```
1  import java.io.*;
2  import java.util.*;
3  import java.util.concurrent.LinkedBlockingQueue;
4  import java.util.regex.Pattern;
5
6  public class YAAM_test {
7
8      private static List<String> readConfigLines(String filename) throws IOExcept
   ion {
9          FileReader fileReader = new FileReader(filename);
10         BufferedReader bufferedReader = new BufferedReader(fileReader);
11         List<String> lines = new LinkedList<>();
12         String line = null;
13         while ((line = bufferedReader.readLine()) ≠ null) {
14             if (¬line.startsWith("#")) {
15                 lines.add(line);
16             }
17         }
18         bufferedReader.close();
19         return lines;
20     }
21
22     public static void main(String[] args) throws InterruptedException, IOExcept
   ion {
23
24         // ---------------------- INIT INPUT AND VARIABLES FROM CONFIG ---------
   -------------
25
26         Logger.init("YAAM_log.txt");
27
28         Scanner sc = new Scanner(System.in);
29         /*File apacheLogs = new File("test_log.txt");
30         Scanner sc = new Scanner(apacheLogs);*/
31
32         String config_general = readConfigLines("config_general").get(0);
33         Logger.currentLogLevel = Logger.intToLogLevel(Integer.parseInt(config_ge
   neral.split(",")[0]));
34         int PARSER_POOL_SIZE = Integer.parseInt(config_general.split(",")[1]);
35
36         List<String> config_statistics = readConfigLines("config_statistics");
37
38         List<String> config_statistics_viewer = readConfigLines("config_statistics_view
   er");
39         String config_statistics_viewer_line1 = config_statistics_viewer.remove(
   0);
40         int statisticsViewWaitMilliseconds = Integer.parseInt(config_statistics_
   viewer_line1.split(",")[0]);
41
42         List<String> config_loggers = readConfigLines("config_loggers");
43
44         List<String> config_rankings = readConfigLines("config_rankings");
45         String config_rankings_line1 = config_rankings.remove(0);
46         int rankingTempFileMaxLines = Integer.parseInt(config_rankings_line1.spl
   it(",")[0]);
47         int rankingTempNumThreads = Integer.parseInt(config_rankings_line1.split
   (",")[1]);
48         int rankingMergerSleepMilliseconds = Integer.parseInt(config_rankings_li
   ne1.split(",")[2]);
49         int maxErrorsToShow = Integer.parseInt(config_rankings_line1.split(",")[
   3]);
50
51         // ---------------------- LOAD STATISTICS AND START STATISTICS UPDATER T
   HREADS ----------------------
```

```
52
53          List<String> statisticsNames = new LinkedList<>();
54          List<Pattern> statisticsRegex = new LinkedList<>();
55          HashMap<String, Statistic> statistics = new HashMap<>();
56          List<LinkedBlockingQueue> statisticUpdatersQueues = new LinkedList<>();
57          List<StatisticUpdater> statisticUpdaters = new LinkedList<>();
58          List<Thread> statisticUpdaterThreads = new LinkedList<>();
59
60          /*String[] config_statistics = {"requests,^(.+ .+ \\[.+\\] \".+ .+ .+\"
    [0-9]{3} .+)$",
61                  "clients,^(.+) .+ \\[.+\\] \".+ .+ .+\" [0-9]{3} .+$",
62                  "errors,^.+ .+ \\[(error)\\] \".+ .+ .+\" [0-9]{3} .+$",
63                  "resources,^.+ .+ \\[.+\\] \".+ (.+) .+\" [0-9]{3} .+$"};*/
64
65
66          for (int i = 0; i < config_statistics.size(); ++i) {
67              String[] splitLine = config_statistics.get(i).split(",");
68              statisticsNames.add(splitLine[0]);
69              statisticsRegex.add(Pattern.compile(splitLine[1]));
70
71              statistics.put(statisticsNames.get(i), new Statistic(statisticsNames
    .get(i)));
72              statisticUpdatersQueues.add(new LinkedBlockingQueue());
73              statisticUpdaters.add(new StatisticUpdater(statisticUpdatersQueues.g
    et(i),
74                      statistics.get(statisticsNames.get(i))));
75              statisticUpdaterThreads.add(new Thread(statisticUpdaters.get(i)));
76              statisticUpdaterThreads.get(i).start();
77          }
78
79          // ---------------------- START STATISTICS VIEWER THREAD ---------------
    -------
80
81          StatisticViewer statisticViewer = new StatisticViewer(statistics, statis
    ticsViewWaitMilliseconds);
82          Thread statisticsViewerThread = new Thread(statisticViewer);
83          statisticsViewerThread.start();
84
85          // ---------------------- START LOGGING THREADS ----------------------
86
87          List<String> loggerNames = new LinkedList<>();
88          List<Pattern> loggerRegex = new LinkedList<>();
89          List<LinkedBlockingQueue> fileLoggersQueues = new LinkedList<>();
90          List<FileLogger> fileLoggers = new LinkedList<>();
91          List<Thread> fileLoggersThreads = new LinkedList<>();
92
93          /*String[] config_loggers = {"full_log,^(.+ .+ \\[.+\\] \".+ .+ .+\" [0-
    9]{3} .+)$",
94                                     "error_log,^.+ (.+) \\[error\\] \".+ .+ .+\"
     [0-9]{3} (.+)$"};*/
95
96          for (int i = 0; i < config_loggers.size(); ++i) {
97              String[] splitLine = config_loggers.get(i).split(",");
98
99              loggerNames.add(splitLine[0]);
100             loggerRegex.add(Pattern.compile(splitLine[1]));
101             fileLoggersQueues.add(new LinkedBlockingQueue());
102             fileLoggers.add(new FileLogger(loggerNames.get(i), fileLoggersQueues
    .get(i)));
103             fileLoggersThreads.add(new Thread(fileLoggers.get(i)));
104             fileLoggersThreads.get(i).start();
105         }
106
```

```
107         // ---------------------- START RANKING THREADS ---------------------
108
109         List<String> rankingNames = new LinkedList<>();
110         List<Pattern> rankingRegex = new LinkedList<>();
111         List<LinkedBlockingQueue> rankingQueues = new LinkedList<>();
112         List<List<RankingLogger>> rankingLoggers = new LinkedList<>();
113         List<List<Thread>> rankingThreads = new LinkedList<>();
114         List<String> finishedLogFiles = new LinkedList<>();
115
116         /*String[] config_rankings = {"errors_ranking,^.+ .+ \\[error\\] \".+ .+
     .+\" [0-9]{3} (.+)$"};*/
117
118         for (int i = 0; i < config_rankings.size(); ++i) {
119             String[] splitLine = config_rankings.get(i).split(",");
120
121             rankingNames.add(splitLine[0]);
122             rankingRegex.add(Pattern.compile(splitLine[1]));
123             rankingQueues.add(new LinkedBlockingQueue());
124             rankingLoggers.add(new LinkedList<>());
125             rankingThreads.add(new LinkedList<>());
126             for (int j = 0; j < rankingTempNumThreads; ++j) {
127                 RankingLogger rankingLogger = new RankingLogger(rankingNames.get
    (i), rankingQueues.get(i),
128                         finishedLogFiles, rankingTempFileMaxLines);
129                 rankingLoggers.get(i).add(rankingLogger);
130                 Thread rankingLoggerThread = new Thread(rankingLogger);
131                 rankingThreads.get(i).add(rankingLoggerThread);
132                 rankingLoggerThread.start();
133             }
134         }
135
136         // ---------------------- START RANKING MERGE THREADS ----------------
    ----
137
138         List<RankingLoggerMerge> rankingMergers = new LinkedList<>();
139         List<Thread> rankingMergerThreads = new LinkedList<>();
140
141         for (int i = 0; i < config_rankings.size(); ++i) {
142             String[] splitLine = config_rankings.get(i).split(",");
143
144             rankingMergers.add(new RankingLoggerMerge(splitLine[0], finishedLogF
    iles,
145                     rankingMergerSleepMilliseconds, maxErrorsToShow));
146             rankingMergerThreads.add(new Thread(rankingMergers.get(i)));
147             rankingMergerThreads.get(i).start();
148         }
149
150         // ---------------------- START PARSER THREAD POOL -------------------
    -
151
152         LinkedBlockingQueue analyzerPoolQueue = new LinkedBlockingQueue();
153
154         Parser[] analyzers = new Parser[PARSER_POOL_SIZE];
155         Thread[] analyzerThreads = new Thread[PARSER_POOL_SIZE];
156
157         List<LinkedBlockingQueue> allQueues = new LinkedList<>();
158         allQueues.addAll(statisticUpdatersQueues);
159         allQueues.addAll(fileLoggersQueues);
160         allQueues.addAll(rankingQueues);
161         List<Pattern> allRegex = new LinkedList<>();
162         allRegex.addAll(statisticsRegex);
163         allRegex.addAll(loggerRegex);
164         allRegex.addAll(rankingRegex);
```

```
165
166         for (int i = 0; i < PARSER_POOL_SIZE; ++i) {
167             analyzers[i] = new Parser(analyzerPoolQueue, allRegex, allQueues);
168             analyzerThreads[i] = new Thread(analyzers[i]);
169             analyzerThreads[i].start();
170         }
171
172
173         // ---------------------- SHUTDOWN HOOK ----------------------
174
175         Runtime.getRuntime ().addShutdownHook ( new Thread () {
176             @Override
177             public void run () {
178                 System.out.println ( "Shutdown hook" );
179                 Logger.log("main", "Closing everything", Logger.logLevel.INFO);
180
181                 try {
182                     // ---------------------- CLOSE STATISTIC UPADTER THREADS --
    --------------------
183
184                     for (int i = 0; i < statisticUpdaterThreads.size(); ++i) {
185                         statisticUpdaters.get(i).stopKeepAlive();
186                         statisticUpdaterThreads.get(i).interrupt();
187                         statisticUpdaterThreads.get(i).join();
188                     }
189
190                     // ---------------------- CLOSE STATISTIC VIEWER THREAD ----
    ------------------
191
192                     statisticViewer.stopKeepAlive();
193                     statisticsViewerThread.interrupt();
194                     statisticsViewerThread.join();
195
196                     // ---------------------- CLOSE LOGGER THREADS -------------
    ---------
197
198                     for (int i = 0; i < fileLoggersThreads.size(); ++i) {
199                         fileLoggers.get(i).stopKeepAlive();
200                         fileLoggersThreads.get(i).interrupt();
201                         fileLoggersThreads.get(i).join();
202                     }
203
204                     // ---------------------- CLOSE RANKING THREADS ------------
    ----------
205
206                     for (int i = 0; i < rankingThreads.size(); ++i) {
207                         List<RankingLogger> oneRankLoggers = rankingLoggers.get(
    i);
208                         List<Thread> oneRankLoggerThreads = rankingThreads.get(i
    );
209                         for (int j = 0; j < oneRankLoggers.size(); ++j) {
210                             oneRankLoggers.get(j).stopKeepAlive();
211                             oneRankLoggerThreads.get(j).interrupt();
212                             oneRankLoggerThreads.get(j).join();
213                         }
214                     }
215
216                     // ---------------------- CLOSE RANKING MERGER THREADS -----
    ------------------
217
218                     for (int i = 0; i < rankingMergerThreads.size(); ++i) {
219                         rankingMergers.get(i).stopKeepAlive();
220                         rankingMergerThreads.get(i).interrupt();
```

```
221                         rankingMergerThreads.get(i).join();
222                     }
223 
224                     // ---------------------- CLOSE PARSER THREADS -------------
    ---------
225 
226                     for (int i = 0; i < PARSER_POOL_SIZE; ++i) {
227                         analyzers[i].stopKeepAlive();
228                         analyzerThreads[i].interrupt();
229                         analyzerThreads[i].join();
230                     }
231 
232                     Logger.log("main", "All done", Logger.logLevel.INFO);
233                 } catch (InterruptedException e) {
234                     Logger.log("main", "Shutdown hook interrupted", Logger.logLevel.INF
    O);
235                 }
236 
237                 Logger.close();
238             }
239         } );
240 
241         // ---------------------- MAIN LOOP ----------------------
242 
243         boolean endSignal = false;
244         while(¬endSignal ∧ sc.hasNextLine()) {
245             String logLine = sc.nextLine();
246             Logger.log("main", "Recibi de la cola: " + logLine, Logger.logLevel.INFO);
247 
248             if (logLine.equals("end")) {
249                 endSignal = true;
250             } else {
251                 analyzerPoolQueue.put(logLine);
252             }
253         }
254     }
255 
256 }
```

```
1  import java.util.*;
2
3  public class StatisticViewer extends GracefulRunnable {
4
5      private Map<String, Statistic> statistics;
6      private int sleepMilliseconds;
7
8      public StatisticViewer(Map<String, Statistic> statistics, int sleepMilliseco
   nds) {
9          super("StatisticsViewer");
10
11         this.statistics = statistics;
12         this.sleepMilliseconds = sleepMilliseconds;
13     }
14
15     @Override
16     public void doWork() {
17
18         try {
19             Thread.sleep(sleepMilliseconds);
20
21             Logger.output("\n");
22
23             // requests por segundo
24             int totalRequests = 0;
25             Map<String, Integer> requestStatistics = statistics.get("requests").ge
   tStatistic();
26             for (String key : requestStatistics.keySet()) {
27                 totalRequests += requestStatistics.getOrDefault(key, 0);
28             }
29             float requestsPerSecond = (float)totalRequests / (sleepMilliseconds
   / 1000);
30             Logger.output("[VIEWER] requests per second: " + requestsPerSecond);
31
32             // requests por cliente
33             int totalDistinctClients = statistics.get("clients").getStatistic().ke
   ySet().size();
34             float requestsPerClient = totalDistinctClients > 0 ? (float)totalReq
   uests / totalDistinctClients : 0;
35             Logger.output("[VIEWER] requests per client: " + requestsPerClient);
36
37             // cantidad de errores
38             int totalErrors = statistics.get("errors").getStatistic().getOrDefault
   ("error", 0);
39             Logger.output("[VIEWER] total errors: " + totalErrors);
40
41             // 10 recursos mas pedidos
42             LimitedSortedSet<String> topResources = new LimitedSortedSet<>(10, n
   ew CountCommaNameComparator());
43             Map<String, Integer> resources = statistics.get("resources").getStatis
   tic();
44             for (String key : resources.keySet()) {
45                 topResources.add(resources.get(key) + "," + key);
46             }
47             Logger.output("[VIEWER] 10 most requested resources: ");
48             for (String resource : topResources) {
49                 Logger.output("\t" + resource);
50             }
51
52             Logger.output("\n");
53
54         } catch (InterruptedException e) {
55             Logger.log(logName, "I was interrupted", Logger.logLevel.INFO);
```

```
56                  }
57              }
58      }
```

```java
import java.util.concurrent.LinkedBlockingQueue;

public class StatisticUpdater extends GracefulRunnable {

    private LinkedBlockingQueue inputQueue;
    private Statistic myStatistic;

    public StatisticUpdater(LinkedBlockingQueue queue, Statistic stat) {
        super("StatisticUpdater " + stat.name);

        this.inputQueue = queue;
        this.myStatistic = stat;
    }

    @Override
    protected void doWork() {

        Logger.log(logName, "Waiting for input", Logger.logLevel.INFO);

        try {
            String value = inputQueue.take().toString();
            Logger.log(logName, "Recibi de la cola: " + value, Logger.logLevel.INFO);

            myStatistic.updateStatistic(value);

        } catch (InterruptedException e) {
            Logger.log(logName, "I was interrupted", Logger.logLevel.INFO);
        }
    }
}
```

```java
import java.util.HashMap;
import java.util.Map;
import java.util.regex.Pattern;

public class Statistic {

    //  ---------------------- CLASS VARIABLES ----------------------

    public String name;
    private Map<String, Integer> statisticValues = new HashMap<>();
    private Object statisticLock = new Object();

    public Statistic(String name) {
        this.name = name;
    }


    //  ---------------------- CLASS METHODS ----------------------

    public void updateStatistic(String key) {

        synchronized (statisticLock) {
            statisticValues.merge(key, 1, Integer::sum);
        }

    }

    public Map<String, Integer> getStatistic() {

        synchronized (statisticLock) {
            Map<String, Integer> temp = new HashMap<>(statisticValues);
            statisticValues.clear();
            return temp;
        }

    }
}
```

```
1  import java.io.*;
2  import java.nio.file.Files;
3  import java.nio.file.Paths;
4  import java.text.SimpleDateFormat;
5  import java.util.*;
6
7  public class RankingLoggerMerge extends GracefulRunnable {
8
9      private List<String> finishedLogfiles;
10     private int sleepMilliseconds;
11     private int numErrorsToList;
12     private String folderName;
13
14     public RankingLoggerMerge(String name, List<String> finishedLogfiles,
15                               int sleepMilliseconds, int numErrorsToList) {
16         super("RankingLoggerMerge " + name);
17
18         this.finishedLogfiles = finishedLogfiles;
19         this.sleepMilliseconds = sleepMilliseconds;
20         this.numErrorsToList = numErrorsToList;
21         this.folderName = name;
22     }
23
24     // en vez de hacer un merge de todos los archivos generados por los
25     // RankingLoggers se usa el ranking anterior (si hay) y los archivos
26     // generados luego de ese
27     private String getOldRankingFilename() {
28
29         File dir = new File(folderName + "/temp/");
30         File[] files = dir.listFiles((d, name) → name.startsWith("ranking"));
31         if (files.length ≤ 0) {
32             return "";
33         }
34
35         String newest = files[0].getName();
36         for (int i = 1; i < files.length; ++i) {
37             if (newest.compareTo(files[i].getName()) < 0) {
38                 newest = files[i].getName();
39             }
40         }
41
42         Logger.log(logName, "Using old ranking: " + folderName +
43                 "/temp/" + newest, Logger.logLevel.INFO);
44         return folderName + "/temp/" + newest;
45     }
46
47     @Override
48     protected void initWork() {
49         try {
50             if (¬Files.exists(Paths.get(folderName))) {
51                 Files.createDirectory(Paths.get(folderName));
52             }
53         } catch (IOException e) {
54             Logger.log(logName, "Couldn't create folder: " +
55                     folderName, Logger.logLevel.ERROR);
56         }
57
58         try {
59             if (¬Files.exists(Paths.get(folderName + "/temp"))) {
60                 Files.createDirectory(Paths.get(folderName + "/temp"));
61             }
62         } catch (IOException e) {
63             Logger.log(logName, "Couldn't create folder: " +
```

```
64                          folderName + "/temp", Logger.logLevel.ERROR);
65          }
66
67          Logger.log(logName, "Starting RUN", Logger.logLevel.INFO);
68      }
69
70      @Override
71      protected void doWork() {
72
73          try {
74
75              Logger.log(logName, "Going to sleep for " +
76                                  (sleepMilliseconds / 1000) + " seconds",
77                                  Logger.logLevel.INFO);
78              Thread.sleep(sleepMilliseconds);
79              Logger.log(logName, "Waking up", Logger.logLevel.INFO);
80
81              // los archivos a mergear se sacan de la lista y despues se borra
82              List<String> currentFinishedLogFiles;
83              synchronized (finishedLogfiles) {
84                  currentFinishedLogFiles = new LinkedList<>(finishedLogfiles);
85                  finishedLogfiles.clear();
86              }
87
88              if (currentFinishedLogFiles.size() > 0) {
89
90                  //Logger.log(logName, "Doing work", Logger.logLevel.INFO);
91                  Logger.output("[RANKING LOGGER MERGE " + folderName
92                          + "] Going to work on merging files");
93
94                  // get newest old ranking file
95                  String oldRanking = getOldRankingFilename();
96                  if (oldRanking ≠ "") {
97                      currentFinishedLogFiles.add(oldRanking);
98                  }
99
100                 // output file
101                 String timestamp = new SimpleDateFormat("yyyy_MM_dd_HH_mm_ss_SSS
    ").format(new Date());
102                 String outFilename = folderName + "/temp/ranking_" + timestamp;
103                 PrintWriter fileWriter;
104                 try {
105                     fileWriter = new PrintWriter(new FileWriter(outFilename, tru
    e));
106                 } catch (IOException e) {
107                     Logger.log(logName, "Error opening output file: " + outFilename, Logg
    er.logLevel.ERROR);
108                     return;
109                 }
110
111                 // se inician los FileReaders y se lee la primera linea de cada
    archivo
112                 List<BufferedReader> fileReaders = new LinkedList<>();
113                 List<String> lines = new LinkedList<>();
114                 for (int i = 0; i < currentFinishedLogFiles.size(); ++i) {
115                     String filename = currentFinishedLogFiles.get(i);
116                     try {
117                         fileReaders.add(new BufferedReader(new FileReader(filena
    me)));
118                         try {
119                             lines.add(fileReaders.get(i).readLine());
120                         } catch (IOException e) {
121                             Logger.log(logName, "Error reading from file: " + filename,
```

```
    Logger.logLevel.ERROR);
122                             fileReaders.remove(i);
123                         }
124                     } catch (FileNotFoundException e) {
125                         Logger.log(logName, "Error opening file: " + filename, Logger.l
    ogLevel.ERROR);
126                     }
127                 }
128 
129                 LimitedSortedSet<String> mostFrequentErrors = new LimitedSortedS
    et<>(numErrorsToList,
130                         new CountCommaNameComparator());
131 
132                 while (fileReaders.size() > 0 ∧ lines.size() > 0) {
133 
134                     if (shouldStop()) {
135                         for (int i = 0; i < fileReaders.size(); ++i) {
136                             try {
137                                 fileReaders.get(i).close();
138                             } catch (IOException e) {
139                                 Logger.log(logName,
140                                         "Error closing file",
141                                         Logger.logLevel.ERROR);
142                             }
143                         }
144                     }
145 
146                     // se busca el siguiente error con mas apariciones
147                     // no es un merge comun, porque hay registros de la forma "1
    ,error1", "2,error1"
148                     // en cada archivo no puede aparecer dos veces un error
149                     // entonces se hace un merge acumulando las apariciones de l
    a linea actual en cada archivo
150                     List<Integer> smallestIndexes = new LinkedList<>();
151                     smallestIndexes.add(0);
152 
153                     String errMsg = lines.get(0).split(",")[0];
154                     int errMsgCount = Integer.parseInt(lines.get(0).split(",")[1
    ]);
155                     for (int i = 1; i < lines.size(); ++i) {
156                         int compVal = lines.get(i).split(",")[0].compareTo(errMs
    g);
157                         if (compVal < 0) {
158                             smallestIndexes.clear();
159                             smallestIndexes.add(i);
160 
161                             String[] line = lines.get(i).split(",");
162                             errMsg = line[0];
163                             errMsgCount = Integer.parseInt(line[1]);
164                         } else if (compVal ≡ 0) {
165                             smallestIndexes.add(i);
166                             errMsgCount += Integer.parseInt(lines.get(i).split("
    ,")[1]);
167                         }
168                     }
169 
170                     fileWriter.println(errMsg + "," + errMsgCount);
171                     mostFrequentErrors.add(errMsgCount + "," + errMsg);
172 
173                     // advance used files
174                     for (int i = 0; i < smallestIndexes.size(); ++i) {
175                         int smallestIndex = smallestIndexes.get(i);
176 
```

```
177                             try {
178                                 lines.set(smallestIndex, fileReaders.get(smallestInd
    ex).readLine());
179                             } catch (IOException e) {
180                                 Logger.log(logName, "Error reading from file: " + smallestIn
    dex,
181                                         Logger.logLevel.ERROR);
182                                 //lines.remove(smallestIndex);
183                                 //fileReaders.remove(smallestIndex);
184                             }
185                         }
186 
187                         // clear empty files
188                         Iterator<String> itLines = lines.iterator();
189                         Iterator<BufferedReader> itFilesReaders = fileReaders.iterat
    or();
190                         while(itLines.hasNext() ∧ itFilesReaders.hasNext()) {
191                             BufferedReader fr = itFilesReaders.next();
192                             String str = itLines.next();
193                             if (str ≡ null) {
194                                 itLines.remove();
195                                 try {
196                                     fr.close();
197                                 } catch (IOException e) {
198                                     Logger.log(logName,
199                                             "Error closing file",
200                                             Logger.logLevel.ERROR);
201                                 }
202                                 itFilesReaders.remove();
203                             }
204                         }
205                     }
206 
207                     fileWriter.close();
208 
209                     // most frequent errors output
210                     try {
211                         File ranking = new File(folderName + "/ranking");
212                         if (ranking.exists()) {
213                             ranking.delete();
214                         }
215                         PrintWriter freqErrorsFileWriter = new PrintWriter(
216                                 new FileWriter(folderName + "/ranking"));
217                         for (String line : mostFrequentErrors) {
218                             freqErrorsFileWriter.println(line);
219                         }
220                         freqErrorsFileWriter.close();
221                     } catch (IOException e) {
222                         Logger.log(logName, "Error opening output file: "
223                                 + folderName + "/ranking",
224                                 Logger.logLevel.ERROR);
225                     }
226 
227                 }
228 
229             } catch (InterruptedException e) {
230                 Logger.log(logName, "I was interrupted", Logger.logLevel.INFO);
231             }
232         }
233 }
```

```
1  import java.io.FileWriter;
2  import java.io.IOException;
3  import java.io.PrintWriter;
4  import java.nio.file.Files;
5  import java.nio.file.Paths;
6  import java.text.Collator;
7  import java.text.SimpleDateFormat;
8  import java.util.*;
9  import java.util.concurrent.LinkedBlockingQueue;
10
11 public class RankingLogger extends GracefulRunnable {
12
13     private LinkedBlockingQueue inputQueue;
14
15     private List<String> finishedLogfiles;
16     private Map<String, Integer> lines = new HashMap<>();
17     private int maxLines;
18     private String folderName;
19
20     public RankingLogger(String name, LinkedBlockingQueue queue, List<String> fi
   nishedLogfiles, int maxLines) {
21
22         super("RankingLogger " + name);
23
24         this.inputQueue = queue;
25         this.maxLines = maxLines;
26         this.finishedLogfiles = finishedLogfiles;
27         this.folderName = name;
28     }
29
30     // se acumulan en memoria hasta cierto numero de errores con su cantidad de
   apariciones
31     // si se pasa ese numero maximo, se hace un dump de los errores ordenados a
   un archivo
32     private void saveLinesToFile() {
33
34         try {
35
36             List<String> stringLines = new LinkedList<>();
37             for (String key : lines.keySet()) {
38                 String line = key + "," + lines.get(key);
39                 stringLines.add(line);
40             }
41             stringLines.sort(new Comparator<String>() {
42                 @Override
43                 public int compare(String o1, String o2) {
44                     return Collator.getInstance().compare(o1, o2);
45                 }
46             });
47
48             String timestamp = new SimpleDateFormat("yyyy_MM_dd_HH_mm_ss_SSS").f
   ormat(new Date());
49             String filename = logName.split(" ")[1] + "/temp/" + Thread.currentThr
   ead().getName() +
50                     "_" + timestamp;
51
52             PrintWriter fileWriter = new PrintWriter(new FileWriter(filename, tr
   ue));
53             for (String line : stringLines) {
54                 fileWriter.println(line);
55             }
56             fileWriter.close();
57             lines.clear();
```

```
58
59              synchronized (finishedLogfiles) {
60                  finishedLogfiles.add(filename);
61              }
62          } catch (IOException e) {
63              Logger.log("RankingLogger " + logName, e.getMessage(), Logger.logLevel.
    ERROR);
64          }
65      }
66
67      @Override
68      protected void initWork() {
69          try {
70              if (¬Files.exists(Paths.get(folderName))) {
71                  Files.createDirectory(Paths.get(folderName));
72              }
73          } catch (IOException e) {
74              Logger.log(logName, "Couldn't create folder: " + folderName, Logger.logLevel
    .ERROR);
75          }
76
77          try {
78              if (¬Files.exists(Paths.get(folderName + "/temp"))) {
79                  Files.createDirectory(Paths.get(folderName + "/temp"));
80              }
81          } catch (IOException e) {
82              Logger.log(logName, "Couldn't create folder: " + folderName + "/temp", Logger
    .logLevel.ERROR);
83          }
84
85          Logger.log(logName, "Starting RUN", Logger.logLevel.INFO);
86      }
87
88      @Override
89      public void doWork() {
90
91          Logger.log(logName, "Waiting for input", Logger.logLevel.INFO);
92
93          try {
94              String line = inputQueue.take().toString();
95              Logger.log(logName, "Recibi de la cola: " + line, Logger.logLevel.INFO);
96
97              if (lines.size() ≡ maxLines ∧ ¬lines.containsKey(line)) {
98                  Logger.log(logName, "Dumping lines to file", Logger.logLevel.INFO);
99                  saveLinesToFile();
100             }
101             lines.merge(line, 1, Integer::sum);
102
103         } catch (InterruptedException e) {
104             Logger.log(logName, "I was interrupted", Logger.logLevel.INFO);
105         }
106     }
107 }
```

```
1  import java.util.List;
2  import java.util.concurrent.LinkedBlockingQueue;
3  import java.util.regex.Matcher;
4  import java.util.regex.Pattern;
5
6  public class Parser extends GracefulRunnable {
7
8      private LinkedBlockingQueue inputQueue;
9
10     private List<Pattern> patterns;
11     private List<LinkedBlockingQueue> queues;
12
13     public Parser(LinkedBlockingQueue queue, List<Pattern> patterns, List<Linked
   BlockingQueue> queues) {
14         super("Parser");
15
16         this.inputQueue = queue;
17
18         this.patterns = patterns;
19         this.queues = queues;
20
21         Logger.log("Parser", "Creating object", Logger.logLevel.INFO);
22     }
23
24     @Override
25     protected void doWork() {
26
27         Logger.log(logName, "Waiting for input", Logger.logLevel.INFO);
28
29         try {
30             String line = inputQueue.take().toString();
31             Logger.log(logName, "Recibi de la cola: " + line, Logger.logLevel.INFO);
32
33             // pass to queues
34             for (int i = 0; i < patterns.size(); ++i) {
35                 Matcher matchResult = patterns.get(i).matcher(line);
36                 if (matchResult.matches()) {
37                     String resultString = matchResult.group(1);
38                     for (int  j = 2; j ≤ matchResult.groupCount(); ++j) {
39                         resultString += " " + matchResult.group(j);
40                     }
41                     queues.get(i).put(resultString);
42                 }
43             }
44
45         } catch (InterruptedException e) {
46             Logger.log(logName, "I was interrupted", Logger.logLevel.INFO);
47         }
48     }
49 }
```

```
1  import java.io.FileWriter;
2  import java.io.IOException;
3  import java.io.PrintWriter;
4  import java.text.SimpleDateFormat;
5  import java.util.Date;
6
7  public class Logger {
8
9      public static logLevel currentLogLevel = logLevel.INFO;
10     private static PrintWriter logWriter = null;
11
12     public enum logLevel {
13         ERROR,
14         WARNING,
15         INFO
16     }
17
18     public static logLevel intToLogLevel(int i) {
19         switch (i) {
20             case 0:
21                 return logLevel.ERROR;
22             case 1:
23                 return logLevel.WARNING;
24             case 2:
25                 return logLevel.INFO;
26             default:
27                 return logLevel.INFO;
28         }
29     }
30
31     private static String logLevelToString(logLevel level) {
32         switch(level) {
33             case ERROR:
34                 return "[ERROR]";
35             case WARNING:
36                 return "[WARNING]";
37             case INFO:
38                 return "[INFO]";
39             default:
40                 return "[INVALID LOGLEVEL]";
41         }
42     }
43
44     public static void init(String filename) {
45         try {
46             logWriter = new PrintWriter(new FileWriter(filename));
47
48             String timeStamp = new SimpleDateFormat("yyyy/MM/dd/ HH:mm:ss").format(
   new Date());
49             logWriter.println("***************" + timeStamp + "***************");
50         } catch (IOException e) {
51             log("Logger", "Couldn't open logfile for writing", logLevel.ERROR);
52         }
53     }
54
55     public static void close() {
56         if (logWriter ≠ null) {
57             logWriter.close();
58         }
59     }
60
61     public static void log(String name, String message, logLevel level) {
62
```

```
63          String logLine = Thread.currentThread().getName() + "\t" +
64                  logLevelToString(level) + "\t" + name + ":" + message;
65
66          // output to screen
67          if (currentLogLevel.ordinal() ≥ level.ordinal()) {
68              System.out.println(logLine);
69          }
70          // output to logfile
71          if (logWriter ≠ null) {
72              logWriter.println(logLine);
73          }
74      }
75
76      public static void output(String outString) {
77          System.out.println(outString);
78          logWriter.println(outString);
79      }
80
81  }
```

Mar 26, 18 2:10 **LimitedSortedSet.java** Page 1/1

```
1  import java.util.Collection;
2  import java.util.Comparator;
3  import java.util.TreeSet;
4
5  // un sorted set que automaticamente borra elementos de si mismo si se pasa del
   maximo
6  class LimitedSortedSet<E> extends TreeSet<E> {
7
8      private int maxSize;
9
10     LimitedSortedSet( int maxSize ) {
11         this.maxSize = maxSize;
12     }
13
14     LimitedSortedSet( int maxSize, Comparator<? super E> comparator ) {
15         super(comparator);
16         this.maxSize = maxSize;
17     }
18
19     @Override
20     public boolean addAll( Collection<? extends E> c ) {
21         boolean added = super.addAll( c );
22         if( size() > maxSize ) {
23             E firstToRemove = (E)toArray( )[maxSize];
24             removeAll( tailSet( firstToRemove ) );
25         }
26         return added;
27     }
28
29     @Override
30     public boolean add( E o ) {
31         boolean added =  super.add( o );
32         /*if( size() > maxSize ) {
33             E firstToRemove = (E)toArray( )[maxSize];
34             removeAll( tailSet( firstToRemove ) );
35         }*/
36         while (size() > maxSize) {
37             remove(last());
38         }
39         return added;
40     }
41 }
```

```
1  public abstract class GracefulRunnable implements Runnable {
2
3      private volatile boolean keepAlive = true;
4      protected String logName;
5
6      public GracefulRunnable(String name) {
7          this.logName = name;
8
9          Logger.log(name, "Creating object", Logger.logLevel.INFO);
10     }
11
12     public void stopKeepAlive() {
13         keepAlive = false;
14     }
15
16     @Override
17     public void run() {
18
19         initWork();
20         while (keepAlive) {
21             doWork(); // if doWork is time consuming, call shouldStop periodical
   ly
22         }
23         endWork();
24     }
25
26     protected void initWork() {
27         Logger.log(logName,"Starting RUN", Logger.logLevel.INFO);
28     }
29
30     protected abstract void doWork();
31
32     protected void endWork() {
33         Logger.log(logName,"Ending RUN", Logger.logLevel.INFO);
34     }
35
36     protected boolean shouldStop() { return ¬keepAlive; }
37 }
```

```
1  import java.io.FileWriter;
2  import java.io.IOException;
3  import java.io.PrintWriter;
4  import java.nio.file.Files;
5  import java.nio.file.Paths;
6  import java.util.concurrent.LinkedBlockingQueue;
7
8  public class FileLogger extends GracefulRunnable {
9
10     private LinkedBlockingQueue inputQueue;
11
12     private String filename;
13     private PrintWriter fileWriter;
14
15     public FileLogger(String name, LinkedBlockingQueue queue) {
16         super("FileLogger " + name);
17         this.inputQueue = queue;
18         this.filename = name;
19     }
20
21     @Override
22     protected void initWork() {
23
24         try {
25             if (¬Files.exists(Paths.get(filename))) {
26                 Files.createDirectory(Paths.get(filename));
27             }
28             fileWriter = new PrintWriter(new FileWriter(filename + "/" + filenam
   e, true));
29         } catch (IOException e) {
30             Logger.log(logName, "Couldn't create file: " + filename + "/" + filename, Lo
   gger.logLevel.ERROR);
31             fileWriter = null;
32         }
33
34         Logger.log(logName, "Starting RUN", Logger.logLevel.INFO);
35     }
36
37     @Override
38     protected void doWork() {
39
40         Logger.log(logName, "Waiting for input", Logger.logLevel.INFO);
41
42         try {
43             String line = inputQueue.take().toString();
44             Logger.log(logName, "Recibi de la cola: " + line, Logger.logLevel.INFO);
45
46             fileWriter.println(line);
47
48         } catch (InterruptedException e) {
49             Logger.log(logName, "I was interrupted", Logger.logLevel.INFO);
50         }
51     }
52
53     @Override
54     protected void endWork(){
55         Logger.log(logName, "Ending RUN", Logger.logLevel.INFO);
56
57         fileWriter.close();
58     }
59 }
```

```java
import java.util.Comparator;


// es un comparador para string de la forma "numero,string" por ejemplo "3,hola"
public class CountCommaNameComparator implements Comparator<String> {
    @Override
    public int compare(String o1, String o2) {

        int o1_int = Integer.parseInt(o1.split(",")[0]);
        String o1_str = o1.split(",")[1];
        int o2_int = Integer.parseInt(o2.split(",")[0]);
        String o2_str = o2.split(",")[1];

        if (o2_int < o1_int) {
            return -1;
        }
        if (o2_int > o1_int) {
            return 1;
        }

        return o2.compareTo(o1);
    }
}
```

Mar 28, 18 13:41 **Table of Content** Page 1/1

Table of Contents