```java
package Pieces;

import Game.Board;
import Game.Colour;
import Game.Player;

/**
 * This class represents the Queen piece
 *
 * @author Param
 */
public class Queen extends Piece {

    public boolean canMove;

    public Queen(Colour colour) {
        super(PieceType.Queen, colour, 9);
    }

    @Override
    public int threats(Board board, int row, int column) {
        Piece[][] currentBoard = board.getBoard();
        Piece toExamine;
        int threatened = 0;
        // check up
        if (row > 0) {
            for (int x = row - 1; x >= 0; x--) {
                toExamine = currentBoard[x][column];
                if (toExamine != null) {
                    if (this.isOppositeColour(toExamine)) {
                        threatened += toExamine.weight;
                    }
                    break;
                }
            }
        }
        // check down
        if (row < 7) {
            for (int x = row + 1; x <= 7; x++) {
                toExamine = currentBoard[x][column];
                if (toExamine != null) {
                    if (this.isOppositeColour(toExamine)) {
                        threatened += toExamine.weight;
                    }
                    break;
                }
            }
        }
        // check left
        if (column > 0) {
            for (int y = column - 1; y >= 0; y--) {
                toExamine = currentBoard[row][y];
                if (toExamine != null) {
                    if (this.isOppositeColour(toExamine)) {
                        threatened += toExamine.weight;
                    }
                    break;
                }
            }
        }
        // check right
        if (column < 7) {
            for (int y = column + 1; y <= 7; y++) {
                toExamine = currentBoard[row][y];
                if (toExamine != null) {
                    if (this.isOppositeColour(toExamine)) {
                        threatened += toExamine.weight;
                    }
                    break;
```

```
                }
            }
        }
        int posx = row;
        int posy = column;
        // diagonal top-left
        while (posx > 0 && posy > 0) {
            toExamine = currentBoard[posx--][posy--];
            if (toExamine != null) {
                if (this.isOppositeColour(toExamine)) {
                    threatened += toExamine.weight;
                }
                break;
            }
        }
        posx = row;
        posy = column;
        // diagonal top-right
        while (posx > 0 && posy < 7) {
            toExamine = currentBoard[posx--][posy++];
            if (toExamine != null) {
                if (this.isOppositeColour(toExamine)) {
                    threatened += toExamine.weight;
                }
                break;
            }
        }
        posx = row;
        posy = column;
        // diagonal bottom-left
        while (posx < 7 && posy > 0) {
            toExamine = currentBoard[posx++][posy--];
            if (toExamine != null) {
                if (this.isOppositeColour(toExamine)) {
                    threatened += toExamine.weight;
                }
                break;
            }
        }
        posx = row;
        posy = column;
        // diagonal bottom-right
        while (posx < 7 && posy < 7) {
            toExamine = currentBoard[posx++][posy++];
            if (toExamine != null) {
                if (this.isOppositeColour(toExamine)) {
                    threatened += toExamine.weight;
                }
                break;
            }
        }
        return threatened;
    }

    @Override
    public int[][] attacks(Board board, int row, int column) {
        Piece[][] currentBoard = board.getBoard();
        Piece toExamine;
        int[][] attacked = new int[8][8];
        if (row > 0) {
            // check up
            for (int x = row - 1; x > 0; x--) {
                toExamine = currentBoard[x][column];
                attacked[x][column]++;
                if (toExamine != null) {
                    attacked[x][column]--;
                    break;
                }
            }
```

```
        }
        if (row < 7) {
            // check down
            for (int x = row + 1; x < 8; x++) {
                toExamine = currentBoard[x][column];
                attacked[x][column]++;
                if (toExamine != null) {
                    attacked[x][column]--;
                    break;
                }
            }
        }
        if (column > 0) {
            // check left
            for (int y = column - 1; y > 0; y--) {
                toExamine = currentBoard[row][y];
                attacked[row][y]++;
                if (toExamine != null) {
                    attacked[row][y]--;
                    break;
                }
            }
        }
        if (column < 7) {
            // check right
            for (int y = column + 1; y < 8; y++) {
                toExamine = currentBoard[row][y];
                attacked[row][y]++;
                if (toExamine != null) {
                    attacked[row][y]--;
                    break;
                }
            }
        }
        int posx = row;
        int posy = column;
        // diagonal top-left
        while (posx > 0 && posy > 0) {
            toExamine = currentBoard[posx--][posy--];
            attacked[posx][posy]++;
            if (toExamine != null) {
                attacked[posx][posy]--;
                break;
            }
        }
        posx = row;
        posy = column;
        // diagonal top-right
        while (posx > 0 && posy < 7) {
            toExamine = currentBoard[posx--][posy++];
            attacked[posx][posy]++;
            if (toExamine != null) {
                attacked[posx][posy]--;
                break;
            }
        }
        posx = row;
        posy = column;
        // diagonal bottom-left
        while (posx < 7 && posy > 0) {
            toExamine = currentBoard[posx++][posy--];
            attacked[posx][posy]++;
            if (toExamine != null) {
                attacked[posx][posy]--;
                break;
            }
        }
        posx = row;
        posy = column;
```

```
208             // diagonal bottom-right
209             while (posx < 7 && posy < 7) {
210                 toExamine = currentBoard[posx++][posy++];
211                 attacked[posx][posy]++;
212                 if (toExamine != null) {
213                     attacked[posx][posy]--;
214                     break;
215                 }
216             }
217             return attacked;
218         }
219 
220         @Override
221         public boolean[][] validMoves(Player opponent, Board board, int row, int column) {
222             Piece[][] currentBoard = board.getBoard();
223             Piece toExamine;
224             // reset to false and check
225             canMove = false;
226             boolean[][] validPositions = new boolean[8][8];
227             if (row > 0) {
228                 // check up
229                 for (int x = row - 1; x >= 0; x--) {
230                     toExamine = currentBoard[x][column];
231                     validPositions[x][column] = true;
232                     if (toExamine != null) {
233                         if (this.isOppositeColour(toExamine)) {
234                             validPositions[x][column] = false;
235                             break;
236                         }
237                         canMove = true;
238                         break;
239                     }
240                     canMove = true;
241                 }
242             }
243             if (row < 7) {
244                 // check down
245                 for (int x = row + 1; x <= 7; x++) {
246                     toExamine = currentBoard[x][column];
247                     validPositions[x][column] = true;
248                     if (toExamine != null) {
249                         if (this.isOppositeColour(toExamine)) {
250                             validPositions[x][column] = false;
251                             break;
252                         }
253                         canMove = true;
254                         break;
255                     }
256                     canMove = true;
257                 }
258             }
259             if (column > 0) {
260                 // check left
261                 for (int y = column - 1; y >= 0; y--) {
262                     toExamine = currentBoard[row][y];
263                     validPositions[row][y] = true;
264                     if (toExamine != null) {
265                         if (this.isOppositeColour(toExamine)) {
266                             validPositions[row][y] = false;
267                             break;
268                         }
269                         canMove = true;
270                         break;
271                     }
272                     canMove = true;
273                 }
274             }
275             if (column < 7) {
276                 // check right
```

```
            for (int y = column + 1; y <= 7; y++) {
                toExamine = currentBoard[row][y];
                validPositions[row][y] = true;
                if (toExamine != null) {
                    if (this.isOppositeColour(toExamine)) {
                        validPositions[row][y] = false;
                        break;
                    }
                    canMove = true;
                    break;
                }
                canMove = true;
            }
        }
        int posx = row;
        int posy = column;
        // diagonal top-left
        while (posx > 0 && posy > 0) {
            toExamine = currentBoard[posx--][posy--];
            validPositions[posx][posy] = true;
            if (toExamine != null) {
                if (!this.isOppositeColour(toExamine)) {
                    validPositions[posx][posy] = false;
                    break;
                }
                canMove = true;
                break;
            }
            canMove = true;
        }
        posx = row;
        posy = column;
        // diagonal top-right
        while (posx > 0 && posy < 7) {
            toExamine = currentBoard[posx--][posy++];
            validPositions[posx][posy] = true;
            if (toExamine != null) {
                if (!this.isOppositeColour(toExamine)) {
                    validPositions[posx][posy] = false;
                    break;
                }
                canMove = true;
                break;
            }
            canMove = true;
        }
        posx = row;
        posy = column;
        // diagonal bottom-left
        while (posx < 7 && posy > 0) {
            toExamine = currentBoard[posx++][posy--];
            validPositions[posx][posy] = true;
            if (toExamine != null) {
                if (!this.isOppositeColour(toExamine)) {
                    validPositions[posx][posy] = false;
                    break;
                }
                canMove = true;
                break;
            }
            canMove = true;
        }
        posx = row;
        posy = column;
        // diagonal bottom-right
        while (posx < 7 && posy < 7) {
            toExamine = currentBoard[posx++][posy++];
            validPositions[posx][posy] = true;
            if (toExamine != null) {
```

```
                if (!this.isOppositeColour(toExamine)) {
                    validPositions[posx][posy] = false;
                    break;
                }
                canMove = true;
                break;
            }
            canMove = true;
        }
        return validPositions;
    }

    @Override
    public boolean validSpecial() {
        return false;
    }

    @Override
    public void modifySpecial() {
        // nothing
    }

    @Override
    public String printToBoard() {
        return this.colour == Colour.White ? "\u2655" : "\u265B";
    }

    @Override
    public String printToLog() {
        return "Q";
    }

    @Override
    public boolean getCanMove() {
        return canMove;
    }
}

```