```java
import components.map.Map;

/**
 *
 * This program processes a glossary text file to generate an HTML glossary. It
 * reads a list of terms and their definitions, sorts them, and creates an HTML
 * index along with individual pages for each term.
 *
 * @author David Park
 *
 */
public final class Glossary {

    /**
     * No argument constructor--private to prevent instantiation.
     */
    private Glossary() {
    }

    /**
     * Processes the input file and writes sorted glossary terms and their
     * definitions to HTML files.
     *
     * @param inputFilePath
     *            the path to the glossary input file
     * @param outputFolderPath
     *            the directory path where HTML files will be saved
     */
    public static void generateHTMLPages(String inputFilePath,
            String outputFolderPath) {
        SimpleReader inFile = new SimpleReader1L(inputFilePath);
        Map<String, String> dictionary = new Map1L<>();
        readTerms(inFile, dictionary);
        inFile.close();

        Queue<String> sortedTerms = sortTerms(dictionary);
        writeIndexHtml(sortedTerms, outputFolderPath);
        writeTermPages(dictionary, outputFolderPath);
    }

    /**
     * Reads terms and their definitions from a file and stores them in a map.
     *
     * @param inFile
     *            the SimpleReader to read from the input file
     * @param dictionary
     *            the map to store terms and definitions
     */
    public static void readTerms(SimpleReader inFile,
            Map<String, String> dictionary) {
        boolean hasActiveKey = false;
        String key = "";
        StringBuilder value = new StringBuilder();
        // Continue reading lines until the end of the file is reached
        while (!inFile.atEOS()) {
            String line = inFile.nextLine();
            // If the line is empty (a blank line)
```

```
69              if (line.trim().isEmpty()) {
70                  // If there is an active key, add the key-value pair to the dictionary
71                  if (hasActiveKey) {
72                      dictionary.add(key, value.toString().trim());
73                      hasActiveKey = false;
74                      value = new StringBuilder();
75                  }
76              } else if (line.indexOf(' ') == -1) {
77                  // if key is found but the previous key is still active
78                  if (hasActiveKey) {
79                      dictionary.add(key, value.toString().trim());
80                      value = new StringBuilder();
81                  }
82                  key = line.trim();
83                  hasActiveKey = true;
84              } else {
85                  value.append(line).append(" ");
86              }
87          }
88          // If active key, add the last key-value pair
89          if (hasActiveKey) {
90              dictionary.add(key, value.toString().trim());
91          }
92      }
93
94      /**
95       * Sorts the terms in alphabetical order using a sequence.
96       *
97       * @param dictionary
98       *            the map containing terms and their definitions
99       * @return a queue containing the sorted terms
100      */
101     public static Queue<String> sortTerms(Map<String, String> dictionary) {
102         Sequence<String> terms = new Sequence1L<>();
103         for (Map.Pair<String, String> term : dictionary) {
104             int position = 0;
105             // Find the correct position to insert in sorted order
106             while (position < terms.length()
107                     && terms.entry(position).compareTo(term.key()) < 0) {
108                 position++;
109             }
110             terms.add(position, term.key()); // Insert the term at the found position
111         }
112
113         // Now transfer the sorted terms from the Sequence to the Queue
114         Queue<String> sortedTerms = new Queue1L<>();
115         while (terms.length() > 0) {
116             sortedTerms.enqueue(terms.remove(0));
117             // Remove from the sequence and enqueue to the queue
118         }
119
120         return sortedTerms; // Return the queue with sorted terms
121     }
122
123     /**
124      * Writes the index HTML file containing links to each term's page.
125      *
```

```java
     * @param terms
     *            the queue of sorted terms
     * @param outputFolderPath
     *            the path where the index.html file will be written
     */
    public static void writeIndexHtml(Queue<String> terms,
            String outputFolderPath) {
        // Create a new SimpleWriter to write the index.html file
        SimpleWriter indexFile = new SimpleWriter1L(
                outputFolderPath + "/index.html");
        // Write the opening HTML tags and the head section
        indexFile.println(
                "<html><head><title>Glossary Index</title></head><body>");
        // Write the main heading for the index page
        indexFile.println("<h1>Glossary Index</h1><ul>");
        // Iterate over each term in the queue
        for (String term : terms) {
            indexFile.println(
                    "<li><a href=\"" + term + ".html\">" + term + "</a></li>");
        }
        // Write the closing HTML tags
        indexFile.println("</ul></body></html>");
        indexFile.close();
    }

    /**
     * Writes HTML pages for each term with their definitions formatted.
     *
     * @param dictionary
     *            the map containing terms and their definitions
     * @param outputFolderPath
     *            the path where term HTML files will be written
     */
    public static void writeTermPages(Map<String, String> dictionary,
            String outputFolderPath) {
        // Iterate over each entry in the dictionary
        for (Map.Pair<String, String> entry : dictionary) {
            // Create a new SimpleWriter to write the HTML file for the current term
            SimpleWriter termFile = new SimpleWriter1L(
                    outputFolderPath + "/" + entry.key() + ".html");
            // Write the opening HTML tags and the head section
            termFile.println("<html><head><title>" + entry.key()
                    + "</title></head><body>");
            // Write the term as a styled heading
            termFile.println(
                    "<h1 style='color:red; font-weight:bold; font-style:italic;'>"
                            + entry.key() + "</h1>");
            // Write the formatted definition of the term
            termFile.println("<p>" + formatDefinition(entry.value(), dictionary)
                    + "</p>");
            // Write a link to return to the index page
            termFile.println(
                    "<p>Return to <a href=\"index.html\">Index</a>.</p>");
            // Write the closing HTML tags
            termFile.println("</body></html>");
            termFile.close();
        }
```

```
183     }
184
185     /**
186      * Formats the definition of a term by embedding hyperlinks to referenced
187      * terms within the glossary. This ensures that each term within the
188      * definition that matches a term in the glossary is clickable and links to
189      * the respective term page.
190      *
191      * @param definition
192      *            the definition to format
193      * @param dictionary
194      *            the map containing all terms and their definitions for lookup
195      * @return the formatted definition with HTML hyperlinks embedded
196      */
197     public static String formatDefinition(String definition,
198             Map<String, String> dictionary) {
199         Set<String> terms = new Set1L<>();
200         // Iterate over each entry in the dictionary and add the terms to the set
201         for (Map.Pair<String, String> entry : dictionary) {
202             terms.add(entry.key()); // Collect all terms for quick lookup
203         }
204
205         // Split definition into words considering punctuation and spaces
206         String[] words = definition.split("\\s+");
207
208         StringBuilder formatted = new StringBuilder();
209         for (int i = 0; i < words.length; i++) {
210             String fragment = words[i];
211             String cleanedFragment = fragment.replaceAll("[^a-zA-Z]", "");
212             // Clean word of punctuation
213
214             if (terms.contains(cleanedFragment)) {
215                 // Replace the term with a hyperlink only if it's a standalone term
216                 formatted.append(fragment.replace(cleanedFragment,
217                         "<a href=\"" + cleanedFragment + ".html\">"
218                                 + cleanedFragment + "</a>"));
219             } else {
220                 formatted.append(fragment);
221             }
222             // Append a space after each word
223             formatted.append(' ');
224         }
225         // Return the formatted definition string after trim
226         return formatted.toString().trim();
227     }
228
229     /**
230      * Main method.
231      *
232      * @param args
233      *            the command line arguments
234      */
235     public static void main(String[] args) {
236         SimpleReader in = new SimpleReader1L();
237         SimpleWriter out = new SimpleWriter1L();
238
239         // call the path to glossary input file
```

```
240         out.print("Enter the path to the glossary input file: ");
241         String inputFilePath = in.nextLine();
242         // get output folder you are going to output to.
243         out.print("Enter the path to the output folder: ");
244         String outputFolderPath = in.nextLine();
245 
246         // run generateHTML
247         generateHTMLPages(inputFilePath, outputFolderPath);
248 
249         in.close();
250         out.close();
251     }
252 }
253 
```