

```
1 import java.util.Comparator;
11
12 /**
13  * Program create a glossary webpage from words and definitions
14  * from an input
15  * file.
16  *
17  * @author Shyam Sai Bethina
18  */
19 public final class Glossary {
20
21     /**
22      * Default constructor--private to prevent instantiation.
23      */
24     private Glossary() {
25
26     }
27
28     /**
29      *
30      * A comparator that orders the queue of words.
31      *
32      */
33
34     private static class StringLT implements Comparator<String> {
35         @Override
36         /*
37          * Compares two strings and returns them in alphabetical
38          * sequence, which
39          * is used to order the words queue later on
40          */
41         public int compare(String one, String two) {
42             return one.compareTo(two);
43         }
44     }
45
46     /**
47      *
48      * Gets the terms from the input file stream and returns a
49      * queue of the
50      * terms.
51      *
52      * @param in
53      *         The input file stream
54      * @return A queue of terms from the input file
55      * @ensures Returned Queue is filled with terms from the input
```

```
file stream
51     *
52     */
53     public static Queue<String> getTerms(SimpleReader in) {
54         //Creates an empty queue to add in terms
55         Queue<String> terms = new Queue1L<>();
56
57         /*
58          * While the input file stream is not at the end, it gets
the next line
59          * within the input, and if the is not empty and or does
not contain a
60          * space within the line, it is a term so it gets added
into the queue
61          */
62         while (!in.atEOS()) {
63             String term = in.nextLine();
64             if (!(term.isEmpty() || term.contains(" "))) {
65                 terms.enqueue(term);
66             }
67
68         }
69
70         return terms;
71     }
72
73     /**
74      * Gets the definitions from the input file stream and returns
a queue of
75      * the definitions.
76      *
77      * @param in
78      *      The input file stream
79      * @return A queue of definitions from the input file
80      * @ensures Returned Queue is filled with definitions from
input file stream
81      */
82     public static Queue<String> getDefinitions(SimpleReader in) {
83         //Creates an empty queue to add in definitions
84         Queue<String> definitions = new Queue1L<>();
85
86         /*
87          * This loops keeps going if the input file stream is not
at the end
```

```
88         */
89         while (!in.atEOS()) {
90             /*
91             * String definition will start out empty, and a test
string of the
92             * next line will be used
93             */
94             String definition = "";
95             String testDefinition = in.nextLine();
96
97             /*
98             * If the test string contains a space, then it is
part of a
99             * definition, and then we set definition to
testDefinition
100            */
101            if (testDefinition.contains(" ")) {
102                definition = testDefinition;
103
104                /*
105                * The while loop stops when the file stream is at
the end or
106                * the next line is empty
107                */
108                while (!testDefinition.equals("")) {
109                    if (!in.atEOS()) {
110                        testDefinition = in.nextLine();
111
112                        /*
113                        * If the next line is not empty and it is
a sentence,
114                        * it gets added to the definition string.
115                        */
116                        if (!testDefinition.isEmpty()
117                            && testDefinition.contains(" ")) {
118                            definition += " " + testDefinition;
119                        } else {
120                            /*
121                            * If the next line is empty or it is
not a
122                            * sentence, then we go out of the
loop
123
124                            */
testDefinition = "";
```

```
125         }
126     } else {
127         /*
128         * If we are at the end of the stream,
129         then we go out of
130         * the loop
131         */
132         testDefinition = "";
133     }
134 }
135 /*
136 * If resulting definition value is not empty, then we
137 add it to the
138 * queue
139 */
140 if (!definition.isEmpty()) {
141     definitions.enqueue(definition);
142 }
143 }
144
145 return definitions;
146
147 }
148
149 /**
150  * Creates an HTML page for each term.
151  *
152  * @param words
153  *         The queue of words from input file
154  * @param definitions
155  *         The queue of definitions from input file
156  * @param folderLocation
157  *         The name of the folder location that user
158  inputed
159  * @requires <pre>
160  * |words| != 0
161  * |definitions| != 0
162  * </pre>
163  * @ensures The HTML page for each term as the term bold-faced
164  and red, has
165  *         the definition with linked terms, and has a link
166  to return to
```

```
164      *           the index page
165      */
166      public static void pageForWords(Queue<String> words,
167      Queue<String> definitions, String folderLocation) {
168          /*
169          * For every word in the words queue, this loop creates a
170          separate HTML
171          * page
172          */
173          for (int i = 0; i < words.length(); i++) {
174              /*
175              * The words and its corresponding definition get
176              dequeues from the
177              * queues. Then a new output file stream gets created
178              and creates a
179              * new HTML file using the words and folder location
180              which is based
181              * on what the user inputted.
182              */
183              String word = words.dequeue();
184              String definition = definitions.dequeue();
185              SimpleWriter out = new SimpleWriter1L(
186                  folderLocation + "/" + word + ".html");
187              out.println("<html>");
188              out.println("    <head>");
189              //The title is the term
190              out.println("        <title>" + word + "</title>");
191              out.println("    </head>");
192              out.println("    <body>");
193              out.println("        <h2>");
194              //The word is bold-faced and gets a red color
195              out.println("            <b><i><font color='red'>" +
196              word
197                  + "</font></i></b>");
198              out.println("        </h2>");
199              /*
200              * The String temp becomes the definition sentence,
201              but each term
202              * within the sentence is linked to it's respective
203              page
204              */
```

```
201         String temp = linkWordsinDefinition(words,
202         definition);
203         out.println("        <blockquote>" + temp + "</
204         <blockquote>");
205         out.println("        <hr>");
206         out.println(
207         "        <p>Return to <a
208         href='index.html'>index</a>.</p>");
209         out.println("    </body>");
210         out.println("</html>");
211         //Closes the output file stream
212         out.close();
213         /*
214         * Enqueues the word and definition to move onto the
215         next words and
216         * definition
217         */
218         words.enqueue(word);
219         definitions.enqueue(definition);
220     }
221 }
222 /**
223  * Outputs the header for the base index HTML file.
224  *
225  * @param out
226  *      The output file stream
227  * @requires out.is_open
228  * @ensures output file has the header for the index HTML file
229  */
230 public static void outputHeader(SimpleWriter out) {
231     /*
232     * Outputs the beginning code of the index HTML file to
233     the output file
234     * stream
235     */
236     out.println("<html>");
237     out.println("    <head>");
238     out.println("        <title>Glossary</title>");
239     out.println("    </head>");
240     out.println("    <body>");
```

```
240         out.println("        <h2>Glossary</h2>");
241         out.println("        <hr />");
242         out.println("        <h3>Index</h3>");
243         out.println("        <ul>");
244     }
245
246     /**
247      * Outputs the footer for the base index HTML file.
248      *
249      * @param out
250      *         The output file stream
251      * @requires out.is_open
252      * @ensures output file has the closing braces for the index
HTML file
253      */
254     public static void outputFooter(SimpleWriter out) {
255         /*
256          * Outputs the closing code of the index HTML file to the
output file
257          * stream
258          */
259         out.println("        </ul>");
260         out.println("    </body>");
261         out.print("</html>");
262     }
263
264     /**
265      * Creates an ordered, bullet-pointed list of the terms, and
the terms are
266      * linked to their respective HTML pages.
267      *
268      * @param out
269      *         The output file stream
270      * @param words
271      *         The queue of words from the input file
272      * @param definitions
273      *         The queue of definitions from the input file
274      * @requires <pre>
275      * out.is_open
276      * |words| != 0
277      * |definitions| != 0
278      * </pre>
279      * @ensures An ordered list of terms in the index HTML file
that is linked
```

```
280      *           to their respective pages
281      */
282      public static void listForWords(SimpleWriter out,
    Queue<String> words,
283      Queue<String> definitions) {
284      /*
285      * Orders the words in the queue by alphabetical order
286      */
287      Comparator<String> order = new StringLT();
288      words.sort(order);
289
290      /*
291      * Each word in the queue gets linked to it's respective
    HTML page
292      */
293      for (int i = 0; i < words.length(); i++) {
294      /*
295      * Each word gets dequeued and enqueued after being
    linked to the
296      * page
297      */
298      String word = words.dequeue();
299      out.println("           <li><a href=" + word +
    ".html>" + word
300      + "</a></li>");
301      words.enqueue(word);
302      }
303      }
304
305      /**
306      * Goes through each word in the definition and determines if
    the word is a
307      * term, if it is, then the word is linked to the term HTML
    page. Then
308      * returns the completed sentence.
309      *
310      * @param words
311      *           The queue of words from the input file
312      * @param definition
313      *           A string representing the definition of a term
314      * @return A string of the definition that links the terms
    within the
315      *           sentence if there are any terms in it
316      * @requires <pre>
```



```

317     * |words| != 0
318     * |definition| != 0
319     * </pre>
320     * @ensures A string of the definition where the terms are
    linked to their
321     *           pages
322     */
323     public static String linkWordsinDefinition(Queue<String>
words,
324         String definition) {
325         /*
326         * Define all possible separator characters
327         */
328         final String separatorStr = " \\t,!.?(){}[];:'";
329         Set<Character> separatorSet = new Set1L<>();
330         generateElements(separatorStr, separatorSet);
331
332         //wordsAndSep is a queue with only words and only
separators
333         Queue<String> wordsAndSep =
nextWordOrSeparator(definition,
334             separatorSet);
335
336         String result = "";
337         for (int i = 0; i < wordsAndSep.length(); i++) {
338             /*
339             * For each element in wordsAndSep, temp becomes a
temporary string
340             * of that element. Then it gets compared to each term
in the words
341             * queue.
342             */
343             String temp = wordsAndSep.dequeue();
344             for (String word : words) {
345                 /*
346                 * If the word does equal a term, then temp
becomes a linked
347                 * word
348                 */
349                 if (temp.equals(word)) {
350                     temp = "<a href=" + word + ".html>" + word +
"</a>";
351                 }
352

```

```
353     }
354     //Restores wordsAndSep
355     wordsAndSep.enqueue(temp);
356
357     /*
358     * Result becomes the previous sentence plus the temp
string and a
359     * whitespace so that the words don't become one whole
word.
360     */
361     result = result + temp;
362
363     }
364
365     return result;
366 }
367
368 /**
369  * Generates the set of characters in the given {@code String}
into the
370  * given {@code Set}.
371  *
372  * @param str
373  *         the given String
374  * @param charSet
375  *         the Set to be replaced
376  * @replaces charSet
377  * @requires <pre>
378  * |str| != 0
379  * |charSet| = 0
380  * </pre>
381  * @ensures charSet = characters of str
382  */
383 public static void generateElements(String str, Set<Character>
charSet) {
384
385     /*
386     * Goes through each character of the string and adds the
non-duplicates
387     * to the set
388     */
389     for (int i = 0; i < str.length(); i++) {
390         char charTemp = str.charAt(i);
391         if (!charSet.contains(charTemp)) {
```

```
392         charSet.add(charTemp);
393     }
394 }
395
396 }
397
398 /**
399  * Returns the first "word" (maximal length string of
400  * characters not in
401  * {@code separators}) or "separator string" (maximal length
402  * string of
403  * characters in {@code separators}) in the given {@code text}
404  * starting at
405  * the given {@code position}.
406  *
407  * @param text
408  *        the {@code String} from which to get the word or
409  *        separator
410  *        string
411  * @param separators
412  *        the {@code Set} of separator characters
413  * @return Queue with only separators and only words
414  * @requires 0 <= position < |text|
415  * @ensures <pre>
416  * The returned Queue will have separators and words, but not
417  * words with separators
418  * </pre>
419  */
420 public static Queue<String> nextWordOrSeparator(String text,
421         Set<Character> separators) {
422     Queue<String> result = new Queue1L<>();
423
424     //Indexes to get the substring of words or separators
425     int firstIndex = 0;
426     int secondIndex = 0;
427     while (firstIndex < text.length()) {
428         String subString;
429         /*
430          * If the character at firstIndex is a separator, then
431          * subString
432          * will equal the string with only separators until
433          * the character is
434          * a letter. If the character at firstIndex is a
435          * letter, then
```

```
428         * subString will equals the string with only letter
    until character
429         * is a separator
430         */
431         if (separators.contains(text.charAt(firstIndex))) {
432             while (secondIndex < text.length()
433                 &&
separators.contains(text.charAt(secondIndex))) {
434                 secondIndex++;
435             }
436         } else {
437             while (secondIndex < text.length()
438                 && !
separators.contains(text.charAt(secondIndex))) {
439                 secondIndex++;
440             }
441         }
442
443         /*
444         * Enqueues the separator or word subString to Queue
    result, and
445         * firstIndex will equal to secondIndex in order reset
    the count
446         */
447         subString = text.substring(firstIndex, secondIndex);
448         result.enqueue(subString);
449         firstIndex = secondIndex;
450     }
451
452     return result;
453 }
454
455 /**
456  * Main method.
457  *
458  * @param args
459  *         the command line arguments; unused here
460  */
461
462     public static void main(String[] args) {
463         /*
464         * Creates input file stream for user input and output
    file stream to
465         * ask questions
```

```
466      */
467      SimpleReader in = new SimpleReader1L();
468      SimpleWriter out = new SimpleWriter1L();
469
470      /*
471      * Gets the desired location and name from user, and
inputName becomes
472      * the answer
473      */
474      out.println("Enter location and name of input file: ");
475      String inputName = in.nextLine();
476
477      /*
478      * This input file stream reads the input file using the
name and
479      * location the user inputed
480      */
481      SimpleReader inputFile = new SimpleReader1L(inputName);
482
483      /*
484      * Asks for the name of the output folder, and folderName
becomes the
485      * answer
486      */
487      out.println("Enter name of output folder: ");
488      String folderName = in.nextLine();
489
490      /*
491      * This output file stream creates a new index HTML file
in the folder
492      * location the user wanted
493      */
494      SimpleWriter outLocation = new SimpleWriter1L(
495          folderName + "/index.html");
496
497      /*
498      * Queue words is filled up with the terms from the input
file, and
499      * inputFile stream is closed because it is not needed
anymore
500      */
501      Queue<String> words = getTerms(inputFile);
502      inputFile.close();
503
```

```
504         //CHECK THIS, do we need a second reader
505         /*
506         * Creates a new input file stream of the same input file
to start at
507         * the beginning of the file, and Queue definitions is
filled up with
508         * definitions from the input file
509         */
510         SimpleReader inputFile2 = new SimpleReader1L(inputName);
511         Queue<String> definitions = getDefinitions(inputFile2);
512
513         /*
514         * Outputs the header for the index HTML file, creates the
pages for the
515         * terms, outputs ordered list of the terms to the index
file, and
516         * outputs the footer for the index HTML file all in the
outLocation
517         * output file stream
518         */
519         outputHeader(outLocation);
520         pageForWords(words, definitions, folderName);
521         listForWords(outLocation, words, definitions);
522         outputFooter(outLocation);
523
524         /*
525         * Closes all the input and output file stream that was
used
526         */
527         in.close();
528         out.close();
529
530         inputFile2.close();
531         outLocation.close();
532
533     }
534
535 }
536
```