```java
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},
                        {-1, 0}, {1, 0}, {-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
     */
```

```
58      public Othello(int width, int height) {
59          this.board = new Board(width, height);
60          checkState();
61      }
62
63      /**
64       * Constructor for Othello with a given start situation.
65       * It is possible, that the game is finished as soon as it is created.
66       * @param width the width of the board
67       * @param height the height of the board
68       * @param situation the situation the player wants to start with
69       */
70      public Othello(int width, int height, String situation) {
71          this.board = new Board(width, height, situation);
72          checkState();
73      }
74
75      /**
76       * Checks for all constructors if black can make a move.
77       * If black can't it's the turn of white. If white can't move either,
78       * the game is finished.
79       */
80      private void checkState() {
81          if (!isMovePossible(Field.BLACK)) {
82              if (!isMovePossible(Field.WHITE)) {
83                  // if no moves are possible, the game is instantly finished
84                  this.isRunning = false;
85              } else {
86                  // if black can't move but white can, it's whites turn
87                  this.currentPlayer = Field.WHITE;
88              }
89          }
90      }
91
92      /**
93       * This method checks if any move is possible for player
94       * @param player the color of the player you want to check
95       * @return {@code true} if any move is possible,
96       *         otherwise {@code false}
97       */
98      private boolean isMovePossible(Field player) {
99          return (getPossibleMoves(player).size() > 0);
100     }
101
102     /**
103      * Get a list of all possible moves.
104      * @param player the player whose possible moves you want to get
105      * @return a list of all possible moves
106      */
107     public List<Position> getPossibleMoves(Field player) {
108         if (!isRunning) {
109             throw new IllegalStateException(ERR_NO_ACTIVE_GAME);
110         }
111
112         List<Position> possibleMoves = new ArrayList<Position>();
113
114         Position pos;
115         for (int x = 0; x < board.width; x++) {
```

```
116             for (int y = 0; y < board.height; y++) {
117                 pos = new Position(x, y);
118                 if (isMovePositionValid(pos)
119                     && (getNrOfSwitches(player, pos) > 0)) {
120                     possibleMoves.add(pos);
121                 }
122             }
123         }
124
125         return possibleMoves;
126     }
127
128     /**
129      * Checks if a position on the board has a color.
130      * If the position is not valid (e.g. negative array index) it
131      * returns {@code false}.
132      * @param pos the position you want to check
133      * @return {@code true} if a color is at this position,
134      *          otherwise {@code false}
135      */
136     private boolean hasPiece(Position pos) {
137         boolean returnVal = false;
138
139         if (board.isPositionOnBoard(pos) && board.get(pos) != null
140             && board.get(pos) != Field.HOLE) {
141             returnVal = true;
142         }
143
144         return returnVal;
145     }
146
147     /**
148      * Check if a move position is valid. This checks if the position
149      * exists on the board, if it is empty and if a piece is adjacent.
150      * @param pos the position you want to check
151      * @return {@code true} if the move position can be valid,
152      *         otherwise {@code false}
153      */
154     private boolean isMovePositionValid(Position pos) {
155         boolean isMovePositionValid = false;
156
157         if (!board.isPositionOnBoard(pos)) {
158             return false;
159         }
160
161         for (int[] field : adjactantFields) {
162             Position tmp = new Position(pos.x + field[0],
163                                         pos.y + field[1]);
164             if (hasPiece(tmp)) {
165                 isMovePositionValid = true;
166             }
167         }
168
169         if (board.get(pos.x, pos.y) != null) {
170             // a piece is already on the field
171             isMovePositionValid = false;
172         }
173
```

```
174         return isMovePositionValid;
175     }
176
177     /**
178      * Set the current player to the next player.
179      */
180     private void nextPlayer() {
181         if (!isRunning) {
182             throw new IllegalStateException(ERR_NO_ACTIVE_GAME);
183         }
184
185         if (currentPlayer == Field.BLACK) {
186             currentPlayer = Field.WHITE;
187         } else {
188             currentPlayer = Field.BLACK;
189         }
190     }
191
192     /**
193      * Make a move, if possible and return a code that indicates what
194      * happened.
195      * @param pos the position you want to set the next piece on
196      * @return 0 if the player could move,
197      * -1 if the player could not move,
198      * 1 if the next regular player had to pass,
199      * 2 if the game ended with this move
200      */
201     public int move(Position pos) {
202         if (!isRunning) {
203             throw new IllegalStateException(ERR_NO_ACTIVE_GAME);
204         }
205
206         int returnCode = -1;
207         int switches;
208
209         if (!board.isPositionOnBoard(pos)) {
210             throw new IllegalArgumentException(ERR_OFFBOARD_MOVE);
211         }
212
213         if (isMovePositionValid(pos)
214             && (getNrOfSwitches(currentPlayer, pos) > 0)) {
215             board.set(pos, currentPlayer);
216
217             // switch all pieces in between
218             for (int[] direction: adjactantFields) {
219                 switches = getNrOfIncludedPieces(currentPlayer, pos,
220                                                  direction[0], direction[1]);
221                 if (switches > 0) {
222                     switchPieces(currentPlayer, pos, direction[0],
                            direction[1]);
223                 }
224             }
225
226             // switch to the next player
227             nextPlayer();
228
229             if (!isMovePossible(getCurrentPlayer())) {
230                 Field nextPlayer = getWaitingPlayer();
```

```
231                 if (isMovePossible(nextPlayer)) {
232                     nextPlayer();
233                     returnCode = 1;
234                 } else {
235                     setFinished();
236                     returnCode = 2;
237                 }
238             } else {
239                 returnCode = 0;
240             }
241 
242             submittedMove = true;
243         }
244 
245         return returnCode;
246     }
247 
248     /**
249      * Get the current player.
250      * @return the current player
251      */
252     public Field getCurrentPlayer() {
253         return currentPlayer;
254     }
255 
256     /**
257      * This method determines the number of pieces of the opponent
258      * between the given position and the next piece of the given player.
259      * @param player The player.
260      * @param pos the position of one piece of this player.
261      * @param xDir this has to be 1, 0 or -1.
262      *             1 means it goes to the right, -1 to the left.
263      *             0 means it doesn't change the x-direction.
264      * @param yDir this has to be 1, 0 or -1.
265      *             1 means it goes to the bottom, -1 to the top.
266      *             0 means it doesn't change the y-direction.
267      * @return the number of pieces of the opponent between the given
         position
268      * and the next piece of the given player.
269      */
270 
271     private int getNrOfIncludedPieces(Field player, Position pos, int xDir,
         int yDir) {
272         int switches = 0;
273         int opponentCount = 0;
274         Field opponent = (player == Field.WHITE ? Field.BLACK : Field.WHITE
            );
275 
276         for (int tmp = 1;
277             // stop the loop if you're no longer on the board
278             (pos.x + tmp * xDir >= 0)    // important if you go to the left
279             && (pos.x + tmp * xDir < board.width) // important if you go to
                 the right
280             && (pos.y + tmp * yDir >= 0) // important if you go to the
                bottom
281             && (pos.y + tmp * yDir < board.height); // important if you go
                to the top
282             tmp++) {
```

```
283
284             Field piece = board.get(pos.x + tmp * xDir, pos.y + tmp * yDir)
                    ;
285
286             if (piece == player) {
287                 switches += opponentCount;
288                 opponentCount = 0;
289                 break;
290             } else if (piece == Field.HOLE) {
291                 return 0;
292             } else if (piece == opponent) {
293                 opponentCount++;
294             } else if (piece == null) {
295                 return 0;
296             }
297         }
298
299         return switches;
300     }
301
302     /**
303      * Switch all pieces from the opponent of player in the given direction
                .
304      * Make sure that in the given direction is one of the pieces of player
                at the end.
305      * @param player the given player who set the new piece
306      * @param pos the position where you want to start
307      * @param xDir one part of the direction
308      * @param yDir other part of the direction
309      */
310     private void switchPieces(Field player, Position pos, int xDir, int
            yDir) {
311         if (!isRunning) {
312             throw new IllegalStateException(ERR_NO_ACTIVE_GAME);
313         }
314
315         Field opponent = (player == Field.WHITE ? Field.BLACK : Field.WHITE
                );
316
317         // this ends always with the break as one piece of player has to be
                at the end
318         for (int tmp = 1;; tmp++) {
319             if (board.get(pos.x + tmp * xDir, pos.y + tmp * yDir) == player
                    ) {
320                 break;
321             } else if (board.get(pos.x + tmp * xDir, pos.y + tmp * yDir) ==
                    opponent) {
322                 board.set(pos.x + tmp * xDir, pos.y + tmp * yDir, player);
323             }
324         }
325     }
326
327     /**
328      * Return the number of pieces that get switched when player sets
329      * a new piece on (x|y)
330      * @param player the given player
331      * @param pos the position of the new piece
332      * @return the number of switched pieces.
```

```
333         */
334        private int getNrOfSwitches(Field player, Position pos) {
335            int switches = 0;
336
337            for (int[] direction : adjactantFields) {
338                switches += getNrOfIncludedPieces(player, pos, direction[0],
                       direction[1]);
339            }
340
341            return switches;
342        }
343
344        /**
345         * Return the result.
346         * @return an array with two elements where the first element
347         *         represents the points
348         * of the white player and the second element the points of the second
               player
349         */
350        public int[] getResult() {
351            int[] result = new int[2];
352            result[0] = countPieces(Field.WHITE);
353            result[1] = countPieces(Field.BLACK);
354            return result;
355        }
356
357        // this method counts the pieces of one player on the board
358        private int countPieces(Field player) {
359            int counter = 0;
360            for (int x = 0; x < board.width; x++) {
361                for (int y = 0; y < board.height; y++) {
362                    if (board.get(x, y) == player) {
363                        counter++;
364                    }
365                }
366            }
367            return counter;
368        }
369
370        /**
371         * Mark the game as finished.
372         */
373        public void setFinished() {
374            if (!isRunning) {
375                throw new IllegalStateException(ERR_NO_ACTIVE_GAME);
376            }
377
378            isRunning = false;
379        }
380
381        /**
382         * Getter for isRunning.
383         * @return {@code true} if the game is still in progress,
384         *         otherwise {@code false}
385         */
386        public boolean isRunning() {
387            return isRunning;
388        }
```

```
389
390     /**
391      * Checks if the rectangle is within the borders of the board and
392      * if the first position is at the top left and the second is at
393      * the bottom right.
394      * @param rectangle the rectangle
395      * @return {@code true} if the rectangle is valid according to the
396      *         specification, otherwise {@code false}
397      */
398     public boolean isValidRectangle(Position[] rectangle) {
399         if (!board.isPositionOnBoard(rectangle[0])) {
400             return false;
401         } else if (!board.isPositionOnBoard(rectangle[1])) {
402             return false;
403         } else if (rectangle[0].x > rectangle[1].x) {
404             return false;
405         } else if (rectangle[0].y > rectangle[1].y) {
406             return false;
407         } else {
408             return true;
409         }
410     }
411
412     /**
413      * Check if a piece is in the specified rectangle.
414      * @param rectangle the specified rectangle
415      * @return {@code true} if a piece is in the specified rectangle,
416      *         otherwise {@code false}
417      */
418     public boolean isColorInRectangle(Position[] rectangle) {
419         if (!isValidRectangle(rectangle)) {
420             throw new IllegalArgumentException(ERR_NO_VALID_RECTANGLE);
421         }
422
423         for (int x = rectangle[0].x; x <= rectangle[1].x; x++) {
424             for (int y = rectangle[0].y; y <= rectangle[1].y; y++) {
425                 if (board.get(x, y) == Field.BLACK || board.get(x, y) ==
                        Field.WHITE) {
426                     return true;
427                 }
428             }
429         }
430
431         return false;
432     }
433
434     /**
435      * Make an hole into the board if possible.
436      * @param rectangle The edges of the rectangle of the hole
437      * @return {@code true} if a hole could be created, otherwise {@code
            false}
438      */
439     public boolean makeHole(Position[] rectangle) {
440         if (submittedMove) {
441             throw new IllegalStateException(ERR_PLAYER_MOVED);
442         } else if (!isValidRectangle(rectangle)) {
443             throw new IllegalArgumentException(ERR_NO_VALID_RECTANGLE);
444         } else if (isColorInRectangle(rectangle)) {
```

```
445                 throw new IllegalArgumentException(ERR_COLOR_IN_RECTANGLE);
446             }
447 
448             for (int x = rectangle[0].x; x <= rectangle[1].x; x++) {
449                 for (int y = rectangle[0].y; y <= rectangle[1].y; y++) {
450                     board.set(x, y, Field.HOLE);
451                 }
452             }
453 
454             // Switch to the other player if the current player can't move any
                    longer
455             if (getPossibleMoves(currentPlayer).size() == 0) {
456                 nextPlayer();
457             }
458 
459             return true;
460         }
461 
462         /**
463          * Was a move already submitted?
464          * @return {@code true} if a move was already submitted, otherwise {
                    @code false}
465          */
466         public boolean wasMoveSubmitted() {
467             return submittedMove;
468         }
469 
470         /**
471          * This method aborts the current game and returns the result.
472          * @return the result as an int array with two elements where {@code
                    result[0]}
473          * represents the points of the white player and {@code result[1]}
                    represents the
474          * points of the black player
475          */
476         public int[] abortGame() {
477             int[] result = getResult();
478             setFinished();
479             return result;
480         }
481 
482         /**
483          * Get the player who can't make a turn by now.
484          * @return the player who can't make a turn by now
485          */
486         public Field getWaitingPlayer() {
487             return getCurrentPlayer() == Field.BLACK ? Field.WHITE : Field.
                    BLACK;
488         }
489 }
```

Othello.java