import java.util.ArrayList;
import java.util.List;

/**
 * The class {@code Othello} represents the game Othello itself.
 * All game logic is done in this class.
 *
 * The Java Doc can be found here:
 * <a href="http://www.martin-thoma.de/programmieren-othello-1adf234d3fS/">
 * martin-thoma.de/programmieren-othello-1adf234d3fS</a>
 *
 * @author Martin Thoma
 */

public class Othello {

    /** Error message: a player already moved */
    public static final String ERR_PLAYER_MOVED
        = "Cannot add hole area. A player did move.";

    /** Error message: no active game */
    public static final String ERR_NO_ACTIVE_GAME = "No active game.";

    /** Error message: the move target isn’t on the board */
    public static final String ERR_OFFBOARD_MOVE
        = "The move position has to be on the board.";

    /** Error message: a color is in the for a hole specified rectangle*/
    public static final String ERR_COLOR_IN_RECTANGLE
        = "You can’t place the hole here. There are color pieces.";

    /** Error message: the specified rectangle isn’t valid */
    public static final String ERR_NO_VALID_RECTANGLE
        = "The specified rectangle isn’t valid. " + "Valid is something like A1:B3 or A1:A1. The first position "; + "has to be on the top left.";

    /** The current player. Always start with black. */
    private Field currentPlayer = Field.BLACK;

    /** Is the current game still in progress? */
    private boolean isRunning = true;

    /** Has already a move command been submitted? */
    private boolean submittedMove = false;

    /** The board with all pieces */
    public final Board board;

    private final int[][] adjactantFields = {{-1, -1}, {0, -1}, {1, -1},

/**
 * Constructor for Othello.
 * It is possible, that the game is finished as soon as it is created.
 * @param width the width of the board
 * @param height the height of the board
 */
public Othello(int width, int height) {
    this.board = new Board(width, height);
    checkState();
}

/**
 * Constructor for Othello with a given start situation.
 * It is possible, that the game is finished as soon as it is created.
 * @param width the width of the board
 * @param height the height of the board
 * @param situation the situation the player wants to start with
 */
public Othello(int width, int height, String situation) {
    this.board = new Board(width, height, situation);
    checkState();
}

/**
 * Checks for all constructors if black can make a move.
 * If black can’t it’s the turn of white. If white can’t move either,
 * the game is finished.
 */
private void checkState() {
    if (!isMovePossible(Field.BLACK)) {
        if (!isMovePossible(Field.WHITE)) {
            // if no moves are possible, the game is instantly finished
            this.isRunning = false;
        } else {
            // if black can’t move but white can, it’s white’s turn
            this.currentPlayer = Field.WHITE;
        }
    }
}

/**
 * This method checks if any move is possible for player
 * @param player the color of the player you want to check
 * @return {code true} if any move is possible,
 * otherwise {code false}
 */
private boolean isMovePossible(Field player) {
    return (getPossibleMoves(player).size() > 0);
```java
/**
 * Get a list of all possible moves.
 * @param player the player whose possible moves you want to get
 * @return a list of all possible moves
 */
public List<Position> getPossibleMoves(Field player) {
    if (!isRunning) {
        throw new IllegalStateException(ERR_NO_ACTIVE_GAME);
    }

    List<Position> possibleMoves = new ArrayList<Position>();
    Position pos;
    for (int x = 0; x < board.width; x++) {
        for (int y = 0; y < board.height; y++) {
            pos = new Position(x, y);
            if (isMovePositionValid(pos)
                    && (getNrOfSwitches(player, pos) > 0)) {
                possibleMoves.add(pos);
            }
        }
    }

    return possibleMoves;
}

/**
 * Checks if a position on the board has a color.
 * If the position is not valid (e.g. negative array index) it
 * returns {code false}.
 * @param pos the position you want to check
 * @return {code true} if a color is at this position,
 * otherwise {code false}
 */
private boolean hasPiece(Position pos) {
    boolean returnVal = false;

    if (board.isPositionOnBoard(pos) && board.get(pos) != null
            && board.get(pos) != Field.HOLE) {
        returnVal = true;
    }

    return returnVal;
}

/**
 * Check if a move position is valid. This checks if the position
 * exists on the board, if it is empty and if a piece is adjacent.
 */
```
* @param pos the position you want to check
* @return {@code true} if the move position can be valid,
* otherwise {@code false}
* /
private boolean isMovePositionValid(Position pos) {
    boolean isMovePositionValid = false;

    if (!board.isPositionOnBoard(pos)) {
        return false;
    }

    for (int[] field : adjactantFields) {
        Position tmp = new Position(pos.x + field[0],
                                    pos.y + field[1]);
        if (hasPiece(tmp)) {
            isMovePositionValid = true;
        }
    }

    if (board.get(pos.x, pos.y) != null) {
        // a piece is already on the field
        isMovePositionValid = false;
    }

    return isMovePositionValid;
}

/**
* Set the current player to the next player.
* /
private void nextPlayer() {
    if (!isRunning) {
        throw new IllegalArgumentException(ERR_NO_ACTIVE_GAME);
    }

    if (currentPlayer == Field.BLACK) {
        currentPlayer = Field.WHITE;
    } else {
        currentPlayer = Field.BLACK;
    }
}

/**
* Make a move, if possible and return a code that indicates what
* happened.
* @param pos the position you want to set the next piece on
* @return 0 if the player could move,
* -1 if the player could not move,
* 1 if the next regular player had to pass,
* 2 if the game ended with this move
* /

public int move(Position pos) {
    if (!isRunning) {
        throw new IllegalStateException(ERR_NO_ACTIVE_GAME);
    }

    int returnCode = -1;
    int switches;

    if (!board.isPositionOnBoard(pos)) {
        throw new IllegalArgumentException(ERR_OFFBOARD_MOVE);
    }

    if (isMovePositionValid(pos) && (getNrOfSwitches(currentPlayer, pos) > 0)) {
        board.set(pos, currentPlayer);

        // switch all pieces in between
        for (int[] direction: adjactantFields) {
            switches = getNrOfIncludedPieces(currentPlayer, pos,
                direction[0], direction[1]);
            if (switches > 0) {
                switchPieces(currentPlayer, pos, direction[0], direction[1]);
            }
        }

        // switch to the next player
        nextPlayer();

        if (!isMovePossible(getCurrentPlayer())) {
            Field nextPlayer = getWaitingPlayer();
            if (isMovePossible(nextPlayer)) {
                nextPlayer();
                returnCode = 1;
            } else {
                setFinished();
                returnCode = 2;
            }
        } else {
            returnCode = 0;
        }

        submittedMove = true;
    }

    return returnCode;
}

/*
 * Get the current player.
 */
* @return the current player

*/

public Field getCurrentPlayer() {
    return currentPlayer;
}

/**
 * This method determines the number of pieces of the opponent
 * between the given position and the next piece of the given player.
 * @param player The player.
 * @param pos the position of one piece of this player.
 * @param xDir this has to be 1, 0 or -1.
 *       1 means it goes to the right, -1 to the left.
 *       0 means it doesn’t change the x-direction.
 * @param yDir this has to be 1, 0 or -1.
 *       1 means it goes to the bottom, -1 to the top.
 *       0 means it doesn’t change the y-direction.
 * @return the number of pieces of the opponent between the given position
 * and the next piece of the given player.
 */

private int getNrOfIncludedPieces(Field player, Position pos, int xDir, int yDir) {
    int switches = 0;
    int opponentCount = 0;
    Field opponent = (player == Field.WHITE ? Field.BLACK : Field.WHITE);

    for (int tmp = 1;
         (pos.x + tmp * xDir >= 0)   // important if you go to the left
         && (pos.x + tmp * xDir < board.width) // important if you go to the right
         && (pos.y + tmp * yDir >= 0)   // important if you go to the bottom
         && (pos.y + tmp * yDir < board.height); // important if you go to the top
         tmp++) {

        Field piece = board.get(pos.x + tmp * xDir, pos.y + tmp * yDir);

        if (piece == player) {
            switches += opponentCount;
            opponentCount = 0;
            break;
        } else if (piece == Field.HOLE) {
            return 0;
        } else if (piece == opponent) {
            opponentCount++;
        } else if (piece == null) {
            return 0;
        }
    }

    return switches;
/**
 * Switch all pieces from the opponent of player in the given direction.
 * Make sure that in the given direction is one of the pieces of player at the end.
 * @param player the given player who set the new piece
 * @param pos the position where you want to start
 * @param xDir one part of the direction
 * @param yDir other part of the direction
 */
private void switchPieces(Field player, Position pos, int xDir, int yDir) {
    if (!isRunning) {
        throw new IllegalStateException(ERR_NO_ACTIVE_GAME);
    }

    Field opponent = (player == Field.WHITE ? Field.BLACK : Field.WHITE);

    // this ends always with the break as one piece of player has to be at the end
    for (int tmp = 1;; tmp++) {
        if (board.get(pos.x + tmp * xDir, pos.y + tmp * yDir) == player) {
            break;
        } else if (board.get(pos.x + tmp * xDir, pos.y + tmp * yDir) == opponent) {
            board.set(pos.x + tmp * xDir, pos.y + tmp * yDir, player);
        }
    }
}

/**
 * Return the number of pieces that get switched when player sets
 * a new piece on (x|y)
 * @param player the given player
 * @param pos the position of the new piece
 * @return the number of switched pieces.
 */
private int getNrOfSwitches(Field player, Position pos) {
    int switches = 0;

    for (int[] direction : adjaclantFields) {
        switches += getNrOfIncludedPieces(player, pos, direction[0], direction[1]);
    }

    return switches;
}

/**
 * Return the result.
 * @return an array with two elements where the first element
 * represents the points
 * of the white player and the second element the points of the second player
 */
public int[] getResult() {
    int[] result = new int[2];
    result[0] = countPieces(Field.WHITE);
    result[1] = countPieces(Field.BLACK);
    return result;
}

// this method counts the pieces of one player on the board
private int countPieces(Field player) {
    int counter = 0;
    for (int x = 0; x < board.width; x++) {
        for (int y = 0; y < board.height; y++) {
            if (board.get(x, y) == player) {
                counter++;
            }
        }
    }
    return counter;
}

/**
 * Mark the game as finished.
 */
public void setFinished() {
    if (!isRunning) {
        throw new IllegalStateException(ERR_NO_ACTIVE_GAME);
    }
    isRunning = false;
}

/**
 * Getter for isRunning.
 * @return {@code true} if the game is still in progress,
 * otherwise {@code false}
 */
public boolean isRunning() {
    return isRunning;
}

/**
 * Checks if the rectangle is within the borders of the board and
 * if the first position is at the top left and the second is at
 * the bottom right.
 * @param rectangle the rectangle
 * @return {@code true} if the rectangle is valid according to the
 * specification, otherwise {@code false}
 */
public boolean isValidRectangle(Position[] rectangle) {
    if (!board.isPositionOnBoard(rectangle[0])) {
return false;
} else if (!board.isPositionOnBoard(rectangle[1])) {
    return false;
} else if (rectangle[0].x > rectangle[1].x) {
    return false;
} else if (rectangle[0].y > rectangle[1].y) {
    return false;
} else {
    return true;
}
}

/**
 * Check if a piece is in the specified rectangle.
 * @param rectangle the specified rectangle
 * @return {@code true} if a piece is in the specified rectangle,
 * otherwise {@code false}
 */
public boolean isColorInRectangle(Position[] rectangle) {
    if (!isValidRectangle(rectangle)) {
        throw new IllegalArgumentException(ERR_NO_VALID_RECTANGLE);
    }

    for (int x = rectangle[0].x; x <= rectangle[1].x; x++) {
        for (int y = rectangle[0].y; y <= rectangle[1].y; y++) {
            if (board.get(x, y) == Field.BLACK || board.get(x, y) == Field.WHITE) {
                return true;
            }
        }
    }

    return false;
}

/**
 * Make an hole into the board if possible.
 * @param rectangle The edges of the rectangle of the hole
 * @return {@code true} if a hole could be created, otherwise {@code false}
 */
public boolean makeHole(Position[] rectangle) {
    if (submittedMove) {
        throw new IllegalStateException(ERR_PLAYER_MOVED);
    } else if (!isValidRectangle(rectangle)) {
        throw new IllegalArgumentException(ERR_NO_VALID_RECTANGLE);
    } else if (isColorInRectangle(rectangle)) {
        throw new IllegalArgumentException(ERR_COLOR_IN_RECTANGLE);
    }

    for (int x = rectangle[0].x; x <= rectangle[1].x; x++) {
        for (int y = rectangle[0].y; y <= rectangle[1].y; y++) {
            board.set(x, y, Field.EMPTY);
        }
    }
}

```java
        board.set(x, y, Field.HOLE);
    }
}

    // Switch to the other player if the current player can't move any longer
    if (getPossibleMoves(currentPlayer).size() == 0) {
        nextPlayer();
    }

    return true;
}

/**
 * Was a move already submitted?
 * @return {@code true} if a move was already submitted, otherwise {@code false}
 */
public boolean wasMoveSubmitted() {
    return submittedMove;
}

/**
 * This method aborts the current game and returns the result.
 * @return the result as an int array with two elements where {@code result[0]} represents the points of the white player and {@code result[1]} represents the points of the black player
 */
public int[] abortGame() {
    int[] result = getResult();
    setFinished();
    return result;
}

/**
 * Get the player who can't make a turn by now.
 * @return the player who can't make a turn by now
 */
public Field getWaitingPlayer() {
    return getCurrentPlayer() == Field.BLACK ? Field.WHITE : Field.BLACK;
}
```