Chess is a classic recreational-competitive board game. It is a very good exercise for our brain. Hence, we decided to recreate this classic game in Java.

**ChessJD**

|  |
| --- |
| **gameComponents**  **Alliance**  **Board**  **Game**  **Player**  **ScorePanel**  **Tile** |

|  |
| --- |
| **gameEntities**  **Bishop**  **King**  **Knight**  **Pawn**  **Piece**  **PieceType**  **Queen**  **Rook** |

**Alliance.java**

package gameComponents;

public enum Alliance {

//Pieces can either be part of the white alliance or the black alliance

WHITE, BLACK

}

Explanation :

In chess, pieces can either be part of the white alliance or the black alliance. Hence we have defined an enumerator consisting of BLACK and WHITE.

**Bishop.java**

package gameEntities;

import gameComponents.Alliance;

import gameComponents.Tile;

public class Bishop extends Piece {

public Bishop(Alliance pieceAlliance) {

super(pieceAlliance);

this.setPieceType(PieceType.BISHOP);

this.setPieceImg("Images/wBishop.png", "Images/bBishop.png");

}

@Override

public boolean isValidMove(Tile startTile, Tile endTile) {

//calculates change in x and y values

int changeY = Math.abs(endTile.getRow() - startTile.getRow());

int changeX = Math.abs(endTile.getColumn() - startTile.getColumn());

//checks if same coloured piece occupies destination tile

if (isAllianceOverlap(startTile, endTile))

return false;

else

return changeX == changeY; //change in x and y values is the same for diagonal motion

}

@Override

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) {

int yChange = endTile.getRow() - startTile.getRow();

int xChange = endTile.getColumn() - startTile.getColumn();

int xDir = Integer.signum(xChange);

int yDir = Integer.signum(yChange);

//if tile is between start and end is occupied than path is not valid

for (int i = 1; i < Math.abs(xChange); i++) {

if (tileMap[startTile.getRow() + i \* yDir][startTile.getColumn() + i \* xDir].isOccupied()) {

return false;

}

}

return true;

}

}

Explanation:

Bishop piece class. Holds information regarding movement of the bishop piece.

Method List:

public boolean isValidMove(Tile startTile, Tile endTile) - Method to check if selected tiles fall within acceptable move parameters for the current piece

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) - Method to check board and see if the move is still valid (checks for obstructions)

**Board.java**

package gameComponents;

import java.awt.Color;

import java.awt.GridLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JFrame;

import javax.swing.JPanel;

import gameEntities.\*;

public class Board extends JPanel implements ActionListener {

private static Tile[][] tileMap; //array to hold the tiles that make up the game board

private boolean firstClick;

private Piece tempPiece;

private Tile startTile;

private Tile endTile;

private final Color SQUARE\_COLOUR\_ONE = new Color(186, 202, 68); //Dark green colour

private final Color SQUARE\_COLOUR\_TWO = new Color(238, 238, 210); //light green colour

public Board() {

super(); //sets default values for the global variables

startTile = null;

endTile = null;

tempPiece = null;

firstClick = true;

//creates and 8 by 8 grid and a 2 dimensional array (8 by 8) of tiles

this.setLayout(new GridLayout(8, 8));

tileMap = new Tile[8][8];

//places the tiles in an alternating pattern

for(int i = 0; i < tileMap.length; i++) { //row # (0, 7)

for(int j = 0; j < tileMap.length; j++) { //Column # (0, 7)

tileMap[i][j] = new Tile(i, j, null);

//creates the checkered colour pattern

if((i + j) % 2 == 0)

tileMap[i][j].setBackground(SQUARE\_COLOUR\_ONE);

else

tileMap[i][j].setBackground(SQUARE\_COLOUR\_TWO);

//adds the tiles and listener

this.add(tileMap[i][j]);

tileMap[i][j].addActionListener(this);

}

}

resetBoard(); //places chess pieces

this.setSize(500, 500); //sets panel size and visibility

this.setVisible(true);

}

public void refreshBoard() {

//loops through array and displays the pieces found on the tile

for(Tile[] tiles : tileMap) {

for(int j = 0; j < tileMap.length; j++) {

tiles[j].displayPiece();

resetColors();

}

}

}

public void resetColors() {

//loops through the array and sets colours in an alternating pattern

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

for(int j = 0; j < tileMap.length; j++) {

if((i + j) % 2 == 0)

tileMap[i][j].setBackground(SQUARE\_COLOUR\_ONE);

else

tileMap[i][j].setBackground(SQUARE\_COLOUR\_TWO);

}

}

}

public void resetBoard() {

for(int i = 0; i < tileMap.length; i++) { //pawns

tileMap[1][i].setPiece(new Pawn(Alliance.BLACK));

tileMap[6][i].setPiece(new Pawn(Alliance.WHITE));

if(i == 0 || i == 7) { //rooks

tileMap[0][i].setPiece(new Rook(Alliance.BLACK));

tileMap[7][i].setPiece(new Rook(Alliance.WHITE));

}

else if(i == 1 || i == 6) { //knights

tileMap[0][i].setPiece(new Knight(Alliance.BLACK));

tileMap[7][i].setPiece(new Knight(Alliance.WHITE));

}

else if(i == 2 || i == 5) { //bishops

tileMap[0][i].setPiece(new Bishop(Alliance.BLACK));

tileMap[7][i].setPiece(new Bishop(Alliance.WHITE));

}

}

//sets the queens and kings

tileMap[0][3].setPiece(new Queen(Alliance.BLACK));

tileMap[7][3].setPiece(new Queen(Alliance.WHITE));

tileMap[0][4].setPiece(new King(Alliance.BLACK));

tileMap[7][4].setPiece(new King(Alliance.WHITE));

refreshBoard(); //displays the different pieces

}

@Override

public void actionPerformed(ActionEvent evt) {

//loops through tile objects checking for button clicks

for(int i = 0; i < tileMap.length; i++) { //loops through rows

for(int j = 0; j < tileMap.length; j++) { //loops through columns

if(tileMap[i][j] == evt.getSource()) {

if(firstClick) { //sets start tile if user has not already selected a tile

firstClick = false;

startTile = tileMap[i][j];

startTile.setBackground(new Color(255, 131, 117)); //sets colour to indicate starting tile

moveOptions();

}

else { //sets end tile if user has selected a starting tile

if(!(startTile.getRow() == i && startTile.getColumn() == j)) {

firstClick = true;

resetColors(); //resets board colours removing red marker

endTile = tileMap[i][j];

}

else { //if user selects the same tile twice than clear start and end tile selection

clearSelection();

}

}

}

}

}

}

public void clearSelection () {

startTile = null;

endTile = null;

resetColors();

firstClick = true;

}

public void moveOptions() {

for(Tile[] tiles : tileMap) {

for(int j = 0; j < tileMap.length; j++) {

if(startTile.getPiece() != null && startTile.getPiece().isValidMove(startTile, tiles[j])) {

if(startTile.getPiece().isValidPath(startTile, tiles[j], tileMap)) {

//changes to yellow colour for all squares where piece is allowed to move

tiles[j].setBackground(new Color(255, 251, 133, 207));

}

}

}

}

}

public void move() {

tempPiece = endTile.getPiece();

setTilePiece(endTile, startTile.getPiece());

setTilePiece(startTile, null);

refreshBoard();

}

public void undoMove() {

setTilePiece(startTile, endTile.getPiece());

setTilePiece(endTile, tempPiece);

refreshBoard();

}

public Tile getTile(int row, int column) {

return tileMap[row][column];

}

public void setTilePiece(Tile tile, Piece newPiece) {

tileMap[tile.getRow()][tile.getColumn()].setPiece(newPiece);

}

public boolean getFirstClick() {

return firstClick;

}

public Tile getStartTile() {

return startTile;

}

public Tile getEndTile() {

return endTile;

}

public Tile[][] getTileMap() {

return tileMap;

}

public static void main(String[] args) {

JFrame testFrame = new JFrame("TESTING FRAME");

Board board = new Board();

testFrame.add(board);

testFrame.setSize(500, 500);

testFrame.setVisible(true);

testFrame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

}

Explanation:

public void refreshBoard() - Method that updates button icons to show changes to piece positions

public void resetColors() - Method rests the board colours to the original green and light green layout

public void resetBoard() - Method places all of the pieces onto the game board

public void actionPerformed(ActionEvent evt) - Loops through and sets start tile and end tile locations depending on the button that has been selected

public void clearSelection () - Method to clear user's piece selection

public void moveOptions() - Method responsible for showing all of the possible moves when a given piece is selected

public void move() - Method to move Pieces around the board

public void undoMove() - Method to undo the last move made by the user

public Tile getTile(int row, int column) - Method to return a tile based on the given row and column values

public void setTilePiece(Tile tile, Piece newPiece) - Method to assign a piece to a given tile

public boolean getFirstClick() - Method to check if the user has only clicked once

public Tile getStartTile() - Method to get starting tile selected by user

public Tile getEndTile() - Method to get ending tile selected by the user

public Tile[][] getTileMap() - Method to return the entire tileMap (provides board information)

public static void main(String[] args) - Self-testing main method

**Game.java**

package gameComponents;

import gameEntities.\*;

import javax.swing.\*;

public class Game {

private final Board gb;

private final ScorePanel playerOnePanel, playerTwoPanel;

private final Player playerOne, playerTwo;

boolean gameOver;

public Game () {

//creates new frame and sets box layout along the y-axis

JFrame gameFrame = new JFrame("Chess Game");

gameFrame.setLayout(new BoxLayout(gameFrame.getContentPane(), BoxLayout.Y\_AXIS));

//initializes data and sets defaults

gameOver = false;

gb = new Board();

playerOne = new Player(Alliance.BLACK, false);

playerTwo = new Player (Alliance.WHITE, true);

playerOnePanel = new ScorePanel(playerOne);

playerTwoPanel = new ScorePanel(playerTwo);

gameFrame.add(playerOnePanel);

gameFrame.add(gb);

gameFrame.add(playerTwoPanel);

//Edits window preferences

gameFrame.setSize(600,800);

gameFrame.setLocation(400, 10);

gameFrame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

gameFrame.setVisible(true);

while(!gameOver) { //runs until the game is over

gb.repaint(); //refreshes the board

//sets turn colours in score panel to show whose turn it is

if(testCheck() == null) {

playerOnePanel.setTurn();

playerTwoPanel.setTurn();

}

else { //removes colours if a player is in check

if(testCheck() == Alliance.WHITE)

playerOnePanel.clearText();

else

playerTwoPanel.clearText();

}

//if valid tiles are selected and the correct alliance has been selected than moves piece

if(gb.getStartTile() != null && gb.getEndTile() != null && gb.getStartTile().getPiece() != null) {

if(playerTwo.isTurn() && gb.getStartTile().getPiece().getPieceAlliance() == Alliance.WHITE || playerOne.isTurn() && gb.getStartTile().getPiece().getPieceAlliance() == Alliance.BLACK) {

if(gb.getFirstClick()) {

movePiece(); //calls method to move pieces

gb.clearSelection(); //clears selection for next itteration of the loop

//Tests to see if any of the player are in check or checkmate

if(testCheck() == Alliance.WHITE) {

playerTwoPanel.setCheck();

} else if (testCheck() == Alliance.BLACK) {

playerOnePanel.setCheck();

} else {

playerOnePanel.clearText();

playerTwoPanel.clearText();

}

}

}

else {

//if player has not selected valid tiles than selection is cleared forcing the player to choose again

gb.clearSelection();

}

}

}

if(testCheck() == Alliance.WHITE) //displays winning message depending on the king in check

JOptionPane.showMessageDialog(null, "Game Over, Black Wins!");

else

JOptionPane.showMessageDialog(null, "Game Over, White Wins!");

System.exit(0);

}

public Alliance testCheck() {

King whiteKing = null, blackKing = null; //variables to hold white and black kings

//finds the locations of the two kings

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

for(int j = 0; j < gb.getTileMap().length; j++) {

if(gb.getTile(i, j).isOccupied() && gb.getTile(i, j).getPiece().getPieceType() == PieceType.KING) {

if(gb.getTile(i, j).getPiece().getPieceAlliance() == Alliance.WHITE)

whiteKing = (King) gb.getTile(i, j).getPiece();

else

blackKing = (King) gb.getTile(i, j).getPiece();

}

}

}

assert whiteKing != null; //makes sure both the white and black kings are present on the board

assert blackKing != null;

//returns the appropriate value depending on if and which kings are under check

if(whiteKing.inCheck(gb.getTileMap(), whiteKing.findKing(gb.getTileMap()))) {

if(whiteKing.checkMate(gb.getTileMap())) { //tests for white checkmate

gameOver = true;

}

return Alliance.WHITE;

}

else if(blackKing.inCheck(gb.getTileMap(), blackKing.findKing(gb.getTileMap()))) {

if(blackKing.checkMate(gb.getTileMap())) { //tests for black checkmate

gameOver = true;

}

return Alliance.BLACK;

}

else {

return null;

}

}

public void movePiece() {

//checks if the move is valid

if(gb.getStartTile().isOccupied() && gb.getStartTile().getPiece().isValidMove(gb.getStartTile(), gb.getEndTile())) {

if(gb.getStartTile().getPiece().isValidPath(gb.getStartTile(), gb.getEndTile(), gb.getTileMap())) {

if(gb.getEndTile().isOccupied()) {

//updates score for black player if white piece is captured

if(gb.getEndTile().getPiece().getPieceAlliance() == Alliance.WHITE) {

playerOne.updateScore(gb.getEndTile().getPiece());

playerOnePanel.setScore();

}

//updates score for white player if black piece is captured

else {

playerTwo.updateScore((gb.getEndTile().getPiece()));

playerTwoPanel.setScore();

}

}

gb.move(); //moves piece

//checks if moved piece is pawn

if(gb.getEndTile().getPiece().getPieceType() == PieceType.PAWN) {

gb.getEndTile().getPawn().setFirstMove(false); //stops pawn from moving 2 spaces after initial move

//handles pawn promotion if pawn has made it to the appropriate square

if((gb.getEndTile().getPawn().getPieceAlliance() == Alliance.WHITE && gb.getEndTile().getRow() == 0) ||(gb.getEndTile().getPawn().getPieceAlliance() == Alliance.BLACK && gb.getEndTile().getRow() == 7)) {

pawnPromotion(gb);

}

}

//if moving a piece results in check than undoes move

if(testCheck() != null && gb.getEndTile().getPiece().getPieceAlliance() == testCheck()) {

if(gb.getEndTile().getPiece().getPieceType() == PieceType.PAWN) {

gb.getEndTile().getPawn().setFirstMove(true);

}

gb.undoMove();

if(gb.getStartTile().getPiece().getPieceAlliance() == Alliance.WHITE) {

playerTwo.fixScore(gb.getEndTile().getPiece());

}

else {

playerOne.fixScore(gb.getEndTile().getPiece());

}

playerOnePanel.setScore();

playerTwoPanel.setScore();

gb.clearSelection();

}

else { //if move is successful moves onto next turn

playerOne.setTurn(!playerOne.isTurn());

playerTwo.setTurn(!playerTwo.isTurn());

}

}

}

}

public void pawnPromotion(Board gb) {

//gets letter from user

char letter = JOptionPane.showInputDialog(null, "Pawn promotion: enter the corresponding character: \n" + "q - Queen \n r - Rook \n k - Knight \n b - Bishop").charAt(0);

//depending on the letter entered promotes pawn

switch (letter) {

case 'r':

gb.getEndTile().setPiece(new Rook(gb.getEndTile().getPiece().getPieceAlliance()));

break;

case 'k':

gb.getEndTile().setPiece(new Knight(gb.getEndTile().getPiece().getPieceAlliance()));

break;

case 'b':

gb.getEndTile().setPiece(new Bishop(gb.getEndTile().getPiece().getPieceAlliance()));

break;

default:

gb.getEndTile().setPiece(new Queen(gb.getEndTile().getPiece().getPieceAlliance()));

break;

}

gb.refreshBoard();

}

public static void main(String[] args) {

new Game();

}

}

Explanation:

Chess game class controls turns and other interactions between the users, board, score panel, and other entities.

Method List:

public Alliance testCheck() - Method to check if the king is under check

public void movePiece() - Method to move piece and handle changes to piece once moved

public void pawnPromotion(Board gb) - Method to prompt user for pawn promotion and change piece type

public static void main(String[] args) - Method to run game window

**King.java**

package gameEntities;

import gameComponents.Alliance;

import gameComponents.Tile;

import java.util.ArrayList;

public class King extends Piece {

private Tile checkedBy;

private ArrayList<Tile> checkedPath = new ArrayList<>();

public King (Alliance pieceAlliance) {

super (pieceAlliance);

this.setPieceType(PieceType.KING);

this.checkedBy = null;

this.setPieceImg("Images/wKing.png", "Images/bKing.png");

}

@Override

public boolean isValidMove(Tile startTile, Tile endTile) {

int changeY = Math.abs(endTile.getRow() - startTile.getRow());

int changeX = Math.abs(endTile.getColumn() - startTile.getColumn());

if(isAllianceOverlap(startTile, endTile))

return false;

else

return changeX <= 1 && changeY <= 1; //can only move one tile

}

public Tile findKing(Tile[][] tileMap) {

//loops until king is found

for(Tile[] tiles : tileMap) {

for(int j = 0; j < tileMap.length; j++) {

if(tiles[j].isOccupied() && tiles[j].getPiece().getPieceType() == PieceType.KING && tiles[j].getPiece().getPieceAlliance() == this.getPieceAlliance()) {

return tiles[j];

}

}

}

return null;

}

@Override

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) {

Piece temp = endTile.getPiece();

endTile.setPiece(startTile.getPiece());

if(!this.inCheck(tileMap, endTile)) {

endTile.setPiece(temp);

return true;

}

endTile.setPiece(temp);

return false;

}

public boolean inCheck(Tile[][] tileMap, Tile kingTile) {

for(Tile[] tiles : tileMap) {

for(int j = 0; j < tileMap.length; j++) {

if(tiles[j].isOccupied() && tiles[j].getPiece().getPieceAlliance() != this.getPieceAlliance()) {

if(tiles[j].getPiece() instanceof Pawn) { //tests for diagonal pawn movement (pawns only kill diagonally)

if(tiles[j].getPiece().isValidMove(tiles[j], kingTile)) {

if(Math.abs(kingTile.getColumn() - tiles[j].getColumn()) > 0) {

this.checkedBy = tiles[j];

return true;

}

}

}

//if king is in path of another piece than it is in check

else if(tiles[j].getPiece().isValidMove(tiles[j], kingTile) && tiles[j].getPiece().isValidPath(tiles[j], kingTile, tileMap)) {

this.checkedBy = tiles[j];

return true;

}

}

}

}

return false;

}

public boolean checkMate (Tile[][] tileMap) {

Tile kingTile = findKing(tileMap);

int rowDiff = kingTile.getRow() - checkedBy.getRow();

int colDiff = kingTile.getColumn() - checkedBy.getColumn();

int tileAmt = Math.max(Math.abs(rowDiff), Math.abs(colDiff));

for(Tile[] tiles: tileMap) { //checks if the king can move

for(int j = 0; j < tileMap.length; j++) {

if(kingTile.getPiece() != null && kingTile.getPiece().isValidMove(kingTile, tiles[j]) && kingTile.getPiece().isValidPath(kingTile, tiles[j], tileMap)) {

return false;

}

}

}

try { //adds path between king and the piece that is the cause for check

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

checkedPath.add(tileMap[checkedBy.getRow() + i \* Integer.signum(rowDiff)][checkedBy.getColumn() + i \* Integer.signum(colDiff)]);

}

} catch (Exception e) {

return true;

}

//loops through and checks if any piece can block the check

for(Tile[] tiles : tileMap) {

for(int j = 0; j < tileMap.length; j++) {

for(Tile[] value : tileMap) {

for(int c = 0; c < tileMap.length; c++) {

if(tiles[j].getPiece() != null && tiles[j].getPiece().getPieceAlliance() == this.getPieceAlliance() && tiles[j].getPiece().isValidMove(tiles[j], value[c]) && tiles[j].getPiece().isValidPath(tiles[j], value[c], tileMap)) {

for(Tile pathTile : checkedPath) {

if(value[c] == pathTile) { //tests if piece is able to move into path

checkedPath.clear();

return false;

}

}

}

}

}

}

}

checkedPath.clear();

return true;

}

}

Explanation:

King piece class. Holds information regarding movement of the King piece. Also tests for check and checkmate.

Method List:

public boolean isValidMove(Tile startTile, Tile endTile) - Method to check if selected tiles fall within acceptable move parameters for the current piece

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) - Method to check board and see if the move is still valid (checks for obstructions)

public Tile findKing(Tile[][] tileMap) - Method to locate king on gameboard

public boolean inCheck(Tile[][] tileMap, Tile kingTile) - Method to test if the king is in check

public boolean checkMate (Tile[][] tileMap) - Method to check if the king is in checkmate

**Knight.java**

package gameEntities;

import gameComponents.Alliance;

import gameComponents.Tile;

public class Knight extends Piece {

public Knight(Alliance pieceAlliance) {

super(pieceAlliance);

this.setPieceType(PieceType.KNIGHT);

this.setPieceImg("Images/wKnight.png", "Images/bKnight.png");

}

@Override

public boolean isValidMove(Tile startTile, Tile endTile) {

int changeY = Math.abs(endTile.getRow() - startTile.getRow());

int changeX = Math.abs(endTile.getColumn() - startTile.getColumn());

//checks for overlap

if(isAllianceOverlap(startTile, endTile))

return false;

else

return (changeX == 2 && changeY == 1) || (changeX == 1 && changeY == 2); //motion for knight

}

@Override

//always true since the knight can jump over pieces

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) {

return true;

}

}

Explanation:

Knight piece class. Holds information regarding movement of the knight piece.

Method List:

public boolean isValidMove(Tile startTile, Tile endTile) - Method to check if selected tiles fall within acceptable move parameters for the current piece

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) - Method to check board and see if the move is still valid (checks for obstructions)

**Pawn.java**

package gameEntities;

import gameComponents.Alliance;

import gameComponents.Tile;

public class Pawn extends Piece {

private boolean firstMove;

public Pawn (Alliance pieceAlliance) {

super(pieceAlliance);

this.setPieceType(PieceType.PAWN);

this.setPieceImg("Images/wPawn.png", "Images/bPawn.png");

firstMove = true;

}

public void setFirstMove(boolean newValue) {

this.firstMove = newValue;

}

@Override

public boolean isValidMove(Tile startTile, Tile endTile) {

int moveSpaces;

//finds change in position for rows (Y) and columns (X)

//since the pawn can only move forward row difference (change in x) is not an absolute value

int changeY = endTile.getRow() - startTile.getRow();

int changeX = Math.abs(endTile.getColumn() - startTile.getColumn());

//pawn can move 2 spaces on first move and 1 on every move thereafter

if(firstMove)

moveSpaces = 2;

else

moveSpaces = 1;

//to insure pawns can only move forward white pieces have their moveSpace

//value turned into a negative integer

if(getPieceAlliance() == Alliance.WHITE)

moveSpaces \*= -1;

//checks if alliance piece is already on destination square

if(isAllianceOverlap(startTile, endTile))

return false;

//if piece is white then change in Y must range from 0 to negative moveSpaces. The piece can only move diagonally by one row (change y) and not 2 (2 % 2 = 0) and (1 % 2 = 1)

else if(getPieceAlliance() == Alliance.WHITE && (changeY >= moveSpaces && changeY < 0) && changeX <= Math.abs(changeY % 2))

return true;

//if the piece is black change in Y must range from 0 to positive moveSpaces

else

return getPieceAlliance() == Alliance.BLACK && (changeY <= moveSpaces && changeY > 0) && changeX <= Math.abs(changeY % 2);

}

@Override

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) {

int yChange = endTile.getRow() - startTile.getRow();

int xChange = endTile.getColumn() - startTile.getColumn();

int yDir = Integer.signum(yChange);

if(Math.abs(yChange) == 2 && xChange == 0) {

//stops pawn moving forward if it has to jump over a piece

if(tileMap[startTile.getRow() + yDir][startTile.getColumn()].isOccupied()) {

return false;

}

//stops pawn from attacking enemy piece in front of it (2 steps in front)

else

return !endTile.isOccupied() || endTile.getPiece().getPieceAlliance() == startTile.getPiece().getPieceAlliance();

}

else if(Math.abs(yChange) == 1 && xChange == 0) {

//stops pawn from attacking enemy piece in front of it (1 step in front)

return !endTile.isOccupied() || endTile.getPiece().getPieceAlliance() == startTile.getPiece().getPieceAlliance();

}

else if(Math.abs(yChange) == 1 && Math.abs(xChange) == 1) {

return endTile.isOccupied() && endTile.getPiece().getPieceAlliance() != startTile.getPiece().getPieceAlliance();

}

return true;

}

}

Explanation:

Pawn piece class. Holds information regarding movement of the Pawn piece.

Method List:

public boolean isValidMove(Tile startTile, Tile endTile) - Method to check if selected tiles fall within acceptable move parameters for the current piece

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) - Method to check board and see if the move is still valid (checks for obstructions)

**Piece.java**

package gameEntities;

import gameComponents.Alliance;

import gameComponents.Tile;

import javax.swing.\*;

public abstract class Piece {

private Alliance pieceAlliance;

private PieceType pieceType;

private ImageIcon pieceImg;

public Piece(Alliance pieceAlliance) {

this.pieceAlliance = pieceAlliance;

this.pieceImg = null;

}

public abstract boolean isValidMove(Tile startTile, Tile endTile);

public abstract boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap);

public ImageIcon getPieceImg() {

return pieceImg;

}

public void setPieceType(PieceType pieceType) {

this.pieceType = pieceType;

}

public PieceType getPieceType() {

return this.pieceType;

}

public void setPieceImg(String imgWFileName, String imgBFileName) {

if (pieceAlliance == Alliance.WHITE)

this.pieceImg = new ImageIcon(imgWFileName);

else

this.pieceImg = new ImageIcon(imgBFileName);

}

public boolean isAllianceOverlap(Tile startTile, Tile endTile) {

return (endTile.isOccupied() && startTile.isOccupied() && endTile.getPiece().getPieceAlliance() == startTile.getPiece().getPieceAlliance());

}

public Alliance getPieceAlliance() {

return pieceAlliance;

}

}

Explanation:

Piece class. Holds information regarding movement of the Pawn piece.

Method List:

public boolean isValidMove(Tile startTile, Tile endTile) - Method to check if selected tiles fall within acceptable move parameters for the current piece

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) - Method to check board and see if the move is still valid (checks for obstructions)

public ImageIcon getPieceImg() - Method to get piece image

public void setPieceType (PieceType pieceType) - Method to set the type of piece

public PieceType getPieceType () - Method to get the type of piece

public void setPieceImg(String imgWFileName, String imgBFileName) - Method to set piece image

public boolean isAllianceOverlap (Tile startTile, Tile endTile) - Method to test if move will overlap

public Alliance getPieceAlliance() - Method to get piece alliance

**Piecetype.java**

package gameEntities;

public enum PieceType {

//List of the different kinds of chess pieces

PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING

}

Explanation:

PieceType class. Holds an enumerator consisting of all the pieces that are there in Chess.

**Player.java**

package gameComponents;

import gameEntities.Piece;

public class Player {

private Alliance playerAlliance;

private boolean isTurn;

private int score;

public Player (Alliance pAlliance, boolean isTurn) {

//sets default values for global variables

this.playerAlliance = pAlliance;

this.isTurn = isTurn;

this.score = 0;

}

public void updateScore(Piece killedPiece) {

//assigns a score from 1,3,5,and 8 depending on the piece captured

switch(killedPiece.getPieceType()) {

case PAWN :

this.score += 1;

break;

case ROOK:

this.score += 5;

break;

case KNIGHT:

this.score += 3;

break;

case BISHOP:

this.score += 3;

break;

case QUEEN:

this.score += 8;

break;

default:

break;

}

}

public void fixScore(Piece revivedPiece) {

//subtracts the corresponding value (added previously) for the piece that is revived

if(revivedPiece != null) {

switch(revivedPiece.getPieceType()) {

case PAWN:

this.score -= 1;

break;

case ROOK:

this.score -= 5;

break;

case KNIGHT:

this.score -= 3;

break;

case BISHOP:

this.score -= 3;

break;

case QUEEN:

this.score -= 8;

break;

default:

break;

}

}

}

public int getScore () {

return this.score;

}

public Alliance getPlayerAlliance () {

return this.playerAlliance;

}

public boolean isTurn() {

return isTurn;

}

public void setTurn (boolean newTurnValue) {

this.isTurn = newTurnValue;

}

}

Explanation:

Player class to handle information regarding player such as player alliance, score, and turn

Method List:

public void updateScore(Piece killedPiece) - Method to update score depending on the piece that has been captured

public void fixScore(Piece revivedPiece) - Method to fix score if a move is undone

public int getScore () - Method returns value for score

public Alliance getPlayerAlliance () - Method return player alliance (either black or white)

public boolean isTurn() - Method checks to see if its the users turn

public void setTurn (boolean newTurnValue) - Method to set new value for users turn

**Queen.java**

package gameEntities;

import gameComponents.Alliance;

import gameComponents.Tile;

public class Queen extends Piece {

public Queen(Alliance pieceAlliance) {

super(pieceAlliance);

this.setPieceType(PieceType.QUEEN);

this.setPieceImg("Images/wQueen.png", "Images/bQueen.png");

}

@Override

public boolean isValidMove(Tile startTile, Tile endTile) {

int changeY = Math.abs(endTile.getRow() - startTile.getRow());

int changeX = Math.abs(endTile.getColumn() - startTile.getColumn());

if(isAllianceOverlap(startTile, endTile))

return false;

else if((changeX > 0 && changeY == 0) || (changeX == 0 && changeY > 0)) //tests for diagonal and linear motion

return true;

else

return changeX == changeY;

}

@Override

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) {

int yChange = endTile.getRow() - startTile.getRow();

int xChange = endTile.getColumn() - startTile.getColumn();

int xDir = Integer.signum(xChange);

int yDir = Integer.signum(yChange);

if(yDir == 0) { //horizontal path obstruction

for(int i = 1; i < Math.abs(xChange); i++) {

if(tileMap[startTile.getRow()][startTile.getColumn() + i \* xDir].isOccupied()) {

return false;

}

}

}

else if(xDir == 0) { //vertical path obstruction

for(int i = 1; i < Math.abs(yChange); i++) {

if(tileMap[startTile.getRow() + i \* yDir][startTile.getColumn()].isOccupied()) {

return false;

}

}

}

else { //diagonal path obstruction

for(int i = 1; i < Math.abs(xChange); i++) {

if(tileMap[startTile.getRow() + i \* yDir][startTile.getColumn() + i \* xDir].isOccupied()) {

return false;

}

}

}

return true;

}

}

Explanation:

Queen piece class. Holds information regarding movement of the Queen piece.

Method List:

public boolean isValidMove(Tile startTile, Tile endTile) - Method to check if selected tiles fall within acceptable move parameters for the current piece

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) - Method to check board and see if the move is still valid (checks for obstructions)

**Rook.java**

package gameEntities;

import gameComponents.Alliance;

import gameComponents.Tile;

public class Rook extends Piece {

public Rook(Alliance pieceAlliance) {

super(pieceAlliance);

this.setPieceType(PieceType.ROOK);

this.setPieceImg("Images/wRook.png", "Images/bRook.png");

}

@Override

public boolean isValidMove(Tile startTile, Tile endTile) {

int changeY = Math.abs(endTile.getRow() - startTile.getRow());

int changeX = Math.abs(endTile.getColumn() - startTile.getColumn());

if(isAllianceOverlap(startTile, endTile))

return false;

else

return (changeX > 0 && changeY == 0) || (changeX == 0 && changeY > 0); //tests for linear movement

}

@Override

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) {

int yChange = endTile.getRow() - startTile.getRow();

int xChange = endTile.getColumn() - startTile.getColumn();

int xDir = Integer.signum(xChange);

int yDir = Integer.signum(yChange);

if(yDir == 0) { //horizontal obstruction

for (int i = 1; i < Math.abs(xChange); i++) {

if(tileMap[startTile.getRow()][startTile.getColumn() + i \* xDir].isOccupied()) {

return false;

}

}

}

else { //vertical obstruction

for(int i = 1; i < Math.abs(yChange); i++) {

if(tileMap[startTile.getRow() + i \* yDir][startTile.getColumn()].isOccupied()) {

return false;

}

}

}

return true;

}

}

Explanation:

Rook piece class. Holds information regarding movement of the Rook piece.

Method List:

public boolean isValidMove(Tile startTile, Tile endTile) - Method to check if selected tiles fall within acceptable move parameters for the current piece

public boolean isValidPath(Tile startTile, Tile endTile, Tile[][] tileMap) - Method to check board and see if the move is still valid (checks for obstructions)

**ScorePanel.java**

package gameComponents;

import javax.swing.\*;

import java.awt.\*;

public class ScorePanel extends JPanel {

private Player gamePlayer;

private JLabel name;

private JLabel score;

public ScorePanel (Player gamePlayer) {

super();

this.setLayout(null); //uses null layout

this.gamePlayer = gamePlayer;

name = new JLabel(gamePlayer.getPlayerAlliance().toString()); //Displays player alliance

name.setFont(new Font ("TimesRoman", Font.BOLD, 50));

name.setBounds(20,20,250,70);

this.add(name);

JLabel scoreTag = new JLabel("Score: "); //Label for score tag

scoreTag.setFont(new Font ("TimesRoman", Font.BOLD, 20));

scoreTag.setBounds(450,20,250,70);

this.add(scoreTag);

score = new JLabel("0"); //Displays player score

score.setFont(new Font ("TimesRoman", Font.BOLD, 50));

score.setBounds(500,20,300,70);

this.add(score);

//sets panel features

this.setSize(500,100);

this.setBackground(new Color(238, 238, 210));

this.setVisible(true);

}

public void setCheck() {

name.setForeground(new Color(255, 131, 117));

}

public void setTurn() {

if(gamePlayer.isTurn())

name.setForeground(new Color(85, 238, 116));

else

clearText();

}

public void clearText() {

name.setForeground(Color.BLACK);

}

public void setScore () {

this.score.setText(Integer.toString(gamePlayer.getScore()));

}

public static void main(String[] args) {

JFrame testFrame = new JFrame("Test");

ScorePanel sp = new ScorePanel(new Player(Alliance.WHITE, true));

testFrame.add(sp);

testFrame.setSize(500, 100);

testFrame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

testFrame.setVisible(true);

}

}

Explanation:

Class creates a panel to show information such as user alliance and score

Method List:

public void setCheck() - Method to change text colour to red when player is check

public void setTurn() - Method to set text colour to green if it’s the current players turn

public void clearText() - Method to restore text colour to default (Black)

public void setScore () - Method to update score on panel

public static void main(String[] args) - Self-testing main method

**Tile.java**

package gameComponents;

import gameEntities.Pawn;

import gameEntities.Piece;

import javax.swing.\*;

public class Tile extends JButton {

private Piece piece;

private int row;

private int column;

public Tile(int row, int column, Piece piece) {

this.row = row;

this.column = column;

this.piece = piece;

this.setBorder(javax.swing.BorderFactory.createEmptyBorder());

}

public boolean isOccupied() {

return piece != null;

}

public void displayPiece() {

if(isOccupied())

this.setIcon(piece.getPieceImg());

else

this.setIcon(null);

}

public int getRow() {

return this.row;

}

public int getColumn() {

return this.column;

}

public Piece getPiece() {

return this.piece;

}

public Pawn getPawn() {

return (Pawn) this.piece;

}

public void setPiece(Piece newPiece) {

this.piece = newPiece;

}

}

Explanation:

Tile class to make up one of the 64 tiles found on the chess board

Method List:

public boolean isOccupied() - Method to check if the current tile has a piece

public void displayPiece() - Method to display the image of the piece found on this tile

public int getRow() - Method to get tile row

public int getColumn() - Method to get tile column

public Piece getPiece() - Method to get piece associated with tile

public Pawn getPawn() - Method to get pawn on tile

public void setPiece(Piece newPiece) - Method to set new piece on tile

Done by -

**Santushti Gandhi - 20058570034**

**Aman Goyal - 20058570005**