```
 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 the graph in a 2D Array of Characters.
 8:  *      Using a stringbuilder and a PrefixTree, it searches
 9:  *      the puzzle for a list of words.
10:  */
11:
12: import java.util.function.BiConsumer;
13: import java.util.LinkedList;
14:
15: public class Graph {
16:
17:     // Properties
18:     private char[][] letters;
19:     private PrefixTree tree;
20:     private StringBuilder stringBuilder;
21:
22:     // Accessors
23:     public void setLetters(char[][] c) { letters = c; }
24:     public char[][] getLetters() { return letters; }
25:
26:     public StringBuilder getStringBuilder() {
27:         // Lazy instantiation
28:         if(stringBuilder == null) stringBuilder = new StringBuilder();
29:
30:         return stringBuilder;
31:     }
32:
33:     public void setStringBuilder(StringBuilder sb) { stringBuilder = sb; }
34:
35:     public void setTree(PrefixTree t) { tree = t; }
36:     public PrefixTree getTree() { return tree; }
37:
38:     // Constructor
39:     public Graph(int size) {
40:         setLetters(new char[size][size]);
41:     }
42:
43:     // Convenience
44:     public void addVertex(int row, int col, char c) {
45:         letters[row][col] = c;
46:     }
47:
48:     // Convenience
49:     public int getSize() { return letters.length; }
50:
51:     // Convenience
52:     public void forEachVertex(BiConsumer<Integer, Integer> lambda) {
53:         for(int i = 0; i < getSize(); i++) {
54:             for(int j = 0; j < getSize(); j++ ) {
55:                 lambda.accept(i, j);
56:             }
57:         }
58:     }
59:
60:     public void dfs(int row, int col, int dx, int dy) {
61:         StringBuilder sb = getStringBuilder();
62:         char[][] letters  = getLetters();
63:
64:         // Initalize a stack.
65:         // This stack holds valid moves. It should potentially
66:         // only hold 1 value at a time.
67:         LinkedList<Node> stack = new LinkedList<Node>();
68:
69:         // Clear the string builder.
70:         sb.setLength(0);
```

```
 71:
 72:         // Do some initalization.
 73:         int currentRow = row, currentCol = col;
 74:         char currentChar = letters[row][col];
 75:
 76:         // Get our root node we're going to use.
 77:         Node current = tree.getRoot(currentChar);
 78:         stack.push(current);
 79:
 80:         // While the stack is not empty (there is valid moves) and
 81:         // the current item is not null
 82:         while(!stack.isEmpty() && (current = stack.pop()) != null) {
 83:             // Append the current character to the string builder.
 84:             sb.append(currentChar);
 85:
 86:             // We found a word! Yay!
 87:             if(current.getEndOfWord() && sb.length() > 3) {
 88:                 System.out.printf("%s (%d,%d,%s)\n", sb.toString(),
 89:                         col + 1, row + 1, direction(dx, dy));
 90:             }
 91:
 92:             if(canMoveAgain(currentRow, currentCol, dx, dy)) {
 93:                 // Increment these two
 94:                 currentRow += dx; currentCol += dy;
 95:                 // Update the current character.
 96:                 currentChar = letters[currentRow][currentCol];
 97:                 stack.push(tree.lookup(current, currentChar));
 98:             }
 99:         }
100:     }
101:
102:     // Determines if you can move again given a row, column, and dx dy.
103:     private boolean canMoveAgain(int row, int col, int dx, int dy) {
104:         row += dx;
105:         col += dy;
106:         return row >= 0 && row < getSize() && col >= 0 && col < getSize();
107:     }
108:
109:     // Builds a directoin string based on the dx and dy values.
110:     private String direction(int dx, int dy) {
111:       StringBuilder direction = new StringBuilder();
112:
113:       if(dx == -1) direction.append("n");
114:       if(dx == 1) direction.append("s");
115:
116:       if(dy == -1) direction.append("w");
117:       if(dy == 1) direction.append("e");
118:
119:       return direction.toString();
120:   }
121: }
```

```
 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:   Node.java
 4:  *    Author:   Jacob A. Zarobsky
 5:  *    Date:   Nov 5, 2015
 6:  *
 7:  *    This file stores the data associated with a Node in the PrefixTree.
 8:  */
 9:
10: import java.util.HashMap;
11:
12: public class Node {
13:     // Properties
14:     // Store our children in a hashmap Constant time access.
15:     private HashMap<Character, Node> children;
16:     // Store our character.
17:     private char letter;
18:     // Flag for end of the word.
19:     private boolean endOfWord;
20:
21:     // Accessors
22:     public HashMap<Character, Node> getChildren() { return children; }
23:     public void setChildren(HashMap<Character, Node> c) { children = c; }
24:
25:     public char getLetter() { return letter; }
26:     public void setLetter(char l) { letter = l; }
27:
28:     public boolean getEndOfWord() { return endOfWord; }
29:     public void setEndOfWord(boolean eow) { endOfWord = eow; }
30:
31:     // Constructor
32:     public Node(char letter) {
33:         setLetter(letter);
34:         setChildren(new HashMap<Character, Node>());
35:     }
36: }
```

```
 1: /*
 2:  *    Project:  Word Search (CS 360 Fall 2015, Project 3)
 3:  *    File:   PrefixTree.java
 4:  *    Author:   Jacob A. Zarobsky
 5:  *    Date:   Nov 17, 2015
 6:  *
 7:  *    This class implements a simple prefix tree.
 8:  */
 9:
10: import java.util.HashMap;
11:
12: public class PrefixTree {
13:
14:     // HashMap of roots, allows us to access the start node
15:     // for any letter in constant time.
16:     private HashMap<Character, Node> roots;
17:
18:     // Returns the given root for the character.
19:     public Node getRoot(char c) {
20:         // Lazy Instantiaion
21:         if(roots == null) roots = new HashMap<Character, Node>();
22:
23:         if(!roots.containsKey(c))
24:             roots.put(c, new Node(c));
25:
26:         return roots.get(c);
27:     }
28:
29:     public void insert(String s) {
30:         // Get the start node.
31:         Node current = getRoot(s.charAt(0));
32:
33:         // Iterate through the characters of the string, moving
34:         // and inserting as we go along.
35:         for(int i = 1; i < s.length(); i++) {
36:             char c = s.charAt(i);
37:             // There's already a node with that character, make that our
38:             // current node.
39:             if(current.getChildren().containsKey(c)) {
40:                 current = current.getChildren().get(c);
41:             } else {
42:                 Node newNode = new Node(c); // Create a new node.
43:                 current.getChildren().put(c, newNode); // Add it to children.
44:                 current = newNode; // Set it to be our current.
45:             }
46:
47:             // If we're at the end of the word then set our flag that
48:             // we're at the end (so we know when looking up that it is a
49:             // valid word.)
50:             if(i == (s.length() - 1)) current.setEndOfWord(true);
51:         }
52:     }
53:
54:     // Returns the child with the given key, if one exisits, otherwise null.
55:     public Node lookup(Node n, char c) {
56:         if(n.getChildren().containsKey(c)) return n.getChildren().get(c);
57:         return null;
58:     }
59: }
```

```
 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 PrefixTree 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(PrefixTree dict) { dictionary = dict; }
31:     public PrefixTree getDictionary() {
32:         // Lazy instantiation
33:         if(dictionary == null) dictionary = new PrefixTree();
34:         return dictionary;
35:     }
36:
37:     // Constructor
38:     public WordSearch(String puzzleSource, String wordSource) {
39:         setPuzzleSource(puzzleSource);
40:         setWordSource(wordSource);
41:     }
42:
43:     // Where the action happens.
44:     public void run() {
45:         initalizeSources();
46:         graph.setTree(dictionary);
47:
48:         graph.forEachVertex((Integer row, Integer col) -> {
49:             for(int i = -1; i < 2; i++) {
50:                 for(int j = -1; j < 2; j++) {
51:                     if( i == 0 && j == 0) continue;
52:                     graph.dfs(row, col, i, j);
53:                 }
54:             }
55:         });
56:     }
57:
58:     // Load up our graph and our dictionary.
59:     private void initalizeSources() {
60:         final FileReader fileReader = new FileReader(getPuzzleSource());
61:         final String delimiters = " ";
62:         fileReader.forEachLine((String line, Integer lineNumber) -> {
63:             // The first line of this file contains the size of the puzzle
64:             // we need to solve. Treat it differntly than all the rest.
65:             if(lineNumber == 1) {
66:                 // Get the size of the puzzle.
67:                 int size = new Integer(line);
68:                 // Initalize a new square graph.
69:                 setGraph(new Graph(size));
70:             } else {
```

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