```
/*-------------------------------------------------------------------------------------------------
 * Author:      Dan Cassidy
 * Date:        2015-07-27
 * Assignment:  HW7-3
 * Source File: TicTacToe.java
 * Language:    Java
 * Course:      CSCI-C 490, Android Programming, MoWe 08:00
-------------------------------------------------------------------------------------------------*/
package dancassidy.tictactoe;

/**
 * Model for the Tic-Tac-Toe game.
 * <p/>
 * Can scale to an arbitrary board size and use an arbitrary winning sequence length.
 *
 * @author Dan Cassidy
 */
public class TicTacToe {
    public enum Mark {X, O}

    public enum Status {IN_PROGRESS, X_WIN, O_WIN, DRAW}

    private static final int DEFAULT_NUM_ROWS = 3;
    private static final int DEFAULT_NUM_COLUMNS = 3;
    private static final int DEFAULT_WIN_LENGTH = 3;

    private final int NUM_ROWS;
    private final int NUM_COLUMNS;
    private final int WIN_LENGTH;
    private final int MAX_SPACES;

    private Mark[][] board;
    private Mark turn;
    private Status status;
    private int usedSpaces;

    /**
     * Default constructor. Simply calls the 3-parameter constructor with the default values.
     */
    public TicTacToe() {
        this(DEFAULT_NUM_ROWS, DEFAULT_NUM_COLUMNS, DEFAULT_WIN_LENGTH);
    }

    /**
     * 3-parameter constructor. If there is a problem with an argument, the default value is used.
     *
     * @param rows      The number of rows on the game board. Should be >= 3.
     * @param columns   The number of columns on the game board. Should be >= 3.
     * @param winLength The length of the sequence required to win. Should be >= 3 and <= the
     *                  smaller of the number of rows and the number of columns.
     */
    public TicTacToe(int rows, int columns, int winLength) {
        NUM_ROWS = (rows < DEFAULT_NUM_ROWS ? DEFAULT_NUM_ROWS : rows);
        NUM_COLUMNS = (columns < DEFAULT_NUM_COLUMNS ? DEFAULT_NUM_COLUMNS : columns);
        MAX_SPACES = NUM_ROWS * NUM_COLUMNS;
        if (winLength < DEFAULT_WIN_LENGTH ||
                winLength > (NUM_ROWS > NUM_COLUMNS ? NUM_COLUMNS : NUM_ROWS))
            WIN_LENGTH = DEFAULT_WIN_LENGTH;
        else
            WIN_LENGTH = winLength;
```

```
61          reset();
62      }
63
64      // BEGIN GETTERS AND SETTERS -->
65      public int getColumns() {
66          return NUM_COLUMNS;
67      }
68
69      public int getRows() {
70          return NUM_ROWS;
71      }
72
73      public int getSpaceStringID(int row, int column) {
74          if (!validCoords(row, column) || board[row][column] == null)
75              return R.string.blank;
76          else
77              return (board[row][column] == Mark.X ? R.string.button_x : R.string.button_o);
78      }
79
80      public Status getStatus() {
81          return status;
82      }
83
84      public int getStatusStringID() {
85          switch (status) {
86              case IN_PROGRESS:
87                  return (turn == Mark.X ? R.string.status_x_turn : R.string.status_o_turn);
88              case X_WIN:
89                  return R.string.status_x_win;
90              case O_WIN:
91                  return R.string.status_o_win;
92              case DRAW:
93                  return R.string.status_draw;
94              default:
95                  return R.string.status_error;
96          }
97      }
98
99      public Mark getTurn() {
100         return turn;
101     }
102
103     public int getWinLength() {
104         return WIN_LENGTH;
105     }
106     // <-- END GETTERS AND SETTERS
107
108     /**
109      * Play a single move at the given game board coordinates.
110      *
111      * @param row    The row where the mark should be placed.
112      * @param column The column where the mark should be placed.
113      */
114     public void playMove(int row, int column) {
115         // If the game had ended, no more moves are accepted.
116         if (status != Status.IN_PROGRESS)
117             return;
118
119         // Verify the row and column values.
120         if (!validCoords(row, column))
```

```
121             return;
122
123         // Verify that the destination is empty.
124         if (board[row][column] == null) {
125             usedSpaces++;
126             board[row][column] = turn;
127
128             // Can't be a winning move until at least (WIN_LENGTH * 2 - 1) spaces have been used.
129             if (usedSpaces >= WIN_LENGTH * 2 - 1)
130                 checkBoard();
131
132             turn = (turn == Mark.X ? Mark.O : Mark.X);
133         }
134     }
135
136     /**
137      * Discards the old game board and creates a new one in its place and sets the turn to X, the
138      * game status to in progress, and the number of used spaces to 0.
139      */
140     public void reset() {
141         board = new Mark[NUM_ROWS][NUM_COLUMNS];
142         turn = Mark.X;
143         status = Status.IN_PROGRESS;
144         usedSpaces = 0;
145     }
146
147     /**
148      * Checks the game board to see if there is a winner or a draw.
149      */
150     private void checkBoard() {
151         // Check for winning sequences.
152         if (checkWin())
153             status = (turn == Mark.X ? Status.X_WIN : Status.O_WIN);
154             // Check for a draw.
155         else if (usedSpaces == MAX_SPACES)
156             status = Status.DRAW;
157     }
158
159     /**
160      * Check for a winning sequence recursively in a given 'direction'. Upon first entry into the
161      * method (<b>numSequential</b> = 1), this method does several things to avoid unnecessary
162      * recursions so it can scale well to an arbitrary board size and winning sequence length.
163      * <ul><li>It verifies that the final row/column aren't going to be outside the bounds of the
164      * board.</li>
165      * <li>It checks the neighboring space in the direction of travel to make sure it matches.</li>
166      * <li>It checks the final destination space (that is, the space that this method will look at
167      * if it reaches the WIN_LENGTH'th depth) to make sure it matches.</li></ul>
168      *
169      * @param row              The row portion of the board space being looked at.
170      * @param column           The column portion of the board space being looked at.
171      * @param rowStepOffset    The row offset applied each step.
172      * @param columnStepOffset The column offset applied each step.
173      * @param numSequential    The number of sequential marks found thus far.
174      * @return boolean, indicating whether a winning sequence has been found (true) or not (false).
175      */
176     private boolean checkSequence(int row, int column, int rowStepOffset, int columnStepOffset,
177                                   int numSequential) {
178         // Perform initial checks. These are to cut down on the recursion that needs to happen.
179         if (numSequential == 1) {
180             int finalRow = row + rowStepOffset * (WIN_LENGTH - 1);
```

```
181             int finalColumn = column + columnStepOffset * (WIN_LENGTH - 1);
182
183             // Bounds check.
184             if (!validCoords(finalRow, finalColumn))
185                 return false;
186
187             // Neighbor check.
188             if (board[row + rowStepOffset][column + columnStepOffset] != turn)
189                 return false;
190
191             // Destination check.
192             if (board[finalRow][finalColumn] != turn)
193                 return false;
194         }
195
196         // Verify that the sequence continues to match.
197         if (board[row][column] != turn)
198             return false;
199             // Check to see if the sequence is of winning length.
200         else if (numSequential == WIN_LENGTH)
201             return true;
202
203         // Move to the next spot in the sequence.
204         return checkSequence(row + rowStepOffset, column + columnStepOffset, rowStepOffset,
205                 columnStepOffset, numSequential + 1);
206     }
207
208     /**
209      * Checks for a winning sequence on the game board. Wrapper for the recursive checkSequence
210      * method.
211      *
212      * @return boolean, indicating whether a winning sequence was found (true) or not (false).
213      */
214     private boolean checkWin() {
215         boolean win = false;
216
217         for (int row = 0; !win && row < NUM_ROWS; row++)
218             for (int column = 0; !win && column < NUM_COLUMNS; column++)
219                 // Only need to check for a winning condition if the board space contains a mark
220                 // that is the same as the current turn. E.g. - Only check for a winning condition
221                 // if it is O's turn and the board contains an 'O' in the current space.
222                 if (board[row][column] == turn)
223                     win = checkSequence(row, column, 0, 1, 1) ||   // Right.
224                             checkSequence(row, column, 1, 0, 1) ||   // Down.
225                             checkSequence(row, column, 1, 1, 1) ||   // Diagonal down right.
226                             checkSequence(row, column, -1, 1, 1);    // Diagonal up right.
227
228         return win;
229     }
230
231     /**
232      * Checks the given row and column values to make sure they are valid (within bounds) for the
233      * current game board.
234      *
235      * @param row    The row value to check.
236      * @param column The column value to check.
237      * @return boolean, indicating whether the given coordinates are valid (true) or not (false).
238      */
239     private boolean validCoords(int row, int column) {
240         return row >= 0 && row < NUM_ROWS && column >= 0 && column < NUM_COLUMNS;
```

```
241         }
242     }
243
```