

========================= Фигуры ==================================

package org.example;  
  
import java.util.Stack;  
  
abstract class Piece {  
  
 final private PieceColor color;  
  
 private PiecePosition currentPosition;  
  
 private int changeX;  
 private int changeY;  
  
 Piece(PieceColor color, PiecePosition position) {  
 this.color = color;  
 this.currentPosition = position;  
 changeY = 0;  
 changeX = 0;  
 }  
  
 abstract boolean isValidMoveType(Move move);  
  
 public boolean isValidTakeType(Move move) {  
 return isValidMoveType(move);  
 }  
  
 public PieceColor getColor() {  
 return this.color;  
 }  
  
 public PiecePosition getPosition() {  
 return this.currentPosition;  
 }  
  
 public void setPosition(PiecePosition position) {  
 this.currentPosition = position;  
 }  
  
 public boolean isStartPositionValid(PiecePosition start, PiecePosition end) {  
 this.changeX = start.getBlock().x - end.getBlock().x;  
 this.changeY = start.getBlock().y - end.getBlock().y;  
 return (start == this.getPosition()) && (start != end);  
 }  
  
 public int getChangeX() {  
 return changeX;  
 }  
  
 public int getChangeY() {  
 return changeY;  
 }  
  
}

public class King extends Piece {  
  
 King(PieceColor color, PiecePosition position) {  
 super(color, position);  
 }  
  
 @Override  
 boolean isValidMoveType(Move move) {  
 return isStartPositionValid(move.startPosition, move.endPosition);  
 }  
}

public class Queen extends Piece {  
  
 Queen(PieceColor color, PiecePosition position) {  
 super(color, position);  
 }  
  
 @Override  
 boolean isValidMoveType(Move move) {  
 return isStartPositionValid(move.startPosition, move.endPosition) &&  
 (Math.*abs*(this.getChangeX()) == Math.*abs*(this.getChangeY()) ||  
 this.getChangeX() == 0 ||  
 this.getChangeY() == 0  
 );  
 }  
  
}

public class Rook extends Piece {  
  
 Rook(PieceColor color, PiecePosition position) {  
 super(color, position);  
 }  
  
 @Override  
 boolean isValidMoveType(Move move) {  
 return isStartPositionValid(move.startPosition, move.endPosition) &&  
 ((this.getChangeX() == 0) || (this.getChangeY() == 0));  
 }  
  
}

public class Knight extends Piece {  
  
 Knight(PieceColor color, PiecePosition position) {  
 super(color, position);  
 }  
  
 @Override  
 boolean isValidMoveType(Move move) {  
 return isStartPositionValid(move.startPosition, move.endPosition) &&  
 ((Math.*abs*(this.getChangeX()) == 2 && Math.*abs*(this.getChangeY()) == 1) ||  
 (Math.*abs*(this.getChangeX()) == 1 && Math.*abs*(this.getChangeY()) == 2)  
 );  
 }  
  
}

public class Bishop extends Piece {  
  
 Bishop(PieceColor color, PiecePosition position) {  
 super(color, position);  
 }  
  
 @Override  
 boolean isValidMoveType(Move move) {  
 return isStartPositionValid(move.startPosition, move.endPosition) &&  
 (Math.*abs*(this.getChangeX()) == Math.*abs*(this.getChangeY()));  
 }  
  
}

public class Pawn extends Piece {  
 Pawn(PieceColor color, PiecePosition position) {  
 super(color, position);  
 }  
  
 @Override  
 boolean isValidMoveType(Move move) {  
 return isValidMoveY(move) && (this.getChangeX() == 0);  
 }  
  
 @Override  
 public boolean isValidTakeType(Move move) {  
 return isValidMoveY(move) && (Math.*abs*(this.getChangeX()) == 1);  
 }  
  
 public boolean isValidMoveY(Move move) {  
 boolean whiteMoves = (this.getColor() == PieceColor.*WHITE*) && (this.getChangeY() == -1);  
 boolean blackMoves = (this.getColor() == PieceColor.*BLACK*) && (this.getChangeY() == 1);  
  
 return isStartPositionValid(move.startPosition, move.endPosition) && (whiteMoves || blackMoves);  
 }  
}

public class EmptySpace extends Piece {  
  
 EmptySpace(PieceColor color, PiecePosition position) {  
 super(PieceColor.*EMPTY*, position);  
 }  
 @Override  
 boolean isValidMoveType(Move move) {  
 return false;  
 }  
 @Override  
 public void setPosition(PiecePosition position) { // does nothing }  
}

========================= Игроки ==================================

abstract class Player {  
 final private PieceColor color;  
  
 Player(PieceColor color) {  
 this.color = color;  
 }  
  
 abstract Move makeMove();  
  
 public PieceColor getColor(){  
 return color;  
 }  
}

public class User extends Player {  
  
 private String userName;  
 private int ELOrating;  
 private String teamName;  
  
 User(PieceColor color, String name, int rating, String team) {  
 super(color);  
 setName(name);  
 setRating(rating);  
 setTeam(team);  
 }  
  
 @Override  
 Move makeMove() {  
 return new Move();  
 }  
  
 public void setName(String name) {  
 this.userName = name;  
 }  
  
 public void setRating(int rating) {  
 this.ELOrating = rating;  
 }  
  
 public void setTeam(String team) {  
 this.teamName = team;  
 }  
  
 public String getName() {  
 return this.userName;  
 }  
  
 public int getRating() {  
 return this.ELOrating;  
 }  
  
 public String getTeam() {  
 return this.teamName;  
 }  
  
}

public class AI extends Player {  
  
 private int ELOrating;  
  
 AI(PieceColor color, int rating) {  
 super(color);  
 setRating(rating);  
 }  
  
 @Override  
 Move makeMove() {  
 return new Move();  
 }  
  
 public void setRating(int rating) {  
 this.ELOrating = rating;  
 }  
  
 public int getRating() {  
 return this.ELOrating;  
 }  
  
}

========================= Пересисляемые ==================================

public enum PieceColor {  
  
 *WHITE*,  
 *BLACK*,  
 *EMPTY*}

public enum State {  
  
 *WHITE\_MOVE*,  
 *BLACK\_MOVE*,  
 *WHITE\_IN\_CHECKMATE*,  
 *BLACK\_IN\_CHECKMATE*,  
 *STALEMATE*}

========================= Вспомогательные классы ==================================

public class PiecePosition {  
  
 private BoardBlock block;  
  
 PiecePosition(BoardBlock block) {  
 setPosition(block);  
 }  
 public void setPosition(BoardBlock block) {  
 if (isValidBlock(block)) {  
 this.block = block;  
 }  
 }  
  
 public BoardBlock getBlock() {  
 return this.block;  
 }  
  
 private boolean isValidBlock(BoardBlock block) {  
 return block.x > 0 && block.y > 0 && block.x < 8 && block.y < 8;  
 }  
}

public class BoardBlock {  
  
 public int x;  
 public int y; // white pieces are on y = 0, black pieces are on y = 7  
  
}

public class Move {  
  
 public PiecePosition startPosition;  
 public PiecePosition endPosition;  
}

========================= Доска ==================================

public class Board {  
  
 private Piece[][] blocks = new Piece[8][8];  
  
 Board(){  
 for (int i = 0; i < 8; i++) {  
 for (int j = 2; j < 5; j++) {  
 blocks[i][j] = new EmptySpace(PieceColor.*EMPTY*, new PiecePosition(getNewBlock(i, j)));  
 }  
 }  
 for (int i = 2; i < 5; i++) {  
 blocks[i][1] = new Pawn(PieceColor.*WHITE*, new PiecePosition(getNewBlock(i, 1)));  
 blocks[i][6] = new Pawn(PieceColor.*BLACK*, new PiecePosition(getNewBlock(i, 6)));  
 }  
 blocks[0][0] = new Rook(PieceColor.*WHITE*, new PiecePosition(getNewBlock(0, 0)));  
 blocks[7][0] = new Rook(PieceColor.*WHITE*, new PiecePosition(getNewBlock(7, 0)));  
 blocks[0][7] = new Rook(PieceColor.*BLACK*, new PiecePosition(getNewBlock(0, 7)));  
 blocks[7][7] = new Rook(PieceColor.*BLACK*, new PiecePosition(getNewBlock(7, 7)));  
  
 blocks[1][0] = new Knight(PieceColor.*WHITE*, new PiecePosition(getNewBlock(1, 0)));  
 blocks[6][0] = new Knight(PieceColor.*WHITE*, new PiecePosition(getNewBlock(6, 0)));  
 blocks[1][7] = new Knight(PieceColor.*BLACK*, new PiecePosition(getNewBlock(1, 7)));  
 blocks[6][7] = new Knight(PieceColor.*BLACK*, new PiecePosition(getNewBlock(6, 7)));  
  
 blocks[2][0] = new Bishop(PieceColor.*WHITE*, new PiecePosition(getNewBlock(2, 0)));  
 blocks[5][0] = new Bishop(PieceColor.*WHITE*, new PiecePosition(getNewBlock(5, 0)));  
 blocks[2][7] = new Bishop(PieceColor.*BLACK*, new PiecePosition(getNewBlock(2, 7)));  
 blocks[5][7] = new Bishop(PieceColor.*BLACK*, new PiecePosition(getNewBlock(5, 7)));  
  
 blocks[3][0] = new Queen(PieceColor.*WHITE*, new PiecePosition(getNewBlock(3, 0)));  
 blocks[4][0] = new Queen(PieceColor.*WHITE*, new PiecePosition(getNewBlock(4, 0)));  
 blocks[4][7] = new King(PieceColor.*BLACK*, new PiecePosition(getNewBlock(4, 7)));  
 blocks[3][7] = new King(PieceColor.*BLACK*, new PiecePosition(getNewBlock(3, 7)));  
 }  
  
 public void setBoard(Piece[][] figures) {  
 this.blocks = figures;  
 }  
  
 public Piece getPiece(BoardBlock block) {  
 return blocks[block.x][block.y];  
 }  
  
 public void setPiece(BoardBlock block, Piece figure) {  
 blocks[block.x][block.y] = figure;  
 }  
  
 private BoardBlock getNewBlock(int i, int j) {  
 BoardBlock block = new BoardBlock();  
 block.x = i;  
 block.y = j;  
 return block;  
 }  
  
}

========================= ИГРА ==================================

public class Game {  
 private Player whitePlayer;  
 private Player blackPlayer;  
 private Board board;  
 private State currentState;  
  
 Game(Player player1, Player player2, Board newBoard) {  
 whitePlayer = player1;  
 blackPlayer = player2;  
 board = newBoard;  
 currentState = State.*WHITE\_MOVE*;  
  
 Player currentPlayer;  
 Move currentMove;  
 Piece currentPiece;  
 BoardBlock startBlock, endBlock;  
  
 do {  
 if (currentState == State.*WHITE\_MOVE*) {  
 currentPlayer = whitePlayer;  
 } else {  
 currentPlayer = blackPlayer;  
 }  
 currentMove = currentPlayer.makeMove();  
 startBlock = currentMove.startPosition.getBlock();  
 endBlock = currentMove.endPosition.getBlock();  
 currentPiece = board.getPiece(startBlock);  
  
 if (currentPiece.isValidMoveType(currentMove) || currentPiece.isValidTakeType(currentMove)) {  
  
 currentPiece.setPosition(currentMove.endPosition);  
  
 if (board.getPiece(endBlock).getColor() == PieceColor.*EMPTY*) {  
 // если поле пустое то перемещаем фигуру  
 board.setPiece(startBlock, board.getPiece(endBlock));  
 board.setPiece(endBlock, currentPiece);  
 } else {  
 // если поле не пустое то съедаем фигуру на поле  
 board.setPiece(startBlock, new EmptySpace(PieceColor.*EMPTY*, currentMove.startPosition));  
 board.setPiece(endBlock, currentPiece);  
 }  
 board.getPiece(startBlock).setPosition(currentMove.endPosition);  
 board.getPiece(endBlock).setPosition(currentMove.startPosition);  
 }  
  
 // Дополнительные проверки и манипуляции  
 //  
  
 if (checkMove(currentPlayer, board)) {  
 // проверка на правльность хода  
 }  
 } while (isPositionValid(board) && gameGoOn(currentState));  
 }  
  
 public boolean checkMove(Player player, Board board) {  
 return true;  
 }  
  
 public boolean isPositionValid(Board board) {  
 return true;  
 }  
  
 public void changeGameState(Board board) {  
 // БЛОК ПРОЕКИ ПОЗМЦИИ НА ДОСКЕ И УСТАНОВКИ ОДНОГО ИЗ УКАЗАННЫХ СОСТОЯНИИ  
 // В СЛУЧАЕ НАЛИЧИЯ СООТВЕСТВУЮЩЕ ПОЗИЦИИ  
 // currentState == State.WHITE\_IN\_CHECKMATE || State.BLACK\_IN\_CHECKMATE || State.STALEMATE  
 }  
  
 if (currentState == State.*WHITE\_MOVE*) {  
 currentState = State.*BLACK\_MOVE*;  
 } else if (currentState == State.*BLACK\_MOVE*) {  
 currentState = State.*WHITE\_MOVE*;  
 }  
 }  
  
 private boolean gameGoOn(State state) {  
 return state == State.*WHITE\_MOVE* ||  
 state == State.*BLACK\_MOVE*;  
 }  
}