```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;

/**
 * This program is designed to create an easy-to-maintain glossary facility,
 * following the instructions of the customer Cy Burnett.
 *
 * @author L. Oden
 *
 */
public final class Glossary {

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

    /**
     * Returns the first "word" (maximal length string of characters not in
     * {@code separators}) or "separator string" (maximal length string of
     * characters in {@code separators}) in the given {@code text} starting at
     * the given {@code position}.
     *
     * @param text
```

```
36      *            the {@code String} from which to get the
  word or separator
37      *            string
38      * @param position
39      *            the starting index
40      * @param separators
41      *            the {@code Set} of separator characters
42      * @return the first word or separator string found in
  {@code text} starting
43      *         at index {@code position}
44      * @requires 0 <= position < |text|
45      * @ensures <pre>
46      * nextWordOrSeparator =
47      *   text[position, position + |nextWordOrSeparator|)
  and
48      * if entries(text[position, position + 1)) intersection
  separators = {}
49      * then
50      *   entries(nextWordOrSeparator) intersection separators
  = {}  and
51      *   (position + |nextWordOrSeparator| = |text|  or
52      *    entries(text[position, position + |
  nextWordOrSeparator| + 1))
53      *      intersection separators /= {})
54      * else
55      *   entries(nextWordOrSeparator) is subset of separators
  and
56      *   (position + |nextWordOrSeparator| = |text|  or
57      *    entries(text[position, position + |
  nextWordOrSeparator| + 1))
58      *      is not subset of separators)
59      * </pre>
60      */
61     public static String nextWordOrSeparator(String text, int
  position,
62             Set<Character> separators) {
63         int p = position;
64         /*
65          * isSeperator determines if the character at
  position p in text is part
```

```
66          * of separator set
67          */
68         boolean isSeparator =
   separators.contains(text.charAt(p));
69         /*
70          * While within text length and each character is
   still its initial
71          * condition (either a separator or not), keep
   incrementing p.
72          */
73         while (p < text.length()
74                 && separators.contains(text.charAt(p)) ==
   isSeparator) {
75             p++;
76         }
77         /*
78          * Return substring of text from starting position to
   position reached
79          * after ending loop p.
80          */
81         return text.substring(position, p);
82     }
83
84     /**
85      * Returns a Map of the lines read from {@code input},
   where the key (term)
86      * is a 1-word, 1-line String, and the value (definition)
   is a several line,
87      * multiple word String. The map cannot have duplicate
   keys (terms).
88      *
89      * @param input
90      *             source of strings, one per line for keys
   (terms), and one per
91      *             several lines for values (definitions).
92      * @return Map of several terms and definitions read from
   the lines of
93      *         {@code input}, where each Map pair is
   separated by an empty line
94      *         in the input file.
```

```
 95      * @requires input.is_open
 96      * @ensures <pre>
 97      * input.is_open  and  input.content = <>  and
 98      * linesFromInput = [maximal Map of different lines from
    #input.content such that
 99      *                    CONTAINS_NO_DUPLICATE_KEYS
    (linesFromInput)]
100      * </pre>
101      */
102     public static Map<String, String>
    mapFromInputLines(SimpleReader input) {
103         assert input != null : "Violation of: input is not
    null";
104         assert input.isOpen() : "Violation of:
    input.is_open";
105
106         // Initialize new set to hold all input lines.
107         Map<String, String> inputLinesMap = new Map1L<>();
108
109         /*
110          * While we're not at the end of stream, initialize
    term to the initial
111          * line that SimpleReader input reads. Then,
    initialize definition by
112          * adding each line that is not Empty. Finally, if
    the key is not
113          * already in the Map, then add the Map pair of the
    term and definition
114          * to the Map.
115          */
116         while (!input.atEOS()) {
117             String term = input.nextLine();
118             String currentLine = input.nextLine();
119             String definition = "";
120             while (!currentLine.isEmpty()) {
121                 definition += currentLine;
122                 currentLine = input.nextLine();
123             }
124             if (!inputLinesMap.hasKey(term)) {
125                 inputLinesMap.add(term, definition);
```

```
126                 }
127             }
128             // Return the set inputLinesSet.
129             return inputLinesMap;
130         }
131 
132         /**
133          *
134          * Returns a sorted Queue, holding the values of the keys
    of mapToSort. This
135          * is based on the alphabetical order of the keys in each
    Map.Pair of the
136          * input Map. This order is found by using a created
    String comparator. This
137          * can be used later to get each Map.Pair from the Map in
    alphabetical
138          * order.
139          *
140          * @param mapToSort
141          *            The map of keys (terms) and values
    (definitions) that is used
142          *            for sorting.
143          *
144          * @return The sorted Queue<String> of the mapToSort keys
145          *
146          * @requires mapToSort is not null
147          *
148          * @ensures Output Queue contains the keys from the input
    Map in
149          *          alphabetical order.
150          */
151         public static Queue<String> sortingKeys(Map<String,
    String> mapToSort) {
152             // Initialize String comparator using implementation
    in this java file.
153             Comparator<String> sort = new StringLT();
154             // Create temp Map variable as new instance, and
    transferFrom mapToSort
155             Map<String, String> temp = mapToSort.newInstance();
156             temp.transferFrom(mapToSort);
```

```
157         /*
158          * Initialize keys to be a new Queue which will hold the value of the
159          * keys of temp
160          */
161         Queue<String> keys = new Queue1L<>();
162         /*
163          * While temp still has elements, remove random Map.Pair from temp and
164          * initialize termPlusDef. Enqueue the key of termPlusDef to keys. Add
165          * these values back to mapToSort every time.
166          */
167         while (temp.size() > 0) {
168             Map.Pair<String, String> termPlusDef = temp.removeAny();
169             keys.enqueue(termPlusDef.key());
170             mapToSort.add(termPlusDef.key(), termPlusDef.value());
171         }
172         /*
173          * Sort the keys Queue using the String comparator initialized above.
174          */
175         keys.sort(sort);
176         // Return sorted Queue.
177         return keys;
178     }
179 
180     /**
181      * Processes a singlePair of term and definition from a Map, printing an
182      * appropriate term & definition page for input Map.Pair to an HTML file
183      * named after the Map.Pair.
184      *
185      * @param singlePair
186      *            A single Map.Pair object from a Map
187      * @param keys
188      *            A queue containing all keys of a Map in
```

```
   alphabetical order.
189      * @param outputFolder
190      *            The folder where all output files are
   stored.
191      * @ensures <pre>
192      * [Saves HTML document with page of a Map.Pair's term
   and definition]
193      * </pre>
194      */
195     public static void processTerm(Queue<String> keys,
196             Map.Pair<String, String> singlePair, String
   outputFolder) {
197         /*
198          * Write code to new HTML page named based on key of
   the input
199          * singlePair
200          */
201         SimpleWriter termPageFileOut = new SimpleWriter1L(
202                 outputFolder + "\\" + singlePair.key() +
   ".html");
203         // Output headers of term HTML file
204         /*
205          * Opening HTML and head tags
206          */
207         termPageFileOut.println("<html>");
208         termPageFileOut.println("<head>");
209         // Opening title tag with key of singlePair as title
   of this term.
210         termPageFileOut.println("<title>" + singlePair.key()
   + "</title>");
211         // Close head tag
212         termPageFileOut.println("</head>");
213         // Open body tag
214         termPageFileOut.println("<body>");
215         // Print header in format specified in project
   instructions.
216         termPageFileOut.println("<h1><em><b
   style='color:red;'>"
217                 + singlePair.key() + "</b></em></h1>");
218         // Open paragraph tag
```

```
219         termPageFileOut.println("<p>");
220 
221         // Initialize separators set. Add various different
    separators of words.
222         Set<Character> separators = new Set1L<>();
223         separators.add(' ');
224         separators.add('.');
225         separators.add(',');
226         separators.add(';');
227         separators.add(':');
228 
229         /*
230          * Starting at position 0, indexTerm has not been
    found yet, and
231          * wordOrSeparator is empty
232          */
233         int position = 0;
234         boolean indexTerm = false;
235         String wordOrSeparator = "";
236         /*
237          * Print  , a single one representing a space in
    HTML, to match the
238          * specified format.
239          */
240         termPageFileOut.print(
241 
    "          
    "
242                 + "  ");
243         // While the position doesn't exceed length of
    description of a pair
244         while (position < singlePair.value().length()) {
245             indexTerm = false;
246             /*
247              * Call wordOrSeparator method to return either a
    full word or full
248              * separator from the description/value
249              */
250             wordOrSeparator =
    nextWordOrSeparator(singlePair.value(), position,
```

```
251                     separators);
252             /*
253              * For each string in keys (sorted Queue), if the wordOrSeparator
254              * equals a value that is a key, then indexTerm is true.
255              */
256             for (String s : keys) {
257                 if (wordOrSeparator.equals(s)) {
258                     indexTerm = true;
259                 }
260             }
261             /*
262              * If indexTerm, then print the particular word as a link to the
263              * HTML page of the term with that name. Otherwise, just print the
264              * word/separator.
265              */
266             if (indexTerm) {
267                 termPageFileOut.print("<a href = \"" + wordOrSeparator
268                         + ".html\">" + wordOrSeparator + "</a>");
269             } else {
270                 termPageFileOut.print(wordOrSeparator);
271             }
272             // Increment position
273             position += wordOrSeparator.length();
274         }
275         // Print closing paragraph header
276         termPageFileOut.println("</p>");
277         // Print horizontal line
278         termPageFileOut.println("<hr>");
279         // Print the return to index. button with a link to index page.
280         termPageFileOut.println(
281                 "<p>Return to <a href = \"index.html\">index</a>.</p>");
282
```

```
283         // Output the footer of term HTML file
284         // Close body
285         termPageFileOut.println("</body>");
286         // Close HTML file
287         termPageFileOut.println("</html>");
288         termPageFileOut.close();
289     }
290 
291     /**
292      * Main method.
293      *
294      * @param args
295      *            the command line arguments
296      */
297     public static void main(String[] args) {
298         SimpleReader inFromConsole = new SimpleReader1L();
299         SimpleWriter outToConsole = new SimpleWriter1L();
300 
301         // Ask for input file and initialize inputFile
302         outToConsole.print("Please enter the name of an input
    file: ");
303         String inputFile = inFromConsole.nextLine();
304         // Ask for output folder and initialize outputFolder.
305         outToConsole.print(
306                 "Please enter the name of an output folder
    where all output "
307                         + "files will be saved: ");
308         String outputFolder = inFromConsole.nextLine();
309 
310         /*
311          * inFromFile reads input from specified file, and
    outToFile writes
312          * output to index.html in the specified folder.
313          */
314         SimpleReader inFromFile = new
    SimpleReader1L(inputFile);
315         SimpleWriter outToFile = new SimpleWriter1L(
316                 outputFolder + "\\index.html");
317 
318         /*
```

```
319         * Initialize termsAndDefinitions to a call to
    mapFromInputLines reading
320         * from specified input file.
321         */
322        /*
323         * Initialize sortedKeys to be a Queue holding the
    keys of
324         * termsAndDefinitions in alphabetical order.
325         */
326        Map<String, String> termsAndDefinitions =
    mapFromInputLines(inFromFile);
327        Queue<String> sortedKeys =
    sortingKeys(termsAndDefinitions);
328
329        // Opening tag of an HTML document
330        outToFile.println("<html>");
331
332        // Opening tag of a head
333        outToFile.println("<head>");
334
335        // Opens title, prints "Index" as title, and closes
    title.
336        outToFile.println("<title>Glossary</title>");
337
338        // Closes the header
339        outToFile.println("</head>");
340
341        // Opens the body
342        outToFile.println("<body>");
343
344        // Opens h1, prints "Index"
345        outToFile.println("<h1>Glossary</h1>");
346
347        // Open second section with horizontal line divider
348        outToFile.println("<hr>");
349
350        // Open second header and print "Index"
351        outToFile.println("<h2>Index</h2>");
352
353        // Opens the bullet point list
```

```
354         outToFile.println("<ul>");
355 
356         // For each string s in sortedKeys (same length as
    termsAndDefinitions)
357         for (String s : sortedKeys) {
358             /*
359              * Single Map.Pair, starting from smallest
    alphabetically is a
360              * result of removing pair from
    termsAndDefinitions at key s.
361              */
362             Map.Pair<String, String> single =
    termsAndDefinitions.remove(s);
363             /*
364              * Call process term to process this single
    Map.Pair and print the
365              * appropriate separate HTML page for it.
366              */
367             processTerm(sortedKeys, single, outputFolder);
368             /*
369              * Creates an unordered list entry, and links the
    Map.Pair key name
370              * to the page with that name.
371              */
372             outToFile.println("<li><a href = \"" +
    single.key() + ".html\">"
373                     + single.key() + "</a></li>");
374         }
375         // Close bullet point list
376         outToFile.println("</ul>");
377         // Close body
378         outToFile.println("</body>");
379         // Close HTML file
380         outToFile.println("</html>");
381 
382         // Print success generation message.
383         outToConsole.println("HTML file successfully
    generated!");
384 
385         /*
```

```
386          * Close input and output streams
387          */
388         inFromConsole.close();
389         outToConsole.close();
390         inFromFile.close();
391         outToFile.close();
392     }
393 
394     /**
395      * Comparator for Strings, implementing
    Comparator<String> and overriding
396      * compare method.
397      */
398     private static class StringLT implements
    Comparator<String> {
399         @Override
400         public int compare(String o1, String o2) {
401             return o1.compareTo(o2);
402         }
403     }
404 
405 }
406 
```