Student: Horatiu-Stefan Spaniol

Group: 30422

Programming Techniques

Assignment 5

**Problem Specification**

**Homework 5**

Stream Processing using Lambda Expressions

**Description**

A smart house features a set of sensors that may be used to record the behavior of a person living in the house. The historical log of the person’s activity is stored as tuples (startTime, endTime, activityLabel), where startTime and endTime represent the date and time when each activity has started and ended while the activity label represents the type of activity performed by the person: Leaving, Toileting, Showering, Sleeping, Breakfast, Lunch, Dinner, Snack, Spare\_Time/TV, Grooming.

The attached log file Activities.txt contains a set of activity records over a certain period of time.

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.

This project is focused on working with lambda expressions and with streams. Both of these features have been introduced in Java 8. Lambda expressions allow the user to implement functions more easily, while streams allow the user to process data with the help of some aggregate operations.

After reading the information from the given text file, we have to process data and then put them into text files.

**Streams in Java 8**

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-  
**Intermediate Operations:**

1. **map:** The map method is used to map the items in the collection to other objects according to the Predicate passed as argument.  
   List number = Arrays.asList(2,3,4,5);  
   List square = number.stream().map(x->x\*x).collect(Collectors.toList());
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
3. **sorted:** The sorted method is used to sort the stream.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List number = Arrays.asList(2,3,4,5,3);  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

//a simple program to demonstrate the use of stream in java

**import** java.util.\*;

**import** java.util.stream.\*;

**class** Demo

{

**public** **static** **void** main(String args[])

  {

    // create a list of integers

    List<Integer> number = Arrays.asList(2,3,4,5);

    // demonstration of map method

    List<Integer> square = number.stream().map(x -> x\*x).

                           collect(Collectors.toList());

    System.out.println(square);

    // create a list of String

    List<String> names =

                Arrays.asList("Reflection","Collection","Stream");

    // demonstration of filter method

    List<String> result = names.stream().filter(s->s.startsWith("S")).

                          collect(Collectors.toList());

    System.out.println(result);

    // demonstration of sorted method

    List<String> show =

            names.stream().sorted().collect(Collectors.toList());

    System.out.println(show);

    // create a list of integers

    List<Integer> numbers = Arrays.asList(2,3,4,5,2);

    // collect method returns a set

    Set<Integer> squareSet =

         numbers.stream().map(x->x\*x).collect(Collectors.toSet());

    System.out.println(squareSet);

    // demonstration of forEach method

    number.stream().map(x->x\*x).forEach(y->System.out.println(y));

    // demonstration of reduce method

**int** even =

       number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

    System.out.println(even);

  }

}

Output:

[4, 9, 16, 25]

[Stream]

[Collection, Reflection, Stream]

[16, 4, 9, 25]

4

9

16

25

6

**Lambda Expressions in Java 8**

Lambda expressions basically express instances of [functional interfaces](https://www.geeksforgeeks.org/functional-interfaces-java/) (An interface with single abstract method is called functional interface. An example is java.lang.Runnable). lambda expressions implement the only abstract function and therefore implement functional interfaces

lambda expressions are added in Java 8 and provide below functionalities.

* Enable to treat functionality as a method argument, or code as data.
* A function that can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand.

Following are the important characteristics of a lambda expression.

* **Optional type declaration** − No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
* **Optional parenthesis around parameter** − No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
* **Optional curly braces** − No need to use curly braces in expression body if the body contains a single statement.
* **Optional return keyword** − The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

**Important points:**

* The body of a lambda expression can contain zero, one or more statements.
* When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
* When there are more than one statements, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.
* // Java program to demonstrate lambda expressions
* // to implement a user defined functional interface.
* // A sample functional interface (An interface with
* // single abstract method
* **interface** FuncInterface
* {
* // An abstract function
* **void** abstractFun(**int** x);
* // A non-abstract (or default) function
* **default** **void** normalFun()
* {
* System.out.println("Hello");
* }
* }
* **class** Test
* {
* **public** **static** **void** main(String args[])
* {
* // lambda expression to implement above
* // functional interface. This interface
* // by default implements abstractFun()
* FuncInterface fobj = (**int** x)->System.out.println(2\*x);
* // This calls above lambda expression and prints 10.
* fobj.abstractFun(5);
* }
* }
* Output:
* 10

For example in solving the first problem, counting the distinct days, I mapped the “day” values and collected the distinct values into a List of integers, for finding the exact number of days, I used the size method of the list.

Here is the method I used:

private static void countDays() // Count how many days of monitored data appears in the log.

{

int differentDays = *monitoredData*.stream().distinct().collect(Collectors.*groupingBy*(MonitoredData::getDay)).size();

*printWriter*.println("1. Count how many days of monitored data appears in the log.");

*printWriter*.println("Different days: " + differentDays + "\n");

}

In the second problem where we have to find the distinct activities the number of apparitions in the text file. I collected the labels by grouping them in function of their names while counting them.

Here is the method I used:

private static void countActivities()

{

Map<String, Long> map = *monitoredData*.stream().collect(Collectors.*groupingBy*(MonitoredData::getActivity,Collectors.*counting*()));

*printWriter*.println("2. Determine a map of type <String, Integer> that maps to each distinct action type the number of occurrences in the log. Write the resulting map into a text file");

map.keySet().forEach(string -> *printWriter*.println(string + "\t\t" + map.get(string)));

}

The third problem is like the second problem but separated for each day. The only difference that they are grouped first after day.

Here is the method I used:

private static void countActivitiesPerDay()

{

Map<String, Map<String, Long>> activityCountForEachDayMap = *monitoredData*.stream()

.collect(Collectors

.*groupingBy*(MonitoredData::getDay,Collectors

.*groupingBy*(MonitoredData::getActivity,

Collectors.*counting*())));

*printWriter*.println("\n3. Generates a data structure of type Map<Integer, Map<String, Integer>> 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");

activityCountForEachDayMap.forEach((activity, map) -> {

*printWriter*.println("\n" + activity);

map.keySet().forEach(string -> *printWriter*.println(string + "\t\t" + map.get(string)));

});

}

The fourth exercise asks us to find the total duration of an activity and put in the text file if it has over 10 hours. So I grouped again after the activity label and then added the number of milliseconds each activity took. Converted the sum into hours and if it was bigger than 10 put it into the file.

Here is the method I used:

private static void activityDurationPerTotal2() throws IOException

{

SimpleDateFormat dataFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

Map<String, Long> map = *monitoredData*.stream().distinct().collect(Collectors.*groupingBy*(MonitoredData::getActivity, Collectors.*summingLong*(md -> {

try

{

return (dataFormat.parse(md.getEndTime()).getTime() - dataFormat.parse(md.getStartTime()).getTime()) / 1000;

}

catch (ParseException e)

{

e.printStackTrace();

}

return 0;

})));

*printWriter*.println("\n4. Determine a data structure of the form Map<String, DateTime> 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.");

map.forEach((activity, duration) -> {

*printWriter*.print(activity + "\t" + duration + " seconds.");

if(duration > 3600)

{

*printWriter*.println(" Takes more than 10 hours(3600 seconds).");

}

else

{

*printWriter*.println();

}

});

And this is the method I used before trying it out with streams:

private static void activityDurationPerTotal() throws IOException

{

HashMap<String, Long> map = new HashMap<String, Long>();

HashSet<String> differentActivities = new HashSet<String>();

*monitoredData*.forEach(md -> differentActivities.add(md.getActivity()));

differentActivities.forEach(string -> map.put(string, (long) 0));

SimpleDateFormat dataFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

*monitoredData*.forEach(md -> {

Long durationOfActivity = map.get(md.getActivity());

Long newDurationOfActivity = (long) 0;

try

{

newDurationOfActivity = durationOfActivity + (dataFormat.parse(md.getEndTime()).getTime() - dataFormat.parse(md.getStartTime()).getTime()) / 1000;

map.put(md.getActivity(), newDurationOfActivity);

}

catch (ParseException e)

{

e.printStackTrace();

}

});

*printWriter*.println("\n4. Determine a data structure of the form Map<String, DateTime> 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.");

map.forEach((activity, duration) -> {

*printWriter*.print(activity + "\t" + duration + " seconds.");

if(duration > 3600)

{

*printWriter*.println(" Takes more than 10 hours(3600 seconds).");

}

else

{

*printWriter*.println();

}

});

}

In the fifth exercise, we have to find the activities that take under 5 minutes in 90% of cases. I tried to do as in the 4-th exercise but things got a bit messy and I gave up. I implemented using simple lambda expressions and logic, and not with any streams.

Here is the method I used:

private static void filterActivitiesOver90PercentUnder5Minutes() throws IOException

{

HashMap<String, Float> under5Map = new HashMap<String, Float>(); // map will contain: activity

HashMap<String, Float> over5Map = new HashMap<String, Float>();

*monitoredData*.forEach(md -> under5Map.put(md.getActivity(), (float) 0));

*monitoredData*.forEach(md -> over5Map.put(md.getActivity(), (float) 0));

SimpleDateFormat dataFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

*monitoredData*.forEach(md -> {

try

{

Float numberOfTimes = null;

if(((dataFormat.parse(md.getEndTime()).getTime() - dataFormat.parse(md.getStartTime()).getTime()) / 1000) > 300)

{

numberOfTimes = over5Map.get(md.getActivity());

numberOfTimes++;

over5Map.put(md.getActivity(), numberOfTimes);

}

else

{

numberOfTimes = under5Map.get(md.getActivity());

numberOfTimes = numberOfTimes + 1;

under5Map.put(md.getActivity(), numberOfTimes);

}

} catch (ParseException e)

{

e.printStackTrace();

}

});

*printWriter*.println("\n5. Filter the activities that have 90% of the monitoring samples with duration less than 5 minutes, collect the results in a List<String> containing only the distinct activity names and write the result in a text file.");

under5Map.forEach((activity, times) -> {

float percentage = times / (times + over5Map.get(activity));

if(percentage > 0.9)

{

*printWriter*.println(activity + ":\nPercentage of < 5 mins: " + percentage\*100 + "%");

}

});

*printWriter*.close();

}

Conclusion:

Through hard work and intensive testing, I have been able to create an application for processing data with streams and lambda expressions. In the end the application can simply be ran and solve the requirements needed in this assignment.

Bibliography:

<https://www.geeksforgeeks.org/stream-in-java/>

https://www.geeksforgeeks.org/lambda-expressions-java-8/

https://www.tutorialspoint.com/java8/java8\_lambda\_expressions.htm