```java
import java.util.Comparator;

import components.map.Map;
import components.map.Map1L;
import components.queue.Queue;
import components.queue.Queue1L;
import components.set.Set;
import components.set.Set1L;
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.simplewriter.SimpleWriter;
import components.simplewriter.SimpleWriter1L;

/**
 * A glossary generator program that creates HTML documentation from text input.
 *
 * <p>
 * Features:
 * <ul>
 * <li>Generates index page with sorted term list</li>
 * <li>Creates individual term pages with hyperlinked definitions</li>
 * <li>Handles multi-line definitions</li>
 * <li>Case-insensitive alphabetical sorting</li>
 * </ul>
 *
 * @author Jeng Zhuang
 */
public final class Glossary {

    /**
     * Private constructor to prevent instantiation.
     */
    private Glossary() {
    }

    /**
     * Generates the main index page listing the terms.
```

```
38      *
39      * @param glossary
40      *            the map containing term-definition pairs
41      * @param outputFolder
42      *            target directory for generated files
43      * @requires outputFolder is a valid directory
44      * @ensures creates 'index.html' in outputFolder with
    sorted term list
45      */
46     private static void generateIndexPage(Map<String, String>
    glossary,
47             String outputFolder) {
48         SimpleWriter out = new SimpleWriter1L(outputFolder + "/
    index.html");
49
50         // Basic structure for the index HTML page
51         out.println("<html>");
52         out.println("<head>");
53         out.println("<title>Glossary</title>");
54         out.println("</head>");
55         out.println("<body>");
56         out.println("<h2>Glossary</h2>");
57         out.println("<hr>");
58         out.println("<h3>Index</h3>");
59         out.println("<ul>");
60
61         // Get and sort terms
62         Set<String> keys = new Set1L<>();
63         for (Map.Pair<String, String> pair : glossary) {
64             keys.add(pair.key());
65         }
66         sortSet(keys, new StringLT());
67
68         // Generate list items
69         for (String term : keys) {
70             out.println("<li><a href=\"" + term + ".html\">" +
    term + "</a></li>");
71         }
72
```

```
 73         out.println("</ul>");
 74         out.println("</body>");
 75         out.println("</html>");
 76         out.close();
 77     }
 78 
 79     /**
 80      * Sorts a set of strings using the specified comparator.
 81      *
 82      * @param set
 83      *            the set to sort
 84      * @param comp
 85      *            the comparator defining the order
 86      * @requires comp != null
 87      * @ensures set's elements are ordered according to comp
 88      */
 89     public static void sortSet(Set<String> set,
    Comparator<String> comp) {
 90         Queue<String> tempQueue = new Queue1L<>();
 91         // Move elements to temporary queue
 92         while (set.size() > 0) {
 93             tempQueue.enqueue(set.removeAny());
 94         }
 95         // Sort using provided comparator
 96         tempQueue.sort(comp);
 97         // Restore sorted elements
 98         while (tempQueue.length() > 0) {
 99             set.add(tempQueue.dequeue());
100         }
101     }
102 
103     /**
104      * Processes the definition to insert hyperlinks for
    glossary terms.
105      *
106      * @param definition
107      *            the original definition
108      * @param glossary
109      *            map of all glossary terms
```

```
110      * @return definition with terms wrapped in anchor tags
111      * @requires glossary != null and definition != null
112      */
113     public static String processDefinition(String definition,
114             Map<String, String> glossary) {
115         Set<String> keys = new Set1L<>();
116         for (Map.Pair<String, String> pair : glossary) {
117             keys.add(pair.key());
118         }
119         // Critical: Sort by length first to prioritize longer
    terms
120         sortSet(keys, new StringLengthDesc());
121 
122         Queue<String> words = new Queue1L<>();
123         int i = 0;
124         // Split definition into tokens (words and separators)
125         while (i < definition.length()) {
126             // Capture non-word characters
127             while (i < definition.length()
128                     && !
    Character.isLetterOrDigit(definition.charAt(i))) {
129 
    words.enqueue(Character.toString(definition.charAt(i)));
130                 i++;
131             }
132 
133             // Capture word characters
134             int start = i;
135             while (i < definition.length()
136                     &&
    Character.isLetterOrDigit(definition.charAt(i))) {
137                 i++;
138             }
139             if (start < i) {
140                 words.enqueue(definition.substring(start, i));
141             }
142         }
143 
144         // Replace terms with hyperlinks
```

```
145         Queue<String> processed = new Queue1L<>();
146         while (words.length() > 0) {
147             String word = words.dequeue();
148             if (keys.contains(word)) {
149                 processed.enqueue("<a href=\"" + word +
    ".html\">" + word + "</a>");
150             } else {
151                 processed.enqueue(word);
152             }
153         }
154 
155         // Build final string
156         StringBuilder result = new StringBuilder();
157         for (String s : processed) {
158             result.append(s);
159         }
160         return result.toString();
161     }
162 
163     /**
164      * Generates an HTML page for a single glossary term.
165      *
166      * @param term
167      *            the term to document
168      * @param definition
169      *            the term's definition
170      * @param glossary
171      *            complete glossary data
172      * @param outputFolder
173      *            target directory for output
174      * @requires term exists in glossary
175      * @ensures creates [term].html in outputFolder
176      */
177     private static void generateTermPage(String term, String
    definition,
178             Map<String, String> glossary, String outputFolder)
    {
179         SimpleWriter out = new SimpleWriter1L(outputFolder +
    "/" + term + ".html");
```

```
180
181         // Page header
182         out.println("<html>");
183         out.println("<head>");
184         out.println("<title>" + term + "</title>");
185         out.println("</head>");
186         out.println("<body>");
187
188         // Term title with styling
189         out.println("<h2><b><i><font color=\"red\">" + term +
    "</font></i></b></h2>");
190         out.println("<blockquote>");
191
192         // Processed definition content
193         String processedDef = processDefinition(definition,
    glossary);
194         out.println(processedDef);
195
196         // Page footer
197         out.println("</blockquote>");
198         out.println("<hr>");
199         out.println("<p>Return to <a
    href=\"index.html\">index</a>.</p>");
200         out.println("</body>");
201         out.println("</html>");
202         out.close();
203     }
204
205     /**
206      * Main method.
207      *
208      * @param args
209      *             command-line arguments
210      */
211     public static void main(String[] args) {
212         SimpleReader in = new SimpleReader1L();
213         SimpleWriter out = new SimpleWriter1L();
214
215         // Get file paths
```

```
216         out.print("Enter input file: ");
217         String inputFile = in.nextLine();
218         out.print("Enter output folder: ");
219         String outputFolder = in.nextLine();
220 
221         // Read glossary data
222         SimpleReader fileReader = new
    SimpleReader1L(inputFile);
223         Map<String, String> glossary = new Map1L<>();
224 
225         String currentTerm = null;
226         StringBuilder currentDefinition = new StringBuilder();
227 
228         // Parse input file
229         while (!fileReader.atEOS()) {
230             String line = fileReader.nextLine();
231             if (line.equals("")) {
232                 // Term-definition separator
233                 if (currentTerm != null) {
234                     glossary.add(currentTerm,
    currentDefinition.toString().trim());
235                     currentTerm = null;
236                     currentDefinition = new StringBuilder();
237                 }
238             } else {
239                 if (currentTerm == null) {
240                     // New term line
241                     currentTerm = line;
242                 } else {
243                     // Append to current definition
244                     if (currentDefinition.length() > 0) {
245                         currentDefinition.append(" ");
246                     }
247                     currentDefinition.append(line);
248                 }
249             }
250         }
251         // Handle final entry
252         if (currentTerm != null) {
```

```
253             glossary.add(currentTerm,
    currentDefinition.toString().trim());
254         }
255         fileReader.close();
256 
257         // Generate documentation
258         generateIndexPage(glossary, outputFolder);
259 
260         Set<String> terms = new Set1L<>();
261         for (Map.Pair<String, String> pair : glossary) {
262             terms.add(pair.key());
263         }
264 
265         for (String term : terms) {
266             String def = glossary.value(term);
267             generateTermPage(term, def, glossary,
    outputFolder);
268         }
269 
270         in.close();
271         out.close();
272     }
273 
274     /**
275      * Case-insensitive alphabetical string comparator.
276      */
277     public static final class StringLT implements
    Comparator<String> {
278         /**
279          * {@inheritDoc}
280          *
281          * @return negative, zero, or positive if s1 is before,
    equal, or after
282          *         s2
283          */
284         @Override
285         public int compare(String s1, String s2) {
286             return s1.compareToIgnoreCase(s2);
287         }
```

```
288     }
289
290     /**
291      * Length-descending string comparator. Prevents partial
    matches by
292      * processing longer terms first.
293      */
294     public static final class StringLengthDesc implements
    Comparator<String> {
295         /**
296          * {@inheritDoc}
297          *
298          * @return positive if s2 is longer, negative if s1 is
    longer
299          */
300         @Override
301         public int compare(String s1, String s2) {
302             return Integer.compare(s2.length(), s1.length());
303         }
304     }
305 }
306
```