```
1  /*-------------------------------------------------------------------------------------------------------
2   * Author:      Dan Cassidy
3   * Date:        2015-07-18
4   * Assignment:  HW4-1
5   * Source File: Game.java
6   * Language:    Java
7   * Course:      CSCI-C 490, Android Programming, MoWe 08:00
8  -------------------------------------------------------------------------------------------------------*/
9
10 /**
11  * Model for the Tic-Tac-Toe game.
12  *
13  * Can scale to an arbitrary board size and use an arbitrary winning sequence length.
14  *
15  * @author Dan Cassidy
16  */
17 public class GameModel
18 {
19     public static enum Mark { X, O }
20     public static enum Status { IN_PROGRESS, X_WIN, O_WIN, DRAW }
21
22     private static final int DEFAULT_NUM_ROWS = 3;
23     private static final int DEFAULT_NUM_COLUMNS = 3;
24     private static final int DEFAULT_WIN_LENGTH = 3;
25
26     private final int NUM_ROWS;
27     private final int NUM_COLUMNS;
28     private final int WIN_LENGTH;
29     private final int MAX_SPACES;
30
31     private Mark[][] board;
32     private Mark turn;
33     private Status status;
34     private int usedSpaces;
35
36     /**
37      * Default constructor. Simply calls the 3-parameter constructor with the default values.
38      */
39     public GameModel()
40     {
41         this(DEFAULT_NUM_ROWS, DEFAULT_NUM_COLUMNS, DEFAULT_WIN_LENGTH);
42     }
43
44     /**
45      * 3-parameter constructor. If there is a problem with an argument, the default value is used.
46      *
47      * @param rows The number of rows on the game board. Should be >= 3.
48      * @param columns The number of columns on the game board. Should be >= 3.
49      * @param winLength The length of the sequence required to win. Should be >= 3 and <= the
50      * smaller of the number of rows and the number of columns.
51      */
52     public GameModel(int rows, int columns, int winLength)
53     {
54         NUM_ROWS = (rows < DEFAULT_NUM_ROWS ? DEFAULT_NUM_ROWS : rows);
55         NUM_COLUMNS = (columns < DEFAULT_NUM_COLUMNS ? DEFAULT_NUM_COLUMNS : columns);
56         MAX_SPACES = NUM_ROWS * NUM_COLUMNS;
57         if (    winLength < DEFAULT_WIN_LENGTH ||
58                 winLength > (NUM_ROWS > NUM_COLUMNS ? NUM_COLUMNS : NUM_ROWS))
59             WIN_LENGTH = DEFAULT_WIN_LENGTH;
60         else
```

```
61              WIN_LENGTH = winLength;
62          reset();
63      }
64
65      // BEGIN GETTERS AND SETTERS -->
66      public int getColumns()
67      {
68          return NUM_COLUMNS;
69      }
70
71      public int getRows()
72      {
73          return NUM_ROWS;
74      }
75
76      public String getSpaceString(int row, int column)
77      {
78          if (!validCoords(row, column) || board[row][column] == null)
79              return "";
80          else
81              return board[row][column].toString();
82      }
83
84      public Status getStatus()
85      {
86          return status;
87      }
88
89      public String getStatusString()
90      {
91          switch (status)
92          {
93              case IN_PROGRESS:
94                  return turn + "'s Turn";
95              case X_WIN:
96                  return "X Wins";
97              case O_WIN:
98                  return "O Wins";
99              case DRAW:
100                 return "Draw";
101             default:
102                 return "Unknown Status";
103         }
104     }
105
106     public Mark getTurn()
107     {
108         return turn;
109     }
110
111     public int getWinLength()
112     {
113         return WIN_LENGTH;
114     }
115     // <-- END GETTERS AND SETTERS
116
117     /**
118      * Play a single move at the given game board coordinates.
119      *
120      * @param row The row where the mark should be placed.
```

```
121         * @param column The column where the mark should be placed.
122         */
123        public void playMove(int row, int column)
124        {
125            // If the game had ended, no more moves are accepted.
126            if (status != Status.IN_PROGRESS)
127                return;
128
129            // Verify the row and column values.
130            if (!validCoords(row, column))
131                return;
132
133            // Verify that the destination is empty.
134            if (board[row][column] == null)
135            {
136                usedSpaces++;
137                board[row][column] = turn;
138
139                // Can't be a winning move until at least (WIN_LENGTH * 2 - 1) spaces have been used.
140                if (usedSpaces >= WIN_LENGTH * 2 - 1)
141                    checkBoard();
142
143                turn = (turn == Mark.X ? Mark.O : Mark.X);
144            }
145        }
146
147        /**
148         * Discards the old game board and creates a new one in its place and sets the turn to X, the
149         * game status to in progress, and the number of used spaces to 0.
150         */
151        public void reset()
152        {
153            board = new Mark[NUM_ROWS][NUM_COLUMNS];
154            turn = Mark.X;
155            status = Status.IN_PROGRESS;
156            usedSpaces = 0;
157        }
158
159        /**
160         * Checks the game board to see if there is a winner or a draw.
161         */
162        private void checkBoard()
163        {
164            // Check for winning sequences.
165            if (checkWin())
166                status = (turn == Mark.X ? Status.X_WIN : Status.O_WIN);
167            // Check for a draw.
168            else if (usedSpaces == MAX_SPACES)
169                status = Status.DRAW;
170        }
171
172        /**
173         * Check for a winning sequence recursively in a given 'direction'. Upon first entry into the
174         * method (<b>numSequential</b> = 1), this method does several things to avoid unnecessary
175         * recursions so it can scale well to an arbitrary board size and winning sequence length.
176         * <ul><li>It verifies that the final row/column aren't going to be outside the bounds of the
177         * board.</li>
178         * <li>It checks the neighboring space in the direction of travel to make sure it matches.</li>
179         * <li>It checks the final destination space (that is, the space that this method will look at
180         * if it reaches the WIN_LENGTH'th depth) to make sure it matches.</li></ul>
```

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

```
241                             checkSequence(row, column, 1, 1, 1) ||   // Diagonal down right.
242                             checkSequence(row, column, -1, 1, 1);    // Diagonal up right.
243
244         return win;
245     }
246
247     /**
248      * Checks the given row and column values to make sure they are valid (within bounds) for the
249      * current game board.
250      *
251      * @param row The row value to check.
252      * @param column The column value to check.
253      * @return boolean, indicating whether the given coordinates are valid (true) or not (false).
254      */
255     private boolean validCoords(int row, int column)
256     {
257         return row >= 0 && row < NUM_ROWS && column >= 0 && column < NUM_COLUMNS;
258     }
259 }
260
```