```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package Game;

import Pieces.King;
import Pieces.Pawn;
import Pieces.Piece;
import Pieces.PieceType;
import Pieces.Rook;

/**
 * This class manages the full game state containing both players, the board,
 * and the turn order to ensure the game operates according to the game rules.
 *
 * @author Ep16fb
 */
public class Game {

    private final Player white;
    private final Player black;
    private Board currentBoard;
    private Colour currentTurn;

    Piece promotionTo;
    boolean castleKingSide;

    public Game() {
        white = new Player(Colour.White);
        black = new Player(Colour.Black);
        currentBoard = new Board();
        currentTurn = Colour.White;
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                Piece piece = currentBoard.getBoard()[i][j];
                if (piece != null) {
                    if (piece.colour == Colour.White) {
                        white.setupAttacks(currentBoard, i, j);
                    }
                    if (piece.colour == Colour.Black) {
                        black.setupAttacks(currentBoard, i, j);
                    }
                }
            }
        }
    }

    public Player getWhite() {
        return white;
    }

    public Player getBlack() {
        return black;
    }

    public Colour getCurrentTurn() {
        return currentTurn;
    }

    public Player getOpponent() {
        return currentTurn == Colour.White ? black : white;
    }

    public Board getBoard() {
        return currentBoard;
    }

```

```
70      /**
71       * This method retrieves a string that has been inputted by a user and
72       * determines the first and second part of the move by removing invalid
73       * characters and then splitting the string.
74       *
75       * @param userInput
76       */
77      public void parseUserInput(String userInput) {
78          String columns = userInput.replaceAll("[^a-g]", "");
79          String rows = userInput.replaceAll("[^1-8]", "");
80          int startC = Board.boardToIndexC(columns.charAt(0));
81          int startR = Character.getNumericValue(rows.charAt(0));
82          int nextC = Board.boardToIndexC(columns.charAt(1));
83          int nextR = Character.getNumericValue(rows.charAt(1));
84          nextBoard(startC, startR, nextC, nextR);
85      }
86
87      /**
88       * Depreciated
89       *
90       * @param log
91       * @return
92       */
93      public Board nextBoard(String log) {
94          //exf8=Q+
95          // Parse log input
96          // Check if piece colour matches piece colour on board
97          // Check if piece type matches piece type on board
98          // Convert boardX and boardY into usable indecies for array access
99          // Call other nextBoard method
100         return currentBoard;
101     }
102
103     /**
104      * See nextBoard(int startR, int startC, int nextR, int nextC)
105      *
106      * @param move
107      * @return
108      */
109     public Board nextBoard(Move move) {
110         return nextBoard(move.startR, move.startC, move.nextR, move.nextC);
111     }
112
113     /**
114      * This method checks that the piece moved on the board is valid and then
115      * applies the move and changes the board state, as there is no other
116      * situation the board needs to be overridden.
117      *
118      * @param startR
119      * @param startC
120      * @param nextR
121      * @param nextC
122      * @return
123      */
124     public Board nextBoard(int startR, int startC, int nextR, int nextC) {
125         Piece toMove = currentBoard.getBoard()[startR][startC];
126         Board next;
127         if (currentTurn == Colour.White) {
128             // ensure a move is not applied on wrong turn
129             if (toMove == null || toMove.colour == Colour.Black) {
130                 return currentBoard;
131             }
132             // will check if move is valid, otherwise does nothing
133             next = white.movePiece(black, currentBoard, startR, startC, nextR, nextC,
                promotionTo);
134             if (currentBoard.equals(next)) {
135                 return currentBoard;
136             }
137             //setBoard(next);
```

```
138             return next;
139         } else {
140             if (toMove == null || toMove.colour == Colour.White) {
141                 return currentBoard;
142             }
143             // will check if move is valid, otherwise does nothing
144             next = black.movePiece(white, currentBoard, startR, startC, nextR, nextC,
                promotionTo);
145             if (currentBoard.equals(next)) {
146                 return currentBoard;
147             }
148             //setBoard(next);
149             return next;
150         }
151     }
152 
153     /**
154      * This ensured that a log defining a proper order of actions is used to
155      * ensure that the board is undone in the reverse order it was applied See
156      * undoMove(Move move).
157      *
158      * @param moveStack
159      * @return
160      */
161     public Board undoMove(Log moveStack) {
162         Move toUndo = moveStack.undoMove();
163         return undoMove(toUndo);
164     }
165 
166     /**
167      * This takes the move and reverts the action applied on the board and
168      * replaces the previous position with a piece it may have captured, and it
169      * also uses the log to check if any special action was performed.
170      *
171      * @param move
172      * @return
173      */
174     public Board undoMove(Move move) {
175         Board previousBoard = new Board(currentBoard);
176         Piece previous = previousBoard.getBoard()[move.nextR][move.nextC];
177         int row = previous.colour == Colour.White ? 7 : 0;
178         // checks if castle (queen side) was performed (O-O-O) not optional
179         if
            (move.getLog().matches("[A-Ga-g1-8BKNPQR]+(x)*[A-Ga-g1-8BKNPQR]+(O-O-O)(=[BNQR])*
            [+#]*")) {
180             previousBoard.getBoard()[row][2] = null; // king 'next'
181             previousBoard.getBoard()[row][3] = null; // rook 'next'
182             previousBoard.getBoard()[row][4] = new King(previous.colour); // king
                previous
183             previousBoard.getBoard()[row][0] = new Rook(previous.colour); // rook
                previous
184             return previousBoard;
185         } // check if castle (king side) was performed (O-O) not optional
186         else if
            (move.getLog().matches("[A-Ga-g1-8BKNPQR]+(x)*[A-Ga-g1-8BKNPQR]+(O-O)(=[BNQR])*[+
            #]*")) {
187             previousBoard.getBoard()[row][6] = null; // king 'next'
188             previousBoard.getBoard()[row][5] = null; // rook 'next'
189             previousBoard.getBoard()[row][4] = new King(previous.colour); // king
                previous
190             previousBoard.getBoard()[row][7] = new Rook(previous.colour); // rook
                previous
191             return previousBoard;
192         } // checks if promotion was performed (=[BNQR]) not optional
193         else if
            (move.getLog().matches("[A-Ga-g1-8BKNPQR]+(x)*[A-Ga-g1-8BKNPQR]+(O-O|O-O-O)*(=[BN
            QR])[+#]*")) {
194             previous = new Pawn(previous.colour);
195         }
```

```
196             previousBoard.getBoard()[move.startR][move.startC] = previous;
197             previousBoard.getBoard()[move.nextR][move.nextC] = move.getCaptured();
198             setBoard(previousBoard); // will also change currentTurn colour
199             System.out.println("Undo Complete!");
200             return previousBoard;
201         }
202 
203         public void setBoard(Board board) {
204             // Ensure the next board is changed using nextBoard before setting
205             currentBoard = board;
206         }
207 
208         public boolean changeTurn() {
209             // changes turn to next player
210             if (currentTurn == Colour.Black) {
211                 currentTurn = Colour.White;
212             } else if (currentTurn == Colour.White) {
213                 currentTurn = Colour.Black;
214             }
215             return true;
216         }
217 
218         /**
219          * This checks if a king has been captured or if the king cannot move and is
220          * in check or stalemate to end the game.
221          *
222          * @return
223          */
224         public boolean isGameEnd() {
225             boolean gameOver = white.getLoss() || black.getLoss();
226             if (gameOver) {
227                 currentBoard.printToLogfinalOutcome(currentTurn);
228                 return gameOver;
229             }
230             Player opponent = currentTurn == Colour.White ? black : white;
231             Piece toExamine;
232             boolean kingMove = false; // default cannot move
233             boolean otherMove = false; // default cannot move
234             boolean kingThreatened = false;
235             for (int i = 0; i < 8; i++) {
236                 for (int j = 0; j < 8; j++) {
237                     // check to see if there is a piece that can move
238                     toExamine = currentBoard.getBoard()[i][j];
239                     if (toExamine != null) {
240                         if (toExamine.colour == currentTurn) {
241                             toExamine.validMoves(opponent, currentBoard, i, j);
242                             if (toExamine.piece == PieceType.King) {
243                                 kingMove = toExamine.getCanMove();
244                                 if (toExamine.isThreatened(currentBoard, i, j) > 0) {
245                                     kingThreatened = true;
246                                 }
247                             } else {
248                                 otherMove = toExamine.getCanMove();
249                             }
250                             if (otherMove == true || kingMove == true) {
251                                 break;
252                             }
253                         }
254                     }
255                 }
256             }
257             // check if it is checkmate
258             if (!kingMove && kingThreatened) {
259                 currentBoard.printToLogfinalOutcome(currentTurn);
260                 return true;
261             }
262             // stalemate occurs if both false
263             gameOver = kingMove == false && otherMove == false;
264             if (gameOver) {
```

```
265                 currentBoard.printToLogfinalOutcome(null);
266             }
267             return gameOver;
268         }
269 
270     }
271 
```