```
 1: /*
 2:  *    Project:  Word Search (CS 360 Fall 2015, Project 3)
 3:  *    File:   Direction.java
 4:  *    Author:   Jacob A. Zarobsky
 5:  *    Date:   Nov 6, 2015
 6:  *
 7:  *    This file stores readable direction values.
 8:  */
 9:
10:  public enum Direction {
11:     NORTH ("n"),
12:     NORTH_EAST ("ne"),
13:     EAST ("e"),
14:     SOUTH_EAST ("se"),
15:     SOUTH ("s"),
16:     SOUTH_WEST ("sw"),
17:     WEST ("w"),
18:     NORTH_WEST ("nw"),
19:     ANY ("n/a"); // Used for the first inital direction.
20:
21:     private final String display;
22:
23:     private Direction(String s) { display = s; }
24:
25:     public String toString() { return display; }
26:  }
```

```
 1: /*
 2: *     Project:  Word Search (CS 360 Fall 2015, Project 3)
 3: *     File:   FileReader.java
 4: *     Author:   Jacob A. Zarobsky
 5: *     Date:   Nov 5, 2015
 6:  *
 7:  *      This class implements a file reader
 8:  *      that uses a lamda to deal with
 9:  *      each individual line.
10:  */
11:
12: import java.io.BufferedReader;
13: import java.io.FileInputStream;
14: import java.io.FileNotFoundException;
15: import java.io.InputStream;
16: import java.io.InputStreamReader;
17: import java.io.PrintStream;
18: import java.io.UnsupportedEncodingException;
19: import java.io.IOException;
20: import java.nio.charset.Charset;
21: import java.util.function.BiConsumer;
22:
23: public class FileReader {
24:     // Properties
25:     private String filePath;
26:     private FileInputStream inputStream;
27:     private InputStreamReader inputStreamReader;
28:     private BufferedReader bufferedReader;
29:     private int currentLine = Integer.MIN_VALUE;
30:
31:     // Accessors
32:     public void setFilePath(String f) { filePath = f; }
33:     public String getFilePath() { return filePath; }
34:
35:     public void setInputStream(FileInputStream i) { inputStream = i; }
36:     public FileInputStream getInputStream() { return inputStream; }
37:
38:     public void setInputStreamReader(InputStreamReader i) {
39:         inputStreamReader = i;
40:     }
41:
42:     public InputStreamReader getInputStreamReader() {
43:         return inputStreamReader;
44:     }
45:
46:     public void setBufferedReader(BufferedReader r) { bufferedReader = r; }
47:     public BufferedReader getBufferedReader() { return bufferedReader; }
48:
49:     private void setCurrentLine(int cl) { currentLine = cl; }
50:     private int getCurrentLine() { return currentLine; }
51:
52:     // Convenience
53:     private void incrementLine() { currentLine ++; }
54:
55:     // Constructor
56:     public FileReader(String path) {
57:         setFilePath(path);
58:     }
59:
60:     // Uses a lamda to deal with every line. Thanks, Java 8
61:     public void forEachLine(BiConsumer<String, Integer> lambda) {
62:         try {
63:             // Create our input stream objects.
64:             setInputStream(new FileInputStream(getFilePath()));
65:             setInputStreamReader(new InputStreamReader
66:                     (getInputStream(), Charset.forName("UTF-8")));
67:             setBufferedReader(new BufferedReader(getInputStreamReader()));
68:
69:             String line = null;
70:             BufferedReader reader = getBufferedReader();
```

```
71:
72:             // Set our current line to 1.
73:             setCurrentLine(1);
74:
75:             while((line = reader.readLine()) != null) {
76:                 // Call the function that was passed in
77:                 lambda.accept(line, new Integer(getCurrentLine()));
78:                 incrementLine();
79:             }
80:
81:             getInputStream().close();
82:             getInputStreamReader().close();
83:             getBufferedReader().close();
84:         } catch (UnsupportedEncodingException ex) {
85:             Main.exitWithError(
86:                 "The encoding used was incompatible with the file.");
87:         } catch (FileNotFoundException ex) {
88:             Main.exitWithError("The file you entered was not found.");
89:         } catch (IOException ex) {
90:             Main.exitWithError(
91:                 "There was an IO error while attempting to read the file.");
92:         }
93:     }
94: }
```

```
 1: /*
 2:  *    Project:  Word Search (CS 360 Fall 2015, Project 3)
 3:  *    File:   Graph.java
 4:  *    Author:   Jacob A. Zarobsky
 5:  *    Date:   Nov 5, 2015
 6:  *
 7:  *    This file stores all the data for the graph used to solve the
 8:  *      puzzle. It also contains methods to traverse the graph.
 9:  */
10:
11: import java.util.LinkedList;
12: import java.util.function.BiConsumer;
13: import java.util.HashSet;
14: import java.util.Map;
15: import java.util.Iterator;
16:
17: public class Graph {
18:     // 2D Array allows direct access to any vertex.
19:     private Vertex[][] verticies;
20:     // Hash Set of all Dictionary terms. Allows for O(1) search.
21:     private HashSet<String> dictionary;
22:
23:     // Accessors
24:     public void setVerticies(Vertex[][] v) { verticies = v; }
25:     public Vertex[][] getVerticies() { return verticies; }
26:
27:     public void setDictionary(HashSet<String> d) { dictionary = d; }
28:     public HashSet<String> getDictionary() { return dictionary; }
29:
30:     // Constructor
31:     public Graph(int rows, int columns) {
32:         verticies = new Vertex[rows][columns];
33:     }
34:
35:     // Convenience
36:     public void addVertex(int row, int column, char letter) {
37:         verticies[row][column] = new Vertex(letter);
38:     }
39:
40:     // Convenience
41:     public void forEachVertex(BiConsumer<Integer, Integer> consumer) {
42:         for(int i = 0; i < verticies.length; i++) {
43:             for(int j = 0; j < verticies[i].length; j++) {
44:                 consumer.accept(i, j);
45:             }
46:         }
47:     }
48:
49:     // This method links each vertex to all adjacent verticies.
50:     public void populateEdges() {
51:         forEachVertex((Integer row, Integer column) -> {
52:             Vertex v = verticies[row][column];
53:             // Add the "North Edge"
54:             if(row > 0)
55:                 v.addEdge(verticies[row-1][column], Direction.NORTH);
56:
57:             // Add the "North East Edge"
58:             if(row > 0 && column < verticies[row].length - 1)
59:                 v.addEdge(verticies[row-1][column+1], Direction.NORTH_EAST);
60:
61:             // Add the "East Edge"
62:             if(column < verticies[row].length - 1)
63:                 v.addEdge(verticies[row][column+1], Direction.EAST);
64:
65:             // Add the South East Edge
66:             if(column < verticies[row].length - 1 && row < verticies.length - 1)
67:                 v.addEdge(verticies[row+1][column+1], Direction.SOUTH_EAST);
68:
69:             // Add the "South Edge"
70:             if(row < verticies.length - 1)
```

```
71:                 v.addEdge(verticies[row+1][column], Direction.SOUTH);
72:
73:             // Add the "South West Edge"
74:             if(row < verticies.length - 1 &&  column > 0)
75:                 v.addEdge(verticies[row+1][column-1], Direction.SOUTH_WEST);
76:
77:             // Add the "West Edge"
78:             if(column > 0)
79:                 v.addEdge(verticies[row][column-1], Direction.WEST);
80:
81:             // Add the Northwest Edge
82:             if(column > 0 && row > 0)
83:                 v.addEdge(verticies[row-1][column-1], Direction.NORTH_WEST);
84:         });
85:     }
86:
87:     // Performs a valid depth first search at each vertex.
88:     public void depthFirstSearch(int row, int column) {
89:         // Inititalize a stack.
90:         LinkedList<DFSStackItem> stack = new LinkedList<DFSStackItem>();
91:         // Use the 2D array to get direct access to our start vertex.
92:         Vertex startVertex = verticies[row][column];
93:
94:         // Some useful variables.
95:         String newString =  startVertex.getLetter() + "";
96:         Vertex toVertex = null;
97:         Direction d = Direction.ANY;
98:
99:         // Push our start vertex.
100:        stack.push(new DFSStackItem(startVertex, d, newString));
101:
102:        while(!stack.isEmpty())
103:        {
104:            DFSStackItem item = stack.pop();
105:            Vertex vertex = item.getVertex();
106:
107:            // This will be used the first time only.
108:            if(item.getDirection() == Direction.ANY) {
109:
110:                Iterator iterator = vertex.getEdges().entrySet().iterator();
111:                while(iterator.hasNext()) {
112:                    Map.Entry pair = (Map.Entry)iterator.next();
113:                    toVertex = (Vertex)pair.getValue();
114:                    newString = item.getCurrentString() + toVertex.getLetter();
115:                    d = (Direction) pair.getKey();
116:                    stack.push(new DFSStackItem(toVertex, d,
117:                        newString));
118:                }
119:                continue; // Done with this iteration. Go on to next one.
120:                // Note: this could be easily done with an if/else block
121:                // as well, however identations/formatting looked terrible
122:                // becaue there are really long lines. Continue seemed like the
123:                // next best option.
124:            }
125:
126:            if(vertex.getEdges().containsKey(item.getDirection())) {
127:                toVertex = vertex.getEdges().get(item.getDirection());
128:                newString = item.getCurrentString() + toVertex.getLetter();
129:
130:                // If we find a word, print it ASAP.
131:                if(newString.length() > 3 &&
132:                    getDictionary().contains(newString)) {
133:                    System.out.printf(
134:                        "%s (%d,%d,%s)\n",
135:                        newString, column + 1, row + 1, item.getDirection());
136:                }
137:
138:                stack.push(new DFSStackItem(toVertex, item.getDirection(),
139:                    newString));
140:            }
```

```
141:         }
142:     }
143:
144:     // Private internal class used for the DFS stack.
145:     class DFSStackItem {
146:         // The Vertex we are going to do our DFS at.
147:         private Vertex vertex;
148:         // The current direction we're traveling.
149:         private Direction direction;
150:         // The sequence of characters thus far.
151:         private String currentString;
152:
153:         // Accessors
154:         public void setVertex(Vertex v) { vertex = v; }
155:         public Vertex getVertex() { return vertex; }
156:
157:         public void setDirection(Direction d) { direction = d; }
158:         public Direction getDirection() { return direction; }
159:
160:         public void setCurrentString(String s) { currentString = s; }
161:         public String getCurrentString() { return currentString; }
162:
163:         // Constructor
164:         public DFSStackItem(Vertex v, Direction d, String s) {
165:             setVertex(v);
166:             setDirection(d);
167:             setCurrentString(s);
168:         }
169:     }
170: }
```

```
 1: /*
 2:  *    Project:  Word Search (CS 360 Fall 2015, Project 3)
 3:  *    File:   Main.java
 4:  *    Author:   Jacob A. Zarobsky
 5:  *    Date:   Nov 5, 2015
 6:  *
 7:  *    This file is the main entry point for the program.
 8:  *      The program reads in a word search and then solves
 9:  *      the word search.
10:  */
11:
12: public class Main {
13:     public static void main(String[] args) {
14:         new WordSearch("puzzle.txt", "words.txt").run();
15:     }
16:
17:     public static void exitWithError(String errorMessage) {
18:         // Print the error in red.
19:         System.err.println(errorMessage);
20:
21:         // Return a number other than 0.
22:         System.exit(1);
23:     }
24: }
```

```
 1: /*
 2:  *    Project:  Word Search (CS 360 Fall 2015, Project 3)
 3:  *    File:   Vertex.java
 4:  *    Author:   Jacob A. Zarobsky
 5:  *    Date:   Nov 6, 2015
 6:  *
 7:  *    This file stores all the data on a vertex.
 8:  */
 9:
10: import java.util.EnumMap;
11:
12:  public class Vertex {
13:      // The letter of this vertex.
14:      private char letter;
15:      // The edges leaving out of this vertex.
16:      private EnumMap<Direction, Vertex> edges;
17:      // Property Accessors.
18:      public void setLetter(char d) { letter = d; }
19:      public char getLetter() { return letter; }
20:
21:      public void setEdges(EnumMap<Direction, Vertex> e) { edges = e; }
22:
23:      // EnumMap was chosen to allow constant time access to any
24:      // direction without creating either a bunch of methods
25:      // or a bunch of pointers and subsequent logic that would have
26:      // to go with it.
27:      public EnumMap<Direction, Vertex> getEdges() {
28:          // Lazy instantiation.
29:          if(edges == null)
30:              edges = new EnumMap<Direction, Vertex>(Direction.class);
31:
32:          return edges;
33:      }
34:
35:      // Constructor
36:      public Vertex(char letter) {
37:         setLetter(letter);
38:      }
39:
40:      // Convenience
41:      public void addEdge(Vertex toVertex, Direction d) {
42:          getEdges().put(d, toVertex);
43:      }
44:  }
```

```
 1: /*
 2:  *    Project:  Word Search (CS 360 Fall 2015, Project 3)
 3:  *    File:   WordSearch.java
 4:  *    Author:   Jacob A. Zarobsky
 5:  *    Date:   Nov 5, 2015
 6:  *
 7:  *    This file runs the WordSearch and stores all
 8:  *      necessary data for the search.
 9:  */
10:
11: import java.util.HashSet;
12:
13: public class WordSearch {
14:     // Private Properties
15:     private String puzzleSource;
16:     private String wordSource;
17:     private Graph graph;
18:     private HashSet<String> dictionary;
19:
20:     // Accessors
21:     public void setPuzzleSource(String pSource) { puzzleSource = pSource; }
22:     public String getPuzzleSource() { return puzzleSource; }
23:
24:     public void setWordSource(String wSource) { wordSource = wSource; }
25:     public String getWordSource() { return wordSource; }
26:
27:     public void setGraph(Graph g) { graph = g; }
28:     public Graph getGraph() { return graph; }
29:
30:     public void setDictionary(HashSet<String> dict) { dictionary = dict; }
31:     public HashSet<String> getDictionary() {
32:         // Lazy instantiation
33:         if(dictionary == null)
34:             dictionary = new HashSet<String>();
35:
36:         return dictionary;
37:     }
38:
39:     // Constructor
40:     public WordSearch(String puzzleSource, String wordSource) {
41:         setPuzzleSource(puzzleSource);
42:         setWordSource(wordSource);
43:     }
44:
45:     // Where the action happens.
46:     public void run() {
47:         initalizeSources();
48:         graph.setDictionary(dictionary);
49:         graph.forEachVertex((Integer row, Integer column) -> {
50:             graph.depthFirstSearch(row, column);
51:         });
52:     }
53:
54:     // Load up our graph and our dictionary.
55:     private void initalizeSources() {
56:         final FileReader fileReader = new FileReader(getPuzzleSource());
57:         final String delimiters = " ";
58:         fileReader.forEachLine((String line, Integer lineNumber) -> {
59:             // The first line of this file contains the size of the puzzle
60:             // we need to solve. Treat it differntly than all the rest.
61:             if(lineNumber == 1) {
62:                 // Get the size of the puzzle.
63:                 int size = new Integer(line);
64:                 // Initalize a new square graph.
65:                 setGraph(new Graph(size, size));
66:             } else {
67:                 // Split the string based on spaces.
68:                 String[] letters = line.split(" ");
69:
70:                 // Add a vertex for every letter in the line.
```

```
71:                 for(int i = 0; i < letters.length; i++)
72:                     graph.addVertex(lineNumber - 2, i, letters[i].charAt(0));
73:             }
74:         });
75:
76:         // Add in all the edges that we didn't do as we parsed in.
77:         graph.populateEdges();
78:
79:         fileReader.setFilePath(getWordSource());
80:
81:         // Read in the dictionary.
82:         fileReader.forEachLine((String line, Integer lineNumber) -> {
83:             getDictionary().add(line);
84:         });
85:     }
86: }
```