```java
import java.util.Comparator;

/**
 * Reads a file of words and their definitions and generates an HTML page for
 * each term with its definition, plus an HTML index page with links to the
 * individual term pages.
 *
 * @author Taylor Keckley
 *
 */
public final class Glossary {

    /**
     * Compare {@code String}s in lexicographic order.
     */
    private static class StringLT implements Comparator<String> {
        @Override
        public int compare(String s1, String s2) {
            return s1.compareTo(s2);
        }
    }

    /**
     * Private constructor so this utility class cannot be instantiated.
     */
    private Glossary() {
    }

    /**
     * Generates the set of characters in the given {@code String} into the
     * given {@code Set}.
     *
     * @param str
     *            the given {@code String}
     * @param strSet
     *            the {@code Set} to be replaced
     * @replaces strSet
     * @ensures strSet = entries(str)
     */
    public static void generateElements(String str, Set<Character> strSet) {
        assert str != null : "Violation of: str is not null";
        assert strSet != null : "Violation of: strSet is not null";

        strSet.clear();

        for (int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);

            /*
             * Puts each character in str in strSet as long as it isn't already
             * in the set.
             */
            if (!strSet.contains(ch)) {
                strSet.add(ch);
            }
```

```
67          }
68      }
69
70      /**
71       * Inputs a "glossary" of terms and their definitions from the given input
72       * and stores them in the given map.
73       *
74       * @param input
75       *            the file input stream
76       * @param termDefMap
77       *            the term -> definition map
78       * @replaces termDefMap
79       * @requires <pre>
80       * [the input stream  is open, and has the format of one term (unique in
81       * the file) on the first line and its definition on the next one or more
82       * lines (terminated by an empty line), another term on the next line, its
83       * definition on the next one or more lines (terminated by an empty line),
84       * etc. The input continues in this fashion through the definition of the
85       * last term, which shall end with its terminating empty line.]
86       * </pre>
87       * @ensures termDefinitionMap contains term -> definition mapping from the
88       *          input file.
89       */
90      public static void getTermDefinitionMap(SimpleReader input,
91              Map<String, String> termDefMap) {
92          assert termDefMap != null : "Violation of: termDefMap is not null";
93          assert input.isOpen() : "Violation of: input.is_open";
94
95          while (!input.atEOS()) {
96              // Gets the term and the following line
97              String term = input.nextLine();
98              String line = input.nextLine();
99              StringBuilder definition = new StringBuilder();
100
101             // Adds lines to the definition until it reaches an empty line.
102             while (!line.contentEquals("")) {
103
104                 definition.append(line);
105
106                 line = input.nextLine();
107
108                 if (!line.contentEquals("")) {
109                     if (line.charAt(0) != ' ' && definition
110                             .charAt(definition.length() - 1) != ' ') {
111                         /*
112                          * Appends a space if the definition continues onto
113                          * another line, the definition string doesn't end with
114                          * a space, and the line string doesn't begin with a
115                          * space.
116                          */
117                         definition.append(" ");
118                     }
119                 }
120             }
121
```

```
122             // Adds the term and its associated definition to the map
123             termDefMap.add(term, definition.toString());
124         }
125
126     }
127
128     /**
129      * Outputs the "opening" tags in the generated HTML file. These are the
130      * expected elements generated by this method:
131      *
132      * <html> <head> <title>the page title</title> </head> <body>
133      * <h2>the page title</h2>
134      * <hr />
135      * <h3>Index</h3>
136      * <ul>
137      *
138      * @param out
139      *            the output stream
140      * @param title
141      *            the title of the html page
142      * @updates out.content
143      * @requires out.is_open
144      * @ensures out.content = #out.content * [the HTML "opening" tags]
145      */
146     public static void outputHeader(SimpleWriter out, String title) {
147         assert out != null : "Violation of: out is not null";
148         assert out.isOpen() : "Violation of: out.is_open";
149
150         out.println(
151                 "<html> <head> <title>" + title + "</title> </head> <body>");
152         out.println("<h2>" + title + "</h2>");
153         out.println("<hr />");
154         out.println("<h3>Index</h3>");
155         out.println("<ul>");
156     }
157
158     /**
159      * Outputs the "closing" tags in the generated HTML file. These are the
160      * expected elements generated by this method:
161      *
162      * </ul>
163      * </body> </html>
164      *
165      * @param out
166      *            the output stream
167      * @updates out.contents
168      * @requires out.is_open
169      * @ensures out.content = #out.content * [the HTML "closing" tags]
170      */
171     public static void outputFooter(SimpleWriter out) {
172         assert out != null : "Violation of: out is not null";
173         assert out.isOpen() : "Violation of: out.is_open";
174
175         out.println("</ul>");
176         out.println("</body> </html>");
```

```
177     }
178
179     /**
180      * Removes and returns the minimum value from stringMap according to
181      * lexicographic order.
182      *
183      * @param stringMap
184      *            the map
185      * @return the minimum value from stringMap
186      * @updates stringMap
187      * @requires <pre>
188      *  stringMap is not null and
189      *  [the relation computed by order.compare is a total preorder]
190      * </pre>
191      * @ensures <pre>
192      *  removeMin removes and returns the key from stringMap that occurs
193      *  first in lexicographic order. min is in #stringMap.
194      * </pre>
195      */
196     public static String removeMin(Map<String, String> stringMap) {
197         Map<String, String> tmp = stringMap.newInstance();
198         tmp.transferFrom(stringMap);
199
200         Queue<String> sortQ = new Queue1L<>();
201
202         // Moves all of the keys from tmp to sortQ
203         while (tmp.size() > 0) {
204             Map.Pair<String, String> tmpPair = tmp.removeAny();
205             String key = tmpPair.key();
206             String value = tmpPair.value();
207
208             sortQ.enqueue(key);
209             stringMap.add(key, value);
210         }
211
212         // Puts queue of keys in lexicographic order
213         Comparator<String> order = new StringLT();
214         sortQ.sort(order);
215
216         // Returns the first (minimum) value of sortQ
217         return sortQ.dequeue();
218     }
219
220     /**
221      * Outputs each term from termDefMap as an item in a list. Each term
222      * contains a link to its term page with its associated definition.
223      *
224      * @param out
225      *            the output stream
226      * @param termDefMap
227      *            the term -> definition map
228      * @updates out.contents
229      * @requires the output stream is open and termDefMap is not null.
230      * @ensures out.content = #out.content * [a list of all the terms from
231      *          termDefMap each with a link to their corresponding term page.]
```

```
232      */
233     public static void outputListItems(SimpleWriter out,
234             Map<String, String> termDefMap) {
235
236         Map<String, String> tmp = termDefMap.newInstance();
237         tmp.transferFrom(termDefMap);
238
239         while (tmp.size() > 0) {
240             /*
241              * Removes the term that occurs first lexicographically.
242              */
243             String min = removeMin(tmp);
244             Map.Pair<String, String> termDefPair = tmp.remove(min);
245             String term = termDefPair.key();
246             String definition = termDefPair.value();
247
248             /*
249              * Prints one term containing a link to its term page as a list
250              * item.
251              */
252             out.println("<li>");
253             out.println("<a href=\"" + term + ".html\">" + term + "</a></li>");
254
255             /*
256              * Restores termDefMap
257              */
258             termDefMap.add(term, definition);
259         }
260
261     }
262
263     /**
264      * Outputs an HTML page with a term and its definition for every term in
265      * termDefMap. If there are any words in the definition that are also in the
266      * glossary (termDefMap), they will have a link to the page for that term
267      * and its associated definition. Includes a link back to the original
268      * index.
269      *
270      * @param folder
271      *            the folder each term page should be put in
272      * @param termDefMap
273      *            the term -> definition map
274      * @param separators
275      *            a set of separator characters
276      * @updates out.contents
277      * @requires the output stream is open and termDefMap and separators are not
278      *           null.
279      * @ensures outputs a term page (file) for every term in termDefMap. Words
280      *          in the definition that are also in the glossary will have a link
281      *          to their term page. There is a link back to the index.
282      */
283     public static void outputTermPages(String folder,
284             Map<String, String> termDefMap, Set<Character> separators) {
285
286         Map<String, String> tmp = termDefMap.newInstance();
```

```
287         tmp.transferFrom(termDefMap);
288
289         while (tmp.size() > 0) {
290             // Removes one term and its associated definition from tmp
291             Map.Pair<String, String> termDefPair = tmp.removeAny();
292             String term = termDefPair.key();
293             String definition = termDefPair.value();
294
295             // Restores termDefMap
296             termDefMap.add(term, definition);
297
298             // Creates an output stream that writes to a term page
299             SimpleWriter out = new SimpleWriter1L(
300                     folder + "/" + term + ".html");
301
302             /*
303              * Prints the opening tags of the term page where the title is the
304              * term in red boldface italics.
305              */
306             out.println(
307                     "<html> <head> <title>" + term + "</title> </head> <body>");
308             out.println("<h2><b><i><font color=\"red\">" + term
309                     + "</font></i></b></h2>");
310
311             outputDefinition(out, definition, termDefMap, separators);
312
313             out.println("<hr />");
314
315             // Prints a link to return to the index
316             out.println("<p>Return to <a href=\"index.html\">index</a>.</p>");
317
318             // Prints the closing tags
319             out.println("</body> </html>");
320
321             // Closes the term page output stream
322             out.close();
323         }
324
325     }
326
327     /**
328      * Outputs the given definition. Any words in that definition that are also
329      * in the glossary contain links to their corresponding term pages.
330      *
331      * @param out
332      *            the output stream
333      * @param definition
334      *            the definition to be output
335      * @param termDefMap
336      *            the term -> definition map
337      * @param separators
338      *            a set of separator characters
339      * @updates out.contents
340      * @requires the output stream is open, and definition, termDefMap, and
341      *           separators are not null.
```

```
342      * @ensures out.content = #out.content * [The definition. Words in the
343      *          definition that are also in the glossary have links to their
344      *          corresponding term pages.]
345      */
346     public static void outputDefinition(SimpleWriter out, String definition,
347             Map<String, String> termDefMap, Set<Character> separators) {
348
349         out.println("<blockquote>");
350         int pos = 0;
351         while (pos < definition.length()) {
352             /*
353              * Finds the next word or separator in the definition starting at
354              * position pos.
355              */
356             String token = nextWordOrSeparator(definition, pos, separators);
357
358             /*
359              * If the token is a term (key) in termDefMap, print the token with
360              * a link to its term page. Otherwise just print the token.
361              */
362             if (termDefMap.hasKey(token)) {
363                 out.print("<a href=\"" + token + ".html\">" + token + "</a>");
364             } else {
365                 out.print(token);
366             }
367             /*
368              * Increase the position so it starts looking for another word or
369              * separator after the current token.
370              */
371             pos += token.length();
372         }
373         out.println("</blockquote>");
374     }
375
376     /**
377      * Returns the first "word" (maximal length string of characters not in
378      * {@code separators}) or "separator string" (maximal length string of
379      * characters in {@code separators}) in the given {@code text} starting at
380      * the given {@code position}.
381      *
382      * @param text
383      *            the {@code String} from which to get the word or separator
384      *            string
385      * @param position
386      *            the starting index
387      * @param separators
388      *            the {@code Set} of separator characters
389      * @return the first word or separator string found in {@code text} starting
390      *         at index {@code position}
391      * @requires 0 <= position < |text|
392      * @ensures <pre>
393      * nextWordOrSeparator =
394      *   text[position, position + |nextWordOrSeparator|)  and
395      * if entries(text[position, position + 1)) intersection separators = {}
396      * then
```

```
397      *   entries(nextWordOrSeparator) intersection separators = {}  and
398      *   (position + |nextWordOrSeparator| = |text|  or
399      *    entries(text[position, position + |nextWordOrSeparator| + 1))
400      *      intersection separators /= {})
401      * else
402      *   entries(nextWordOrSeparator) is subset of separators  and
403      *   (position + |nextWordOrSeparator| = |text|  or
404      *    entries(text[position, position + |nextWordOrSeparator| + 1))
405      *      is not subset of separators)
406      * </pre>
407      */
408     public static String nextWordOrSeparator(String text, int position,
409             Set<Character> separators) {
410         assert text != null : "Violation of: text is not null";
411         assert separators != null : "Violation of: separators is not null";
412         assert 0 <= position : "Violation of: 0 <= position";
413         assert position < text.length() : "Violation of: position < |text|";
414
415         StringBuilder result = new StringBuilder();
416         char ch = text.charAt(position);
417         int i = position;
418
419         /*
420          * The if statement executes if the character at position is in
421          * separators, which means the result will be a separator. The else
422          * statement executes if the character at position is not in separators,
423          * which means the result will be a word.
424          */
425         if (separators.contains(ch)) {
426             while (i < text.length() && separators.contains(text.charAt(i))) {
427                 ch = text.charAt(i);
428                 result.append(ch);
429                 i++;
430             }
431         } else {
432             while (i < text.length() && !separators.contains(text.charAt(i))) {
433                 ch = text.charAt(i);
434                 result.append(ch);
435                 i++;
436             }
437         }
438         return result.toString();
439     }
440
441     /**
442      * Main method.
443      *
444      * @param args
445      *            the command line arguments
446      */
447     public static void main(String[] args) {
448         SimpleReader in = new SimpleReader1L();
449         SimpleWriter out = new SimpleWriter1L();
450
451         out.print("Enter an input file: ");
```

```
452         String fileName = in.nextLine();
453
454         out.print("Enter the name of a folder: ");
455         String folder = in.nextLine();
456
457         /*
458          * Creates a set of separator characters.
459          */
460         String separatorsString = "\t, \n:;.?!-/";
461         Set<Character> separators = new Set1L<>();
462         generateElements(separatorsString, separators);
463
464         /*
465          * Opens the file containing the terms and definitions then puts the
466          * those in a map.
467          */
468         SimpleReader inputFile = new SimpleReader1L(folder + "/" + fileName);
469         Map<String, String> termDefMap = new Map1L<>();
470         getTermDefinitionMap(inputFile, termDefMap);
471
472         /*
473          * Opens an output stream that writes an html header in the top-level
474          * index of the glossary.
475          */
476         SimpleWriter indexWriter = new SimpleWriter1L(folder + "/index.html");
477         outputHeader(indexWriter, "Glossary");
478
479         /*
480          * Outputs each definition as a list item in the glossary index.
481          */
482         outputListItems(indexWriter, termDefMap);
483
484         /*
485          * Outputs a page for every term with the term and its associated
486          * definition.
487          */
488         outputTermPages(folder, termDefMap, separators);
489
490         /*
491          * Outputs the footer of the index.
492          */
493         outputFooter(indexWriter);
494
495         /*
496          * Close input and output streams
497          */
498         inputFile.close();
499         indexWriter.close();
500         in.close();
501         out.close();
502     }
503
504 }
505
```