```
 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:  {
12:     NORTH ("n"),
13:     NORTH_EAST ("ne"),
14:     EAST ("e"),
15:     SOUTH_EAST ("se"),
16:     SOUTH ("s"),
17:     SOUTH_WEST ("sw"),
18:     WEST ("w"),
19:     NORTH_WEST ("nw"),
20:     ANY ("n/a"); // Used for the first inital direction.
21:
22:     private final String display;
23:
24:     private Direction(String s) { display = s; }
25:
26:     public String toString() { return display; }
27:  }
```

```
 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: {
25:     // Properties
26:     private String filePath;
27:     private FileInputStream inputStream;
28:     private InputStreamReader inputStreamReader;
29:     private BufferedReader bufferedReader;
30:     private int currentLine = Integer.MIN_VALUE;
31:
32:     // Accessors
33:     public void setFilePath(String f) { filePath = f; }
34:     public String getFilePath() { return filePath; }
35:
36:     public void setInputStream(FileInputStream i) { inputStream = i; }
37:     public FileInputStream getInputStream() { return inputStream; }
38:
39:     public void setInputStreamReader(InputStreamReader i)
40:     {
41:         inputStreamReader = i;
42:     }
43:
44:     public InputStreamReader getInputStreamReader()
45:     {
46:         return inputStreamReader;
47:     }
48:
49:     public void setBufferedReader(BufferedReader r) { bufferedReader = r; }
50:     public BufferedReader getBufferedReader() { return bufferedReader; }
51:
52:     private void setCurrentLine(int cl) { currentLine = cl; }
53:     private int getCurrentLine() { return currentLine; }
54:
55:     // Convenience
56:     private void incrementLine() { currentLine ++; }
57:
58:     // Constructor
59:     public FileReader(String path)
60:     {
61:         setFilePath(path);
62:     }
63:
64:     // Uses a lamda to deal with every line. Thanks, Java 8
65:     public void forEachLine(BiConsumer<String, Integer> lambda)
66:     {
67:         try {
68:             // Create our input stream objects.
69:             setInputStream(new FileInputStream(getFilePath()));
70:             setInputStreamReader(new InputStreamReader
```

```
   71:                     (getInputStream(), Charset.forName("UTF-8")));
   72:             setBufferedReader(new BufferedReader(getInputStreamReader()));
   73:
   74:             String line = null;
   75:             BufferedReader reader = getBufferedReader();
   76:
   77:             // Set our current line to 1.
   78:             setCurrentLine(1);
   79:
   80:             while((line = reader.readLine()) != null)
   81:             {
   82:                 // Call the function that was passed in
   83:                 lambda.accept(line, new Integer(getCurrentLine()));
   84:                 incrementLine();
   85:             }
   86:
   87:             getInputStream().close();
   88:             getInputStreamReader().close();
   89:             getBufferedReader().close();
   90:         }
   91:         catch (UnsupportedEncodingException ex)
   92:         {
   93:             Main.exitWithError(
   94:                 "The encoding used was incompatible with the file.");
   95:         }
   96:         catch (FileNotFoundException ex)
   97:         {
   98:             Main.exitWithError("The file you entered was not found.");
   99:         }
  100:         catch (IOException ex)
  101:         {
  102:             Main.exitWithError(
  103:                 "There was an IO error while attempting to read the file.");
  104:         }
  105:     }
  106: }
```

```
 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; // Used for the Stack.
12: import java.util.function.BiConsumer;
13: import java.util.HashSet; // Used for the dictionary of words.
14: import java.util.Map; // Used for Map.Entrys
15: import java.util.Iterator;
16:
17: public class Graph
18: {
19:     // 2D Array allows direct access to any vertex.
20:     private Vertex[][] verticies;
21:     // Hash Set of all Dictionary terms. Allows for O(1) search.
22:     private HashSet<String> dictionary;
23:
24:     // Accessors
25:     public void setVerticies(Vertex[][] v) { verticies = v; }
26:     public Vertex[][] getVerticies() { return verticies; }
27:
28:     public void setDictionary(HashSet<String> d) { dictionary = d; }
29:     public HashSet<String> getDictionary() { return dictionary; }
30:
31:     // Constructor
32:     public Graph(int rows, int columns)
33:     {
34:         verticies = new Vertex[rows][columns];
35:     }
36:
37:     // Convenience
38:     public void addVertex(int row, int column, char letter)
39:     {
40:         verticies[row][column] = new Vertex(letter);
41:     }
42:
43:     // Convenience
44:     public void forEachVertex(BiConsumer<Integer, Integer> consumer)
45:     {
46:         for(int i = 0; i < verticies.length; i++)
47:         {
48:             for(int j = 0; j < verticies[i].length; j++)
49:             {
50:                 consumer.accept(i, j);
51:             }
52:         }
53:     }
54:
55:     // This method links each vertex to all adjacent verticies.
56:     public void populateEdges()
57:     {
58:         forEachVertex((Integer row, Integer column) -> {
59:             Vertex v = verticies[row][column];
60:             // Add the "North Edge"
61:             if(row > 0)
62:                 v.addEdge(verticies[row-1][column], Direction.NORTH);
63:
64:             // Add the "North East Edge"
65:             if(row > 0 && column < verticies[row].length - 1)
66:                 v.addEdge(verticies[row-1][column+1], Direction.NORTH_EAST);
67:
68:             // Add the "East Edge"
69:             if(column < verticies[row].length - 1)
70:                 v.addEdge(verticies[row][column+1], Direction.EAST);
```

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

```
141:                 // If we find a word, print it ASAP.
142:                 if(newString.length() > 3 &&
143:                     getDictionary().contains(newString))
144:                 {
145:                     System.out.printf(
146:                         "%s (%d,%d,%s)\n",
147:                         newString, column + 1, row + 1, item.getDirection());
148:                 }
149:
150:                 stack.push(new DFSStackItem(toVertex, item.getDirection(),
151:                     newString));
152:             }
153:         }
154:     }
155:
156:     // Private internal class used for the DFS stack.
157:     class DFSStackItem
158:     {
159:         // The Vertex we are going to do our DFS at.
160:         private Vertex vertex;
161:         // The current direction we're traveling.
162:         private Direction direction;
163:         // The sequence of characters thus far.
164:         private String currentString;
165:
166:         // Accessors
167:         public void setVertex(Vertex v) { vertex = v; }
168:         public Vertex getVertex() { return vertex; }
169:
170:         public void setDirection(Direction d) { direction = d; }
171:         public Direction getDirection() { return direction; }
172:
173:         public void setCurrentString(String s) { currentString = s; }
174:         public String getCurrentString() { return currentString; }
175:
176:         // Constructor
177:         public DFSStackItem(Vertex v, Direction d, String s)
178:         {
179:             setVertex(v);
180:             setDirection(d);
181:             setCurrentString(s);
182:         }
183:     }
184: }
```

```
 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: {
14:     public static void main(String[] args)
15:     {
16:         new WordSearch("puzzle.txt", "words.txt").run();
17:     }
18:
19:     public static void exitWithError(String errorMessage)
20:     {
21:         // Print the error in red.
22:         System.err.println(errorMessage);
23:
24:         // Return a number other than 0.
25:         System.exit(1);
26:     }
27: }
```

```
 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:  {
14:      // The letter of this vertex.
15:      private char letter;
16:      // The edges leaving out of this vertex.
17:      private EnumMap<Direction, Vertex> edges;
18:      // Property Accessors.
19:      public void setLetter(char d) { letter = d; }
20:      public char getLetter() { return letter; }
21:
22:      public void setEdges(EnumMap<Direction, Vertex> e) { edges = e; }
23:
24:      // EnumMap was chosen to allow constant time access to any
25:      // direction without creating either a bunch of methods
26:      // or a bunch of pointers and subsequent logic that would have
27:      // to go with it.
28:      public EnumMap<Direction, Vertex> getEdges()
29:      {
30:          // Lazy instantiation.
31:          if(edges == null)
32:              edges = new EnumMap<Direction, Vertex>(Direction.class);
33:
34:          return edges;
35:      }
36:
37:      // Constructor
38:      public Vertex(char letter)
39:      {
40:         setLetter(letter);
41:      }
42:
43:      // Convenience
44:      public void addEdge(Vertex toVertex, Direction d)
45:      {
46:          getEdges().put(d, toVertex);
47:      }
48:  }
```

```
 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.ArrayList;
12: import java.util.StringTokenizer;
13: import java.util.Collections;
14: import java.util.HashSet;
15:
16: public class WordSearch
17: {
18:     // Private Properties
19:     private String puzzleSource;
20:     private String wordSource;
21:     private Graph graph;
22:     private HashSet<String> dictionary;
23:
24:     // Accessors
25:     public void setPuzzleSource(String pSource) { puzzleSource = pSource; }
26:     public String getPuzzleSource() { return puzzleSource;}
27:
28:     public void setWordSource(String wSource) { wordSource = wSource; }
29:     public String getWordSource() { return wordSource; }
30:
31:     public void setGraph(Graph g) { graph = g; }
32:     public Graph getGraph() { return graph; }
33:
34:     public void setDictionary(HashSet<String> dict) { dictionary = dict; }
35:     public HashSet<String> getDictionary() {
36:         // Lazy instantiation
37:         if(dictionary == null)
38:             dictionary = new HashSet<String>();
39:
40:         return dictionary;
41:     }
42:
43:     // Constructor
44:     public WordSearch(String puzzleSource, String wordSource)
45:     {
46:         setPuzzleSource(puzzleSource);
47:         setWordSource(wordSource);
48:     }
49:
50:     // Where the action happens.
51:     public void run()
52:     {
53:         initalizeSources();
54:         graph.setDictionary(dictionary);
55:         graph.forEachVertex((Integer row, Integer column) -> {
56:             graph.depthFirstSearch(row, column);
57:         });
58:     }
59:
60:     // Load up our graph and our dictionary.
61:     private void initalizeSources()
62:     {
63:         final FileReader fileReader = new FileReader(getPuzzleSource());
64:         final String delimiters = " ";
65:         fileReader.forEachLine((String line, Integer lineNumber) ->
66:         {
67:             // The first line of this file contains the size of the puzzle
68:             // we need to solve. Treat it differntly than all the rest.
69:             if(lineNumber == 1)
70:             {
```

```
71:                 // Get the size of the puzzle.
72:                 int size = new Integer(line);
73:                 // Initalize a new square graph.
74:                 setGraph(new Graph(size, size));
75:             }
76:             else
77:             {
78:                 // Split the string based on spaces.
79:                 String[] letters = line.split(" ");
80:
81:                 // Add a vertex for every letter in the line.
82:                 for(int i = 0; i < letters.length; i++)
83:                     graph.addVertex(lineNumber - 2, i, letters[i].charAt(0));
84:             }
85:         });
86:
87:         // Add in all the edges that we didn't do as we parsed in.
88:         graph.populateEdges();
89:
90:         fileReader.setFilePath(getWordSource());
91:
92:         // Read in the dictionary.
93:         fileReader.forEachLine((String line, Integer lineNumber) ->
94:         {
95:             getDictionary().add(line);
96:         });
97:     }
98: }
```