

```
1 import java.util.Comparator;
13
14 /**
15  * Put a short phrase describing the program here.
16  *
17  * @author Nicholas Cheong
18  *
19  */
20 public final class Glossary {
21
22     /**
23      * Private constructor so this utility class cannot be
24      instantiated.
25      */
26     private Glossary() {
27
28     /**
29      *
30      * @author Nicholas Cheong
31      *
32      */
33     private static class StringLT implements Comparator<String>
34     {
35         @Override
36         public int compare(String o1, String o2) {
37             return o1.compareTo(o2);
38         }
39     }
40
41     /**
42      * Inputs a list of words and their definitions from the
43      given file and
44      * stores them in the given {@code Map}.
45      *
46      * @param fileName
47      *         the name of the input file
48      * @param wordDefinition
49      *         the word and definition -> word and
50      definition map
51      * @replaces wordDefinition
52      * @requires <pre>
53      * [file named fileName exists but is not open, and has the
54      * format of one "word" (unique in the file) and one
```

```
definition, with
52     * word and definition separated by ' '
53     * </pre>
54     * @ensures [wordDefinition contains word and definition ->
mapping from
55     *         file fileName]
56     */
57     public static void getMap(String fileName,
58                             Map<String, String> wordDefinition) {
59
60         // reads every line in the file
61         // reads the first line after the space to be the word
(key)
62         // reads every line after to be the corresponding
definition (value)
63         SimpleReader in = new SimpleReader1L(fileName);
64         while (!in.atEOS()) {
65             String thisLine = in.nextLine();
66             String word = thisLine;
67             String definition = "";
68             String nextLine = in.nextLine();
69             while (!nextLine.equals("")) {
70                 definition += nextLine + " ";
71                 nextLine = in.nextLine();
72             }
73
74             wordDefinition.add(word, definition);
75         }
76
77         // close output
78         in.close();
79
80     }
81
82     /**
83     *
84     * @param wordDefinition
85     *         the word and definition -> word and
definition map
86     * @param wordBank
87     *         the word -> word queue
88     * @replaces wordBank
89     * @ensures [wordBank contains all words from
wordDefinition map]
```

```
90     */
91
92     public static void wordQueue(Map<String, String>
wordDefinition,
93         Queue<String> wordBank) {
94
95         Comparator<String> cs = new StringLT();
96
97         // enqueue each word in every pair of the map to the
wordBank queue
98         for (Map.Pair<String, String> p : wordDefinition) {
99             wordBank.enqueue(p.key());
100         }
101
102         wordBank.sort(cs);
103
104     }
105
106     /**
107     * Generates the set of characters in the given {@code
String} into the
108     * given {@code Set}.
109     *
110     * @param str
111     *         the given {@code String}
112     * @param charSet
113     *         the {@code Set} to be replaced
114     * @replaces charSet
115     * @ensures charSet = entries(str)
116     */
117     public static void generateElements(String str,
Set<Character> charSet) {
118         assert str != null : "Violation of: str is not null";
119         assert charSet != null : "Violation of: charSet is not
null";
120
121         Set<Character> tempSet = new Set1L<>();
122
123         for (int i = 0; i < str.length(); i++) {
124             if (!tempSet.contains(str.charAt(i))) {
125                 tempSet.add(str.charAt(i));
126             }
127
128         }
```

```
129         charSet.transferFrom(tempSet);
130     }
131 }
132
133 /**
134  * Returns the first "word" (maximal length string of
135  * characters not in
136  * {@code separators}) or "separator string" (maximal
137  * length string of
138  * characters in {@code separators}) in the given {@code
139  * text} starting at
140  * the given {@code position}.
141  *
142  * @param text
143  *        the {@code String} from which to get the word
144  *        or separator
145  *        string
146  * @param position
147  *        the starting index
148  * @param separators
149  *        the {@code Set} of separator characters
150  * @return the first word or separator string found in
151  *        {@code text} starting
152  *        at index {@code position}
153  * @requires 0 <= position < |text|
154  * @ensures <pre>
155  * nextWordOrSeparator =
156  *   text[position, position + |nextWordOrSeparator|) and
157  *   if entries(text[position, position + 1)) intersection
158  *   separators = {}
159  * then
160  *   entries(nextWordOrSeparator) intersection separators =
161  *   {} and
162  *   (position + |nextWordOrSeparator| = |text| or
163  *   entries(text[position, position + |
164  *   nextWordOrSeparator| + 1))
165  *   intersection separators /= {})
166  * else
167  *   entries(nextWordOrSeparator) is subset of separators
168  * and
169  *   (position + |nextWordOrSeparator| = |text| or
170  *   entries(text[position, position + |
171  *   nextWordOrSeparator| + 1))
172  *   is not subset of separators)
```

```
163     * </pre>
164     */
165     public static String nextWordOrSeparator(String text, int
position,
166         Set<Character> separators) {
167         assert text != null : "Violation of: text is not null";
168         assert separators != null : "Violation of: separators
is not null";
169         assert 0 <= position : "Violation of: 0 <= position";
170         assert position < text.length() : "Violation of:
position < |text|";
171
172         String result = "";
173         int positionCopy = position;
174
175         if (!separators.contains(text.charAt(positionCopy))) {
176             while (positionCopy < text.length()
177                 && !
separators.contains(text.charAt(positionCopy))) {
178
179                 result += text.charAt(positionCopy);
180                 positionCopy++;
181             }
182         } else {
183             while (positionCopy < text.length()
184                 &&
separators.contains(text.charAt(positionCopy))) {
185                 result += text.charAt(positionCopy);
186                 positionCopy++;
187             }
188         }
189
190         return result;
191     }
192
193     /**
194     * Outputs the "opening" tags in the generated HTML file.
These are the
195     * expected elements generated by this method:
196     *
197     * <html> <head> <title>Glossary</title> </head> <body>
198     * <h2>Glossary</h2>
199     * <hr>
200     * <h3>Index</h3>
```

```
201     * <ul>
202     *
203     * @param out
204     *         the output stream
205     * @updates out.content
206     * @ensures out.content = #out.content * [the HTML
    "opening" tags]
207     */
208     public static void generateIndex(SimpleWriter out) {
209
210         out.println("<html>");
211         out.println("<head>");
212         out.println("<title>");
213         out.println("Glossary");
214         out.println("</title>");
215         out.println("</head>");
216         out.println("<body>");
217         out.println("<h2>Glossary</h2>");
218         out.println("<hr>");
219         out.println("<h3>Index</h3>");
220         out.println("<ul>");
221     }
222
223     /**
224     * Outputs each word in the index HTML and outputs their
    definitions from
225     * wordDefinition map to their own generated HTML file.
    Will also link to
226     * other definitions if word in glossary is used in the
    definition of
227     * another word
228     *
229     * @param wordDefinition
230     *         the word and definition -> word and
    definition map
231     * @param word
232     *         the word imported from wordBank queue
233     * @param definition
234     *         the corresponding definition imported from
    wordDefinition map
235     * @param out
236     *         the output stream
237     * @param outputFile
238     *         the folder where HTML will be stored
```

```
239     * @updates out.content
240     * @ensures out.content = #out.content * word and HTMLS of
    each word with
241     *         corresponding definitions
242     *
243     */
244     public static void generateHTML(Map<String, String>
    wordDefinition,
245         String word, String definition, SimpleWriter out,
246         String outputFile) {
247
248         // creating separator set
249         String separatorStr = " \t,";
250         Set<Character> separatorSet = new Set1L<>();
251         generateElements(separatorStr, separatorSet);
252
253         SimpleWriter outFile = new SimpleWriter1L(
254             outputFile + "/" + word + ".html");
255         String newDefinition = "";
256         String linked = "";
257         int i = 0;
258         // prints word to the index html
259         out.println("<a href = " + word + ".html>");
260         out.println(word);
261         out.println("</a>");
262
263         // prints the definition to the word html
264         outFile.println("<head>");
265         outFile.println("<title>");
266         outFile.println(word);
267         outFile.println("</title>");
268         outFile.println("</head>");
269         outFile.println("<body>");
270         outFile.println("<h2>");
271         outFile.println("<b>");
272         outFile.println("<i>");
273         outFile.println("<font color = \"red\">" + word + "</
    font>");
274         outFile.println("</i>");
275         outFile.println("</b>");
276         outFile.println("</h2>");
277         outFile.println("<blockquote>");
278
279         while (i < definition.length()) {
```

```
280         String link = nextWordOrSeparator(definition, i,
separatorSet);
281
282         if (wordDefinition.containsKey(link)) {
283
284             linked = "<a href = " + link + ".html>" + link
+ "</a>";
285             newDefinition += linked;
286
287         } else {
288             newDefinition += link;
289         }
290         i += link.length();
291     }
292
293     outFile.println(newDefinition);
294
295     outFile.println("</blockquote>");
296     outFile.println("<hr>");
297     outFile.println("<p>");
298     outFile.println("Return to");
299     outFile.println("<a href = index.html>");
300     outFile.println("index</a>.");
301     outFile.println("</p>");
302     outFile.println("</body>");
303     outFile.println("</html>");
304
305     // closing outputs
306     outFile.close();
307
308 }
309
310 /**
311  * Generates list and pulls each word from wordBank queue
and matches with
312  * corresponding definition before printing it. These
expected elements
313  * generated by this method:
314  *
315  * <li>word</li>
316  *
317  * @param wordBank
318  *         the word -> word queue
319  * @param wordDefinition
```



```
320      *           the word and definition -> word and
      definition map
321      * @param out
322      *           the output stream
323      * @param outputFile
324      *           the folder where HTML will be stored
325      * @ensures out.content = #out.content * list of words
326      *
327      */
328      public static void generateWord(Queue<String> wordBank,
329      Map<String, String> wordDefinition, SimpleWriter
      out,
330      String outputFile) {
331
332      String word = wordBank.dequeue();
333      String definition = wordDefinition.value(word);
334      out.println("<li>");
335      generateHTML(wordDefinition, word, definition, out,
      outputFile);
336      out.println("</li>");
337      wordBank.enqueue(word);
338  }
339
340  /**
341   * Outputs the "closing" tags in the generated HTML file.
      These are the
342   * expected elements generated by this method:
343   *
344   * </ul>
345   * </body> </html>
346   *
347   * @param out
348   *           the output stream
349   * @updates out.content
350   * @ensures out.content = #out.content * [the HTML
      "closing" tags]
351   */
352      public static void generateCloser(SimpleWriter out) {
353      out.println("</ul>");
354      out.println("</body>");
355      out.println("</html>");
356  }
357
358  /**
```

```
359     * Main method.
360     *
361     * @param args
362     *         the command line arguments
363     */
364     public static void main(String[] args) {
365         SimpleReader in = new SimpleReader1L();
366         SimpleWriter out = new SimpleWriter1L();
367
368         // prompts user for input and output files
369         out.println("What is the name of the input file?");
370         String newName = in.nextLine();
371
372         out.println("Where do you want to store output
373 files?");
374         String outputFile = in.nextLine();
375
376         SimpleWriter outFile = new SimpleWriter1L(outputFile +
377 "/index.html");
378
379         // initializes wordDefinition map and wordBank queue
380         Map<String, String> wordDefinition = new Map1L<>();
381         Queue<String> wordBank = new Queue1L<>();
382
383         // gets map and queue
384         // sorts queue in alphabetical
385         getMap(newName, wordDefinition);
386         wordQueue(wordDefinition, wordBank);
387
388         // generating HTML files
389         generateIndex(outFile);
390         for (int i = 0; i < wordBank.length(); i++) {
391             generateWord(wordBank, wordDefinition, outFile,
392 outputFile);
393         }
394         generateCloser(outFile);
395
396         // closing outputs
397         in.close();
398         out.close();
399         outFile.close();
400     }
401 }
```

Glossary.java

Saturday, April 16, 2022, 7:14 PM

400