// It moves the row and the column based on the direction passed to the method.

// If expect Row is 1 and expect col is 0, it changes row everytime by 1, meaning it is checking downwards.

row += expectR;

col += expectC;

while(row > -1 && row < array.length && col > -1 && col < array.length){

String currentType = array[row][col].getPieceType().toLowerCase();

//If the piece in the direction has other pieces that blocks the pathway, immediately return not attacking the king

if(currentType.equals("r") || currentType.equals("n") ||

currentType.equals("b") || currentType.equals("q") || currentType.equals("p")

|| (currentType.equals("k")&& (array[row][col].getColor().equals(piece.getColor())))){

return false;

}

// If it is attacking the opponental king without any interruptions return true (meaning king is in check)

else if(currentType.equals("k") && !(array[row][col].getColor().equals(piece.getColor())))

return true;

row += expectR;

col += expectC;

}

return false;

// REMEMBER : This method only is applicable BISHOP, QUEEN, AND ROOK as they attack their respective directions and it is easier to check using a loop

}

// ---------------------------------------------------------------------------------------------------------------------------

// METHODS for Stalemate

public boolean checkForStalemate(int row, int col, String co){

//If king is already in check, there is no stalemate

if(checkForCheck(row, col, co))

return false;

ArrayList<Boolean> canMove = new ArrayList<Boolean>();

// Uses the exact algorithm from check for check, but now it checks whether the piece could move or not

for(int i = 0; i < array.length; i++){

//Checks if the opponental piece could move

for(int j = 0; j < array[i].length; j++){

if(!array[i][j].getPieceType().equals("#") ||

!(array[i][j].getPieceType().equals("/")) ||

!(array[i][j].getColor().equals(getColor())))

canMove.add(movementChecker(array[i][j]));

}

}

//IF any pieces could move, return false mentioning that it is NOT A STALEMATE

//If pieces cannot move (meaning everything in the array is false), return true for stalemate

for(boolean x : canMove)

if(x)

return false;

return true;

}

public boolean movementChecker(Piece piece){

String type = piece.getPieceType().toLowerCase();

ArrayList<Boolean> movements = new ArrayList<Boolean>();

if(type.equals("q")){

movements.add(findKing(piece, piece.getRow(), piece.getCol(), -1, 0));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), 1, 0));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), 0, -1));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), 0, 1));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), -1, 1));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), 1, -1));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), -1, -1));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), 1, 1));

} else if(type.equals("r")){

movements.add(findKing(piece, piece.getRow(), piece.getCol(), -1, 0));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), 1, 0));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), 0, -1));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), 0, 1));

} else if(type.equals("b")){

movements.add(findKing(piece, piece.getRow(), piece.getCol(), -1, 1));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), 1, -1));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), -1, -1));

movements.add(findKing(piece, piece.getRow(), piece.getCol(), 1, 1));

} else if(type.equals("n")){

// If it is a knight

for(int i = 0; i < 8; i++){

int currentRow = piece.getRow();

int currentCol = piece.getCol();

switch(i){ //Changes the areas knights could attack by adding rows and cols

case(0) : currentRow -= 1; currentCol += 2; break;

case(1) : currentRow -= 2; currentCol += 1; break;

case(2) : currentRow += 2; currentCol -= 1; break;

case(3) : currentRow += 1; currentCol -= 2; break;

case(4) : currentRow -= 1; currentCol -= 2; break;

case(5) : currentRow -= 2; currentCol -= 1; break;

case(6) : currentRow += 2; currentCol += 1; break;

case(7) : currentRow += 1; currentCol += 2; break;

}

// Check if the knight can really attack the position given

if(checkBounds(currentRow, currentCol))

//Check if it can attack the opponent king

if(array[piece.getRow()][piece.getCol()].movePiece(currentRow, currentCol, piece.getColor()) && !(array[currentRow][currentCol].getColor().equals(getColor()))

&& !array[piece.getRow()][piece.getCol()].checkForCheck(currentRow, currentCol, piece.getColor()))

movements.add(true);

}

} else if(type.equals("p")){

// Checks the movement for the pawn, and if it attacks the king

for(int i = 0; i < 3; i++){

int currentRow = piece.getRow();

int currentCol = piece.getCol();

if(piece.getColor().equals("b")){

switch(i){

case(0) : currentRow += 1; currentCol -= 1; break;

case(1) : currentRow += 1; currentCol += 1; break;

case(2) : currentRow += 1; break;

}

}else{

switch(i){

case(0) : currentRow -= 1; currentCol -= 1; break;

case(1) : currentRow -= 1; currentCol += 1; break;

case(2) : currentRow -= 1; break;

}

}

if(checkBounds(currentRow, currentCol))

if(array[piece.getRow()][piece.getCol()].movePiece(currentRow, currentCol, piece.getColor()) && !(array[currentRow][currentCol].getColor().equals(getColor()))

&& !array[piece.getRow()][piece.getCol()].checkForCheck(currentRow, currentCol, piece.getColor()))

movements.add(true);

}

} else if(type.equals("k")){

// Checks if the opponental king attacks the king

for(int i = 0; i < 7; i++){

int currentRow = piece.getRow();

int currentCol = piece.getCol();

switch(i){

case(0) : currentRow += 1; break;

case(1) : currentCol += 1; break;

case(2) : currentRow -= 1; break;

case(3) : currentCol -= 1; break;

case(4) : currentRow -= 1; currentCol -= 1; break;

case(5) : currentRow += 1; currentCol -= 1; break;

case(6) : currentRow -= 1; currentCol += 1; break;

case(7) : currentRow += 1; currentCol += 1; break;

}

if(checkBounds(currentRow, currentCol))

if(array[piece.getRow()][piece.getCol()].movePiece(currentRow, currentCol, piece.getColor()) && !(array[currentRow][currentCol].getColor().equals(getColor()))

&& !array[piece.getRow()][piece.getCol()].checkForCheck(currentRow, currentCol, piece.getColor()))

movements.add(true);

}

}

for(boolean x : movements)

if(x)

return true;

return false;

}

public boolean checkOpen(Piece piece, int row, int col, int expectedRow, int expectedCol){

row += expectedRow;

col += expectedCol;

ArrayList<Boolean> movements = new ArrayList<Boolean>();

while(row > -1 && row < array.length && col > -1 && col < array.length){

if(piece.movePiece(row, col, piece.getColor()) && !(array[piece.getRow()][piece.getCol()].checkForCheck(row, col, piece.getColor()))){

movements.add(true);

}

else

movements.add(false);

row += expectedRow;

col += expectedCol;

}

for(boolean x : movements)

if(x)

return true;

return false;

}

// -----------------------------------------------------------------------------------------------------------------------------

//Methods for points

public void score(Piece piece) //Jon and Sherwin code

{ //score(board[r][c])

//determine whether player1 is white or black

if(Multiplayer.player1Color.equals("white"))//playerOne is white

{

switch(piece.getPieceType())

{

//Capture White by Player 2

case "P": p2Score++; break;

case "N": p2Score += 3; break;

case "B": p2Score += 3; break;

case "R": p2Score += 5; break;

case "Q": p2Score += 9; break;

//Capture Black by Player 1

case "p": p1Score++; break;

case "n": p1Score += 3; break;

case "b": p1Score += 3; break;

case "r": p1Score += 5; break;

case "q": p1Score += 9; break;

default: p1Score += 0;

}

}

else //playerOne is black

{

switch(piece.getPieceType())

{

//Capture White by Player 1

case "P": p1Score++; break;

case "N": p1Score += 3; break;

case "B": p1Score += 3; break;

case "R": p1Score += 5; break;

case "Q": p1Score += 9; break;

//Capture Black by Player 2

case "p": p2Score++; break;

case "n": p2Score += 3; break;

case "b": p2Score += 3; break;

case "r": p2Score += 5; break;

case "q": p2Score += 9; break;

default: p1Score += 0;

}

}

}

public int getPlayerOneScores()

{

return p1Score;

}

public int getPlayerTwoScores()

{

return p2Score;

}

public boolean checkmate(Piece checker, String co)

{

for(int r = 0; r<array.length; r++)

for(int c = 0; c<array[r].length; c++)

if(!(checkOpen(r,c) || array[r][c].getColor().equals(co)))

{

if(array[r][c].getPieceType().equalsIgnoreCase("r") || array[r][c].getPieceType().equalsIgnoreCase("q"))

{

for(int k = 0;k<2;k++)//checks Rook directions

for(int i = -1;i<2;i+=2)

for(int j = 1;j<7;j++)

if(checkBounds(r+(j\*i\*k),c+(j\*i\*(1-k))))

{

if(!checkOpen(r+(j\*i\*k),c+(j\*i\*(1-k))))

j=7;

if(!checkForCheck(r+(j\*i\*k),c+(j\*i\*(1-k)), co))

return false;

}

else

j=7;

}

else if(array[r][c].getPieceType().equalsIgnoreCase("b") || array[r][c].getPieceType().equalsIgnoreCase("q"))

{

for(int k = -1;k<2;k+=2)//check Bishop directions

for(int i = -1;i<2;i+=2)

for(int j = 1;j<7;j++)

if(checkBounds(r+(k\*j),c+(j\*i)))

{

if(!checkOpen(r+(j\*i\*k),c+(j\*i\*(1-k))))

j=7;

if(!checkForCheck(r+(k\*j),c+(j\*i),co))

return false;

}

else

j=7;

}

else if(array[r][c].getPieceType().equalsIgnoreCase("n"))

{

for(int i = -1;i<2;i+=2)//checks Knight direction

for(int j = -1;j<2;j+=2)

for(int k = 0;k<2;k++)

if(checkBounds(r+(i\*(k+1)),c+(j\*(2-k))))

if(!checkForCheck(r+(i\*(k+1)),c+(j\*(2-k)),co))

return false;

}

else if(array[r][c].getPieceType().equals("p") && co.equals("b"))

{

for(int i = -1; i<2; i+=2)

if(!(checkOpen(r+1,c+i) || checkForCheck(r+1,c+i,co)))

return false;

if(checkOpen(r+1,c) && !checkForCheck(r+1,c,co))

return false;

}

else if(array[r][c].getPieceType().equals("P") && co.equals("w"))

{

for(int i = -1; i<2; i+=2)

if(!(checkOpen(r-1,c+i) || checkForCheck(r-1,c+i,co)))

return false;

if(checkOpen(r-1,c) && !checkForCheck(r-1,c,co))

return false;

}

}

return true;

}

}