

```

1 import java.io.BufferedReader;
17
18 /**
19  * Read txt and find the amount words that appear most frequently. The amount is
20  * an integer entered by the user.
21  *
22  * @author Qinuo Shi & Yiming Cheng
23  */
24 public final class TagCloudStandard {
25
26     /**
27      * Private constructor so this utility class cannot be instantiated.
28      */
29     private TagCloudStandard() {
30     }
31
32     /**
33      * there separator are used in countingWords method.
34      */
35     public static final String Separator = "\\ \\t\\n\\r,-.!?[]';:/()@&~`\"";
36
37     /**
38      * countingWords is for inserting the separated word and its number of uses
39      * into map.
40      *
41      * @param contentIn
42      *         BufferedReader
43      * @param wordNum
44      *         the map include all words and their occurrences
45      * @requires BufferedReader should not be empty
46      * @update wordwordNum
47      * @ensures wordNum should not be empty
48      *
49      */
50     public static void countingWords(BufferedReader contentIn,
51                                     Map<String, Integer> wordNum) {
52         assert contentIn != null : "Violation of: txt file is not null";
53
54         /**
55          * Store all possible non-alphabetic symbols in a set.
56          */
57         Set<Character> sepSet = new HashSet<Character>();
58         for (int i = 0; i < Separator.length(); i++) {
59             sepSet.add(Separator.charAt(i));
60         }
61
62         /**
63          * Read content in the file and write them into map.
64          */
65         String contentLine;
66         try {
67             contentLine = contentIn.readLine();
68             while (contentLine != null) {
69                 int pos = 0;
70                 while (pos < contentLine.length()) {
71                     String word = nextWordOrSeparator(contentLine, pos, sepSet);
72                     if (!sepSet.contains(word.charAt(0))) {
73                         if (!(wordNum.containsKey(word))) {
74                             wordNum.put(word, 1);
75                         } else {
76                             /**
77                              * If a word is already recorded, change the map

```

```

78         * value, and leaving the set unchanged.
79         */
80         int num = wordNum.get(word) + 1;
81         wordNum.put(word, num);
82     }
83     }
84     pos += word.length();
85 }
86 contentLine = contentIn.readLine();
87 }
88 } catch (IOException e1) {
89     e1.printStackTrace();
90     System.out.print("There are some errors with recording content");
91     return;
92 }
93 }
94
95 /**
96  * Returns the first "word" (maximal length string of characters not in
97  * {@code separators}) or "separator string" (maximal length string of
98  * characters in {@code separators}) in the given {@code text} starting at
99  * the given {@code position}.
100  *
101  * @param str
102  *      the {@code String} from which to get the word or separator
103  *      string
104  * @param pos
105  *      the starting index
106  * @param sepSet
107  *      the {@code Set} of separator characters
108  * @return the first word or separator string found in {@code str} starting
109  *      at index {@code pos}
110  * @requires 0 <= pos < |str|
111  * @ensures <pre>
112  * nextWordOrSeparator =
113  *   str[pos, pos + |nextWordOrSeparator|) and
114  *   if entries(str[pos, pos + 1)) intersection separators = {}
115  * then
116  *   entries(nextWordOrSeparator) intersection separators = {} and
117  *   (pos + |nextWordOrSeparator| = |str| or
118  *   entries(str[pos, pos + |nextWordOrSeparator| + 1))
119  *   intersection separators != {})
120  * else
121  *   entries(nextWordOrSeparator) is subset of separators and
122  *   (pos + |nextWordOrSeparator| = |str| or
123  *   entries(str[pos, pos + |nextWordOrSeparator| + 1))
124  *   is not subset of separators)
125  * </pre>
126  */
127 public static String nextWordOrSeparator(String str, int pos,
128     Set<Character> sepSet) {
129     assert str != null : "Violation of: str is not null";
130     assert sepSet != null : "Violation of: separators is not null";
131     assert 0 <= pos : "Violation of: 0 <= pos";
132     assert pos < str.length() : "Violation of: pos < |str|";
133
134     int endPos = -1;
135     String word = "";
136     /*
137      * Use for loop to find the term's position.
138      */
139     for (int i = pos; i < str.length(); i++) {

```

```

140         if (sepSet.contains(str.charAt(i)) && endPos == -1) {
141             endPos = i;
142         }
143     }
144     /*
145     * Depending on the case, intercepts the corresponding substring.
146     */
147     if (endPos == pos) {
148         word = str.substring(pos, endPos + 1);
149     } else if (endPos == -1) {
150         word = str.substring(pos);
151     } else {
152         word = str.substring(pos, endPos);
153     }
154
155     return word;
156 }
157
158 /**
159  * compare two integers and return a value.
160  */
161 private static class CompareNum
162     implements Comparator<Map.Entry<String, Integer>> {
163     @Override
164     public int compare(Entry<String, Integer> p1,
165         Entry<String, Integer> p2) {
166         int cmp = p2.getValue().compareTo(p1.getValue());
167         if (cmp != 0) {
168             return cmp;
169         }
170         return p2.getValue().compareTo(p1.getValue());
171     }
172 }
173
174
175 /**
176  * compare two strings and return a value.
177  */
178 private static class CompareString implements Comparator<String> {
179
180     @Override
181     public int compare(String p1, String p2) {
182         int cmp = p1.toLowerCase().compareTo(p2.toLowerCase());
183         if (cmp != 0) {
184             return cmp;
185         }
186
187         return p2.compareTo(p1);
188     }
189 }
190
191
192 /**
193  * Sorting the {@code sortByStr} with top {@code n} counts in the
194  * {@code sortByInt} and returns the String that contains the maximum and
195  * mininum number of count in {@code sortByStr} separated by character ';'.
196  *
197  * @param wordNum
198  *         the map include all words and their occurrences
199  * @param orderedInt
200  *         a PriorityQueue to sort all int value
201  * @param orderedString

```

```

202     *           a SortedMap for recording all ordered things for html output
203     *           in other method
204     * @param number
205     *           the amount of the most frequent words
206     * @return a String that record the maximum and the minimum occurrences
207     * @requires the sortingMap and the number should not be empty
208     * @ensures MaxMin need record "maximum value ; minimum value"
209     */
210     private static String sortingMap(Map<String, Integer> wordNum,
211                                     PriorityQueue<Map.Entry<String, Integer>> orderedInt,
212                                     SortedMap<String, Integer> orderedString, int number) {
213         assert wordNum != null : "Violation of: the map is not null";
214         assert number > 0 : "Violation of: Number must be positive";
215
216         String maxAndMin = "";
217
218         /*
219          * Put each pair into a PriorityQueue, then the int value would be
220          * ordered.
221          */
222         for (Map.Entry<String, Integer> pairToPQueue : wordNum.entrySet()) {
223             orderedInt.add(pairToPQueue);
224         }
225
226         for (int i = 0; i < number; i++) {
227             Map.Entry<String, Integer> pair = orderedInt.remove();
228             int occurrence = pair.getValue();
229             String word = pair.getKey();
230
231             /*
232              * After ordering, the first one is the most frequent one, and last
233              * one is the least one.
234              */
235             if (i == 0) {
236                 maxAndMin = pair.getValue().toString() + ";";
237             }
238             if (i == number - 1) {
239                 maxAndMin += pair.getValue().toString();
240             }
241
242             /*
243              * Put them together again for output html file.
244              */
245             orderedString.put(word, occurrence);
246         }
247
248         return maxAndMin;
249     }
250
251     /**
252     * read the txt file and enter all the words into set and map.
253     *
254     * @param orderedString
255     *           its contents are used to identify the most frequently used
256     *           words, as well as the number of occurrences.
257     * @param out
258     *           the output
259     * @param htmlName
260     *           the name of output file
261     * @param maxMin
262     *           a String that record the maximum and the minimum occurrences
263     * @requires orderedString should not be empty, htmlName should not be

```

```

264      *          empty, maxMin should not be empty
265      */
266      public static void creatHtml(SortedMap<String, Integer> orderedString,
267          PrintWriter out, String htmlName, String maxMin) {
268          assert orderedString != null : "Violation of: the orderedString is not null";
269          assert htmlName != null : "Violation of: the htmlName is not null";
270          assert maxMin != null : "Violation of: the maxMin is not null";
271
272          /*
273           * Output front part of html.
274           */
275          out.println("<html>");
276          out.println("  <head>");
277          out.println("    <title> " + "Top " + orderedString.size() + " words in "
278              + htmlName + "</title>");
279          out.println(
280              "    <link href=\"http://web.cse.ohio-state.edu/software/2231/web-
281              sw2/assignments/projects/tag-cloud-generator/data/tagcloud.css\" rel =\"stylesheet\" type=
282              \"text/css\"");
283          out.println("  </head>");
284          out.println("  <body>");
285          out.println("    <h2> Top " + orderedString.size() + " words in "
286              + htmlName + "</h2>");
287          out.println("    <hr>");
288          out.println("    <div class = \"cdiv\">");
289          out.println("      <p class=\"cbox\">");
290
291          /*
292           * Subtract the maximum and minimum value from maxMin.
293           */
294          int symbolpos = maxMin.indexOf(';');
295          int max = Integer.parseInt(maxMin.substring(0, symbolpos));
296          int min = Integer.parseInt(maxMin.substring(symbolpos + 1));
297
298          /*
299           * Output each line of middle part of html.
300           */
301          final int maxTypeSize = 48;
302          final int minTypeSize = 11;
303          for (Map.Entry<String, Integer> wordList : orderedString.entrySet()) {
304              String word = wordList.getKey();
305              int numOfWorks = orderedString.get(word);
306              int typeSize;
307
308              /*
309               * Use a formula to calculate the type size of each word.
310               */
311              if (max - min != 0) {
312                  typeSize = ((maxTypeSize - minTypeSize) * (numOfWorks - min)
313                      / (max - min)) + minTypeSize;
314              } else {
315                  typeSize = maxTypeSize;
316              }
317              out.println("      <span style=\"cursor:default\" class=\"\" + "f"
318                  + typeSize + "\" title=\"count: " + numOfWorks + "\">"
319                  + word + "</span>");
320          }
321
322          /*
323           * Output last part of html.
324           */
325          out.println("    </p>");

```

```
324         out.println("    </div>");
325         out.println(" </body>");
326         out.println("</html>");
327
328     }
329
330     /**
331     * Main method.
332     *
333     * @param args
334     *         the command line arguments
335     */
336     public static void main(String[] args) {
337
338         BufferedReader in = new BufferedReader(
339             new InputStreamReader(System.in));
340
341         /*
342         * Ask the users to enter the txt file name they want to check.
343         */
344         String fileName;
345         try {
346             System.out.println("Enter a txt file name: ");
347             fileName = in.readLine();
348         } catch (IOException e) {
349             System.err
350                 .println("There are some errors with entering file name..");
351             return;
352         }
353
354         /*
355         * Ask the users to enter the html file name they want to write in.
356         */
357         String htmlName;
358         try {
359             System.out.println("Enter a html file name: ");
360             htmlName = in.readLine();
361         } catch (IOException e) {
362             System.err
363                 .println("There are some errors with entering html name..");
364             return;
365         }
366
367         /*
368         * Ask the users to enter a number for the amount of the most frequent
369         * words they want to check.
370         */
371         int num;
372         try {
373             System.out.println(
374                 "Enter a positive number for the count of the most frequent words: ");
375             num = Integer.parseInt(in.readLine());
376             while (!(num > 0)) {
377                 System.out.println(num
378                     + " is not a positive number, enter another number: ");
379                 num = Integer.parseInt(in.readLine());
380             }
381         } catch (IOException e) {
382             System.err.println(
383                 "There are some errors with enter the number for the count of the most
384                 frequent words.");
385             return;
386         }
387     }
388 }
```

```
385     }
386
387     /*
388     * Build a BufferedReader type to read the txt file.
389     */
390     BufferedReader fileContent;
391     try {
392         fileContent = new BufferedReader(new FileReader(fileName));
393     } catch (IOException e) {
394         System.err.println(
395             "There are some errors with reading the txt file.");
396         return;
397     }
398
399     /*
400     * Build a map which will store the contents in txt files.
401     */
402     Map<String, Integer> wordNum = new HashMap<String, Integer>();
403     countingWords(fileContent, wordNum);
404
405     /*
406     * If users' value is larger the amount of all word, just arrange all the
407     * words in the txt file.
408     */
409     if (num > wordNum.size()) {
410         num = wordNum.size();
411     }
412
413     /*
414     * Build a PrintWriter type to write html file.
415     */
416     PrintWriter htmlContent;
417     try {
418         htmlContent = new PrintWriter(
419             new BufferedWriter(new FileWriter(htmlName)));
420     } catch (IOException e) {
421         System.err.println(
422             "There are some errors with writing the html file.");
423         return;
424     }
425
426     /*
427     * Build a map to store the contents of the file. Build a comparator to
428     * arrange. Build a PriorityQueue to order int value.
429     */
430     Comparator<String> orderString = new CompareString();
431     Comparator<Map.Entry<String, Integer>> orderInt = new CompareNum();
432     PriorityQueue<Map.Entry<String, Integer>> orderedInt = new
PriorityQueue<Map.Entry<String, Integer>>(
433         orderInt);
434     SortedMap<String, Integer> orderedString = new TreeMap<String, Integer>(
435         orderString);
436
437     /*
438     * Return a String that record the maximum and the minimum occurrences
439     * int the map which called wordWithOccurrence.
440     */
441     String maxMin = sortingMap(wordNum, orderedInt, orderedString, num);
442
443     /*
444     * Output the content in html format to a html file.
445     */
```

```
446         creatHtml(orderedString, htmlContent, htmlName, maxMin);
447
448         /*
449         * Close all things.
450         */
451         try {
452             in.close();
453             fileContent.close();
454             htmlContent.close();
455         } catch (IOException e) {
456             System.err.println("There are some errors with closing things");
457         }
458     }
459
460 }
461
```