

```
1 import java.util.Comparator;
12
13
14 /**
15  * Program to take in an input file of words and outputs an HTML
16  * file of the
17  * words and the amount of times each word appeared in the input
18  * file.
19  *
20  * @author Shyam Sai Bethina
21  */
22 public final class WordCounter {
23
24     /**
25      * Default constructor--private to prevent instantiation.
26      */
27     private WordCounter() {
28     }
29
30     /**
31      *
32      * A comparator that orders the queue of words.
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      * Gets the lines from the input file stream and returns a
48      * queue of the
49      * lines.
50      *
51      * @param in The input file stream
52      * @return A queue of lines from the input file
```

```
52     * @ensures Returned Queue is filled with lines from the input
    file stream
53     *
54     */
55     public static Queue<String> getLines(SimpleReader in) {
56         //Creates an empty queue to add in lines
57         Queue<String> lines = new Queue1L<>();
58
59         /*
60         * While the input file stream is not at the end, it gets
the next line
61         * within the input, and if the is not empty, it gets
added to the Queue
62         */
63         while (!in.atEOS()) {
64             String line = in.nextLine();
65             if (!(line.isEmpty())) {
66                 lines.enqueue(line);
67             }
68         }
69
70
71         return lines;
72     }
73
74     /**
75     * Returns the first "word" (maximal length string of
characters not in
76     * {@code separators}) or "separator string" (maximal length
string of
77     * characters in {@code separators}) in the given {@code text}
starting at
78     * the given {@code position}. Adds only words to the Queue.
79     *
80     * @param text
81     *         the {@code String} from which to get the word or
separator
82     *         string
83     * @param words
84     *         Queue to be replaced with only words from the
text
85     * @requires 0 <= position < |text|
86     * @ensures <pre>
87     * The returned Queue will have words, but not words with
```

```
separators
88      * </pre>
89      */
90      public static void nextWordOrSeparator(Queue<String> words,
String text) {
91
92          /*
93           * Define all possible separator characters
94           */
95          final String separatorStr = " \\t,!.?(){}[];:'-";
96          Set<Character> separatorSet = new Set1L<>();
97          /*
98           * Goes through each character of the string and adds the
non-duplicates
99           * to the set
100          */
101          for (int i = 0; i < separatorStr.length(); i++) {
102              char charTemp = separatorStr.charAt(i);
103              if (!separatorSet.contains(charTemp)) {
104                  separatorSet.add(charTemp);
105              }
106          }
107
108          //Indexes to get the substring of words or separators
109          int firstIndex = 0;
110          int secondIndex = 0;
111          while (firstIndex < text.length()) {
112              String subString;
113              /*
114               * This boolean will be used to determine whether the
string we are
115               * indexing is a word or separators
116               */
117              boolean word = false;
118
119              /*
120               * If the character at firstIndex is a separator, then
subString
121               * will equal the string with only separators until
the character is
122               * a letter. If the character at firstIndex is a
letter, then
123               * subString will equals the string with only letter
until character
```

```
124         * is a separator
125         */
126         if (separatorSet.contains(text.charAt(firstIndex))) {
127             while (secondIndex < text.length()
128                 &&
129                 separatorSet.contains(text.charAt(secondIndex))) {
130                 secondIndex++;
131             }
132         } else {
133             while (secondIndex < text.length()
134                 && !
135                 separatorSet.contains(text.charAt(secondIndex))) {
136                 secondIndex++;
137             }
138             /*
139             * Since the characters in this block don't belong
140             * to the
141             * separate, they belong to words, which will make
142             * the word
143             * boolean true
144             */
145             word = true;
146         }
147
148         /*
149         * If the resulting subString is a word, this block
150         * enqueues the
151         * word to Queue words, and firstIndex will equal to
152         * secondIndex in
153         * order reset the count
154         */
155         if (word) {
156             subString = text.substring(firstIndex,
157                 secondIndex);
158             words.enqueue(subString.toLowerCase());
159         }
160
161         firstIndex = secondIndex;
162     }
163 }
164
165 /**
166  *
```

```
161     * @param counts
162     *           The map to add the words as the keys and the
    number of times
163     *           the words appear as the value
164     * @param lines
165     *           The queue of lines from the input file
166     * @ensures <pre>
167     * The map will have all the words from the input file and the
    occurrences
168     * of each word in the file as the value
169     * </pre>
170     */
171     public static void addToMap(Map<String, Integer> counts,
172                                Queue<String> lines) {
173         /*
174         * Creates a queue and adds in all the words from all the
    lines from the
175         * input file
176         */
177         Queue<String> words = new Queue1L<String>();
178         for (String line : lines) {
179             nextWordOrSeparator(words, line);
180         }
181
182         /*
183         * Goes through each words to check if it is in the map
    already. If it
184         * is, then it updates the count value of the word. If it
    is not in the
185         * map, then it adds it into the map with a value of 1
186         */
187         for (String word : words) {
188             if (!counts.containsKey(word)) {
189                 counts.add(word, 1);
190             } else {
191                 int temp = counts.value(word);
192                 temp++;
193                 counts.replaceValue(word, temp);
194             }
195         }
196     }
197 }
198
199 /**
```

```
200     * Outputs the header for the index HTML file.
201     *
202     * @param out
203     *         The output file stream
204     * @param fileName
205     *         The name of the file the user desired
206     * @requires out.is_open
207     * @ensures output file has the header for the index HTML file
208     */
209     public static void outputHeader(SimpleWriter out, String
fileFileName) {
210         /*
211         * Outputs the beginning code of the index HTML file to
the output file
212         * stream
213         */
214         out.println("<html>");
215         out.println("    <head>");
216         out.println("        <title>Words Counted in " + fileFileName +
"</title>");
217         out.println("    </head>");
218         out.println("    <body>");
219         out.println("        <h2>Words Counted in " + fileFileName +
"</h2>");
220         out.println("        <hr/>");
221         out.println("        <table border='1'>");
222         out.println("            <tbody>");
223         out.println("                <tr><th>Words</th><th>Counts</
th></tr>");
224     }
225 }
226
227 /**
228     * Outputs the footer for the index HTML file.
229     *
230     * @param out
231     *         The output file stream
232     * @requires out.is_open
233     * @ensures output file has the closing braces for the index
HTML file
234     */
235     public static void outputFooter(SimpleWriter out) {
236         /*
237         * Outputs the closing code of the index HTML file to the
```

```
    output file
238        * stream
239        */
240        out.println("                </tbody>");
241        out.println("                </table>");
242        out.println("    </body>");
243        out.print("</html>");
244    }
245
246    /**
247     * Outputs the words and corresponding counts to the table in
the index HTML
248     * file.
249     *
250     * @param counts
251     *         The map of the words and their corresponding
occurrences in
252     *         the input file.
253     * @param out
254     *         The output file stream
255     * @requires out.is_open
256     * @ensures output file has the code to output the table of
words and counts
257     *         in the HTML file.
258     */
259    public static void outputCounts(Map<String, Integer> counts,
260        SimpleWriter out) {
261        /*
262         * This queue has all the words within in the map
263         */
264        Queue<String> words = new Queue1L<String>();
265        for (Map.Pair<String, Integer> pair : counts) {
266            words.enqueue(pair.key());
267        }
268
269        /*
270         * The queue gets sorted in alphabetical order/
271         */
272        Comparator<String> order = new StringLT();
273        words.sort(order);
274
275        /*
276         * We go through the queue, and output the word and the
corresponding
```

```
277         * count of the word. Since the queue is in order, the
        list will appear
278         * in alphabetical order on the HTML file
279         */
280         for (String word : words) {
281             int count = counts.value(word);
282             out.println("                <tr><th>" + word + "</
th><th>" + count
283                         + "</th></tr>");
284         }
285     }
286
287     /**
288     * Main method.
289     *
290     * @param args
291     *         the command line arguments; unused here
292     */
293     public static void main(String[] args) {
294         /*
295         * Creates input file stream for user input and output
        file stream to
296         * ask questions
297         */
298         SimpleReader in = new SimpleReader1L();
299         SimpleWriter out = new SimpleWriter1L();
300
301         /*
302         * Gets the input file name from user, and inputName
        becomes the answer
303         */
304         out.println("Enter location and name of input file: ");
305         String inputName = in.nextLine();
306
307         /*
308         * This input file stream reads the input file using the
        name the user
309         * inputed
310         */
311         SimpleReader inputFile = new SimpleReader1L(inputName);
312
313         /*
314         * Asks for the name of the output file name, and fileName
        becomes the
```



```
315         * answer
316         */
317         out.println("Enter name of output file: ");
318         String fileName = in.nextLine();
319
320         /*
321         * This output file stream creates a new file with the
name the user
322         * wanted
323         */
324         SimpleWriter outFile = new SimpleWriter1L(fileName);
325
326         /*
327         * Queue lines is filled up with the lines from the input
file, and
328         * inputFile stream is closed because it is not needed
anymore
329         */
330         Queue<String> lines = getLines(inputFile);
331
332         /*
333         * The counts map has all the words and counts of each
word
334         */
335         Map<String, Integer> counts = new Map1L<String,
Integer>();
336         addToMap(counts, lines);
337
338         /*
339         * The next three lines outputs the header, the list, and
the footer of
340         * the HTML file to the desire file
341         */
342         outputHeader(outFile, inputName);
343         outputCounts(counts, outFile);
344         outputFooter(outFile);
345
346         /*
347         * Closes all the input and output streams
348         */
349         inputFile.close();
350         in.close();
351         out.close();
352         outFile.close();
```

WordCounter.java

Thursday, January 20, 2022, 4:10 PM

```
353     }  
354  
355 }  
356
```