Assignment 5 documentation

Streams

Litan Virgil

Group 30425

TA: Claudia Pop

Contents

1. Functional Requirements

Define a class MonitoredData having startTime, endTime and activityLabel as instance variables and read the input file data into the data structure monitoredData of type List. Using stream processing techniques and lambda expressions introduced by Java 8, write the following set of short programs for processing the monitoredData.

1. Count the distinct days that appear in the monitoring data.

2. Determine a map of type that maps to each distinct action type the number of occurrences in the log. Write the resulting map into a text file.

3. Generates a data structure of type Map> that contains the activity count for each day of the log (task number 2 applied for each day of the log) and writes the result in a text file.

4. Determine a data structure of the form Map that maps for each activity the total duration computed over the monitoring period. Filter the activities with total duration larger than 10 hours. Write the result in a text file.

5. Filter the activities that have 90% of the monitoring samples with duration less than 5 minutes, collect the results in a List containing only the distinct activity names and write the result in a text file.Objectives

2. Problem analysis

The first step in order to be able to process the data is to acquire it from the given file. Second step was to actually process the data an finally write the results into files.

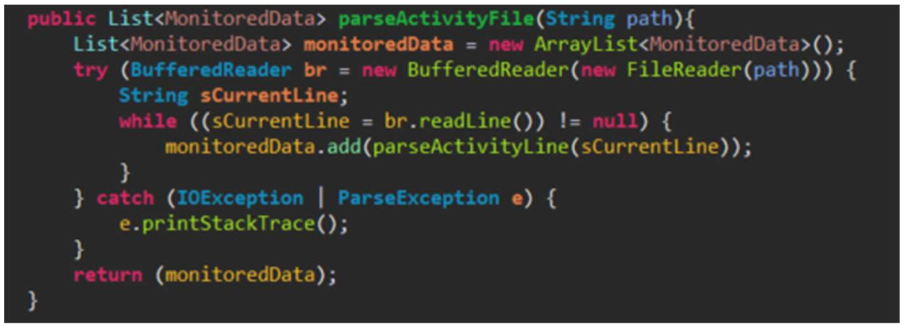
3. Design

The data parser is implemented using the singleton design pattern. Each task is solved in a separated static procedure in the main class.

4. Implementation

4.1 Reading the data

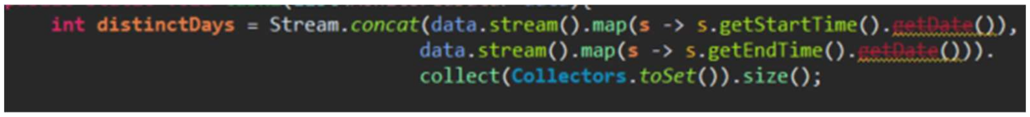
The data was read from the file using a BufferedReader and each line was parsed using a dedicated method and then added to the return list.



```java
public List<MonitoredData> parseActivityFile(String path){
    List<MonitoredData> monitoredData = new ArrayList<MonitoredData>();
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        String sCurrentLine;
        while ((sCurrentLine = br.readLine()) != null) {
            monitoredData.add(parseActivityLine(sCurrentLine));
        }
    } catch (IOException | ParseException e) {
        e.printStackTrace();
    }
    return (monitoredData);
}
```

4.2 Solving task 1

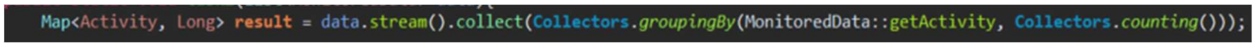
As an activity could start in a day and finish in the next one, looking just at the start date of an activity is not enough. So, firstly all days were concatenated in a stream, then in order to remove duplicates it was converted to a set and finally the size of the set was the result.



```java
    int distinctDays = Stream.concat(data.stream().map(s -> s.getStartTime().getDate()),
                                     data.stream().map(s -> s.getEndTime().getDate())).
                                     collect(Collectors.toSet()).size();
```

4.2 Solving task 2

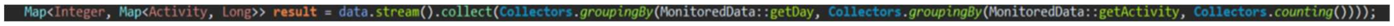
In order to map the activities with the corresponding number of occurrences, each the collector groupingBy was used.



```java
    Map<Activity, Long> result = data.stream().collect(Collectors.groupingBy(MonitoredData::getActivity, Collectors.counting()));
```

4.4 Solving task 3

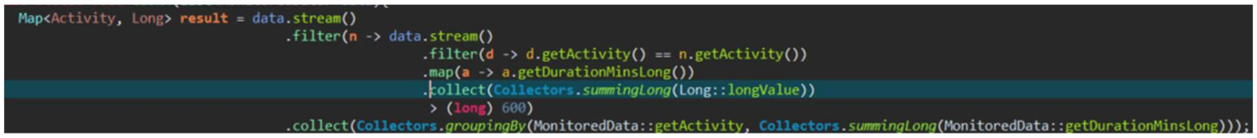
The same strategy as above was used but this time a nested grouping was needed. The first level of nesting beeing the day of the activity.



```java
    Map<Integer, Map<Activity, Long>> result = data.stream().collect(Collectors.groupingBy(MonitoredData::getDay, Collectors.groupingBy(MonitoredData::getActivity, Collectors.counting())));
```

4.5 Solving task 4

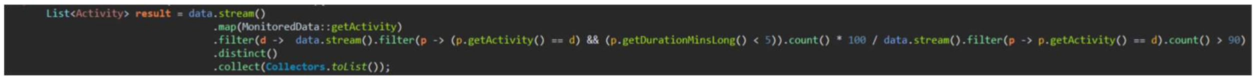
In order to solve this task nested stream processing was needed. So, in order to filter out activities with greater total time than 10 hours the total duration for each activity had to be computed several times. Further, the remaining activities where grouped and the total time was re-computed.



```java
    Map<Activity, Long> result = data.stream()
                                     .filter(n -> data.stream()
                                                      .filter(d -> d.getActivity() == n.getActivity())
                                                      .map(a -> a.getDurationMinsLong())
                                                      .collect(Collectors.summingLong(Long::longValue))
                                                       > (long) 600)
                                     .collect(Collectors.groupingBy(MonitoredData::getActivity, Collectors.summingLong(MonitoredData::getDurationMinsLong)));
```

4.6 Solving task 5

For the last task also multiple stream processing instructions were needed. The filter step is the most important. In order to determine the total number of activities of a given type a simple count() was used, and in order to determine the number of activities with a smaller time than 5 minutes a filter() was needed.



```java
    List<Activity> result = data.stream()
                                .map(MonitoredData::getActivity)
                                .filter(d ->  data.stream().filter(p -> (p.getActivity() == d) && (p.getDurationMinsLong() < 5)).count() * 100 / data.stream().filter(p -> p.getActivity() == d).count() > 90)
                                .distinct()
                                .collect(Collectors.toList());
```