**Lambda expressions and**

**Stream processing**

***-Project documentation-***

Student: Cocoş Tudor-Adrian

Group: 30423

**Contents:**

1. Objective of the assignment
2. Problem analysis
   1. Assumptions
   2. Modelling
   3. Scenarios
   4. Use cases
3. Design
   1. Design choices
   2. UML diagram
   3. Class design
   4. Algorithms
   5. User Interface
4. Implementation
5. Conclusions
6. Bibliography

**1) Objective of the assignment**

The main objective of this assignment is to design and implement an application to analyse the behaviour of a person recorded by a set of sensors.

The secondary objectives are the steps needed to be taken to achieve the main objective. First of all, we need to analyse the concept of Lambda Expressions and Stream Processing because they introduce a whole new way of looking at data processing through the introduction of stream processing in Java 8. In order to achieve any of this the lambda expression have been added, in order for Java to keep up with the other programming languages when it comes to functional programming. After the theoretical concepts have been studied we get to apply them by firstly extracting the data about that person from a given text file. After that, based on the tasks needed to be performed we will shape the methods to our solution in order to obtain the wanted results.

**2) Problem analysis**

*a) Assumptions*

Given the assignment, we assume that the application should have be able to read data from the given text file in a certain configuration: date and time of the start, date and time of the end and the name of the activity. They need to be stored in tuples of strings: start time, end time and activity label, where the activities are predefined and limited in number.

*b) Modelling*

To make the data as simple as possible there are a few fields to describe the main entity: Monitored Data. The three strings that define this class represent the date and time at which the activity has started, the date and time at which the activity has ended and the label of the activity.

*c) Scenarios*

The application is designed to work on the data set that is given. In terms of scenarios and usability it doesn’t have any other outside of this task. The idea is for us to learn and to try to master the use of streams and lambda expressions. The experienced gained is really useful for the time when these concepts will be applied in the real life, while working.

The focus of the scenario is to interpret the data that was collected by recording the behaviour of a person with sensors. The data consists of the daily habits and activities of that subject on a span of several days. We will need to follow closely the tasks provided and to provide the right answer to them by working closely, if not exclusively, with streams and lambda expressions.

*d) Use cases*

As mentioned earlier, the one and only use of this application is to display the information asked in the tasks on the given data. Only the contents of the input text file may change, but even there it should maintain the same structure in order to achieve data integrity.

**3) Design**

*a) Design choices*

The design of the whole application revolves around the use of streams and lambda functions, making it possible to implement it fully in 2 classes with a relative short size of the implementation for each.

A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements. Stream operations are either intermediate or terminal. Intermediate operations return a stream, so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result, a chain of stream operations being also known as operation pipeline.

Stream is not Collection, as Collections focus on storage of data elements for efficient access, while Streams focus on aggregate computations on data elements from a data source that could be a collection. Streams are consuming data from collections, arrays or I/O resources. Another view: streams are smart iterators over collections.

Streams allow writing code that is declarative (concise and reliable), composable (increase flexibility), parallelizable (increase performance) and pipelined. Many stream operations return a stream thus allowing operations to be chained into large pipelines. Pipeline enables optimizations such as laziness and short-circuiting, while a pipeline of operations can be viewed as database-like query on the data source.

Streams have no storage: a collection must store in memory (internal or external) all its elements. A stream has no storage => it does not store elements, as it pulls (on-demand) elements from a data and passes them to a pipeline of operations for processing. Streams can represent a sequence of infinite elements. A stream pulls its elements from a data source that can be a collection, a function that generates data, an I/O channel, etc. The design of streams is based on internal iteration. They are designed to support functional programming. Stream operations don’t modify the source data. Similar to Functional Programming, you specify what operations should be performed on the data elements using the built-in methods provided by Streams API – typically by passing a lambda expression to those methods thus customizing the behaviour of those operations. Streams are designed to be processed in parallel with no additional work from the developers. They support lazy operations and cannot be reused.

Parallel stream processing is useful because: modern computers are equipped with multicore processors from which the emerging need for parallel processing. Java Streams may process the elements in parallel. Streams take care of the details of using the Fork/Join framework internally.

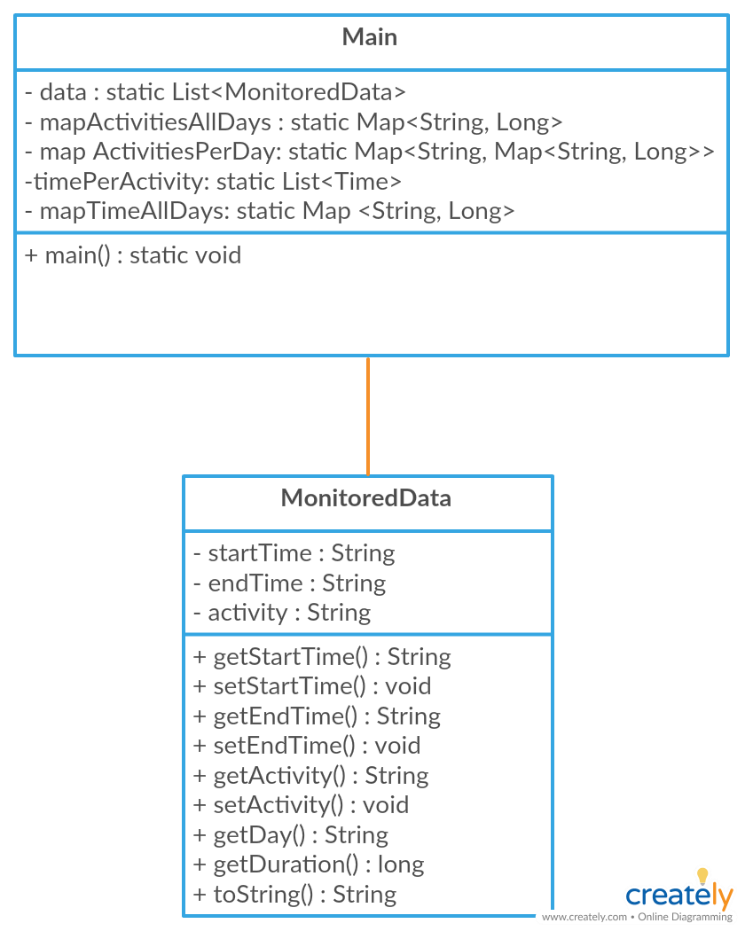
Stream operations consist of: intermediate operations (or lazy operations) and terminal operations (or eager operations). A stream is inherently lazy until you call a terminal operation on it. An intermediate operation on a stream produces another stream. Each intermediate operation takes elements from an input stream and transforms the elements to produce an output stream. The terminal operation takes inputs from a stream and produces the result.

Intermediate operations return another stream as the return type. They allow operations to be connected to form a query. Intermediate operations don’t perform any processing until a terminal operation is invoked on the stream pipeline—they’re lazy. The main reason is Intermediate operations can usually be merged and processed into a single pass by the terminal operation.

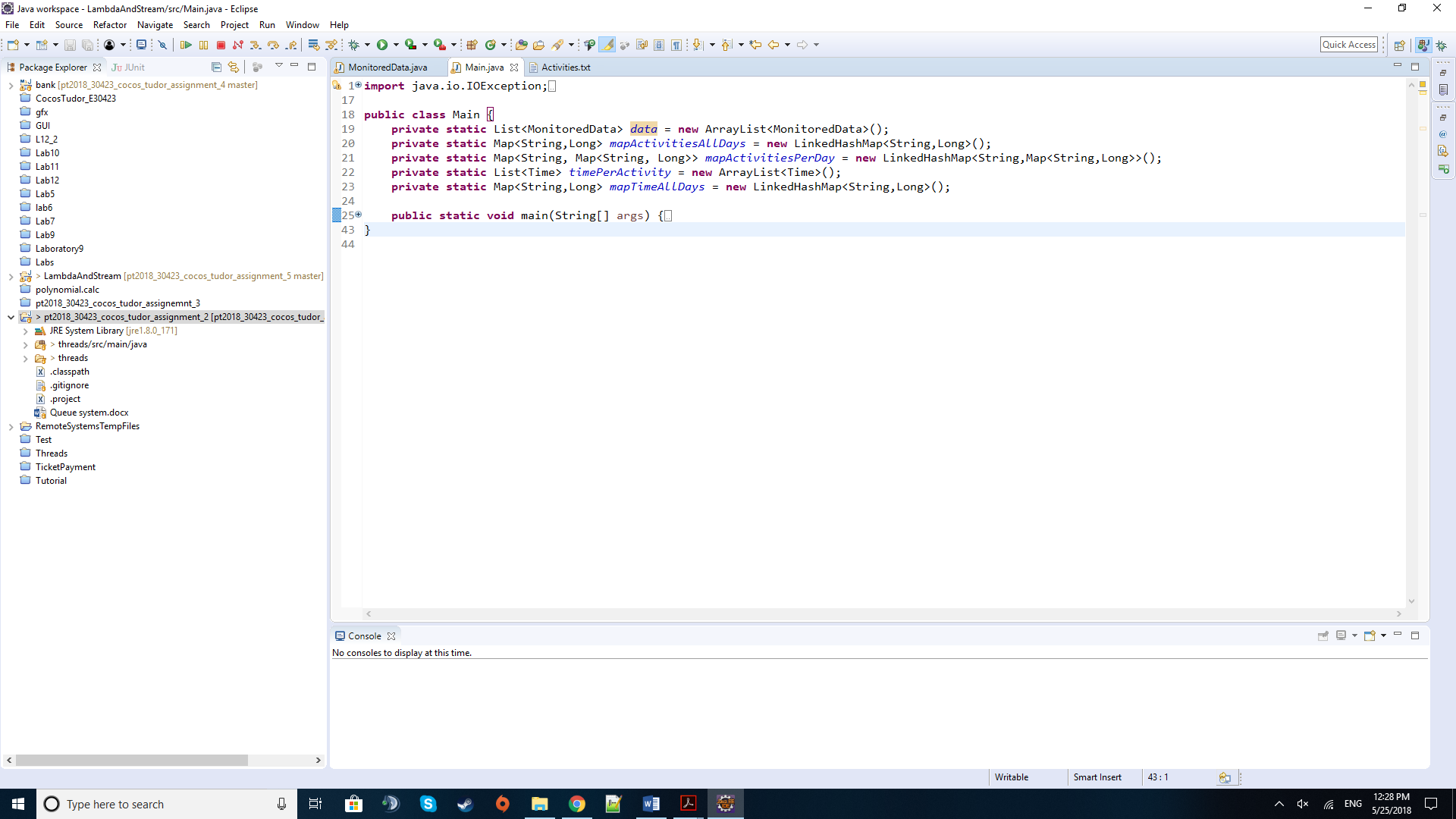
Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behaviour of the operation. Most of those operations must be both non-interfering and stateless. A function is non-interfering when it does not modify the underlying data source of the stream, with a function being stateless when the execution of the operation is deterministic.

Streams can be created from various data sources, especially collections. Lists and Sets support new methods: stream() and parallelStream(), to either create a sequential or a parallel stream. Java 8 streams cannot be reused. As soon as you call any terminal operation the stream is closed.

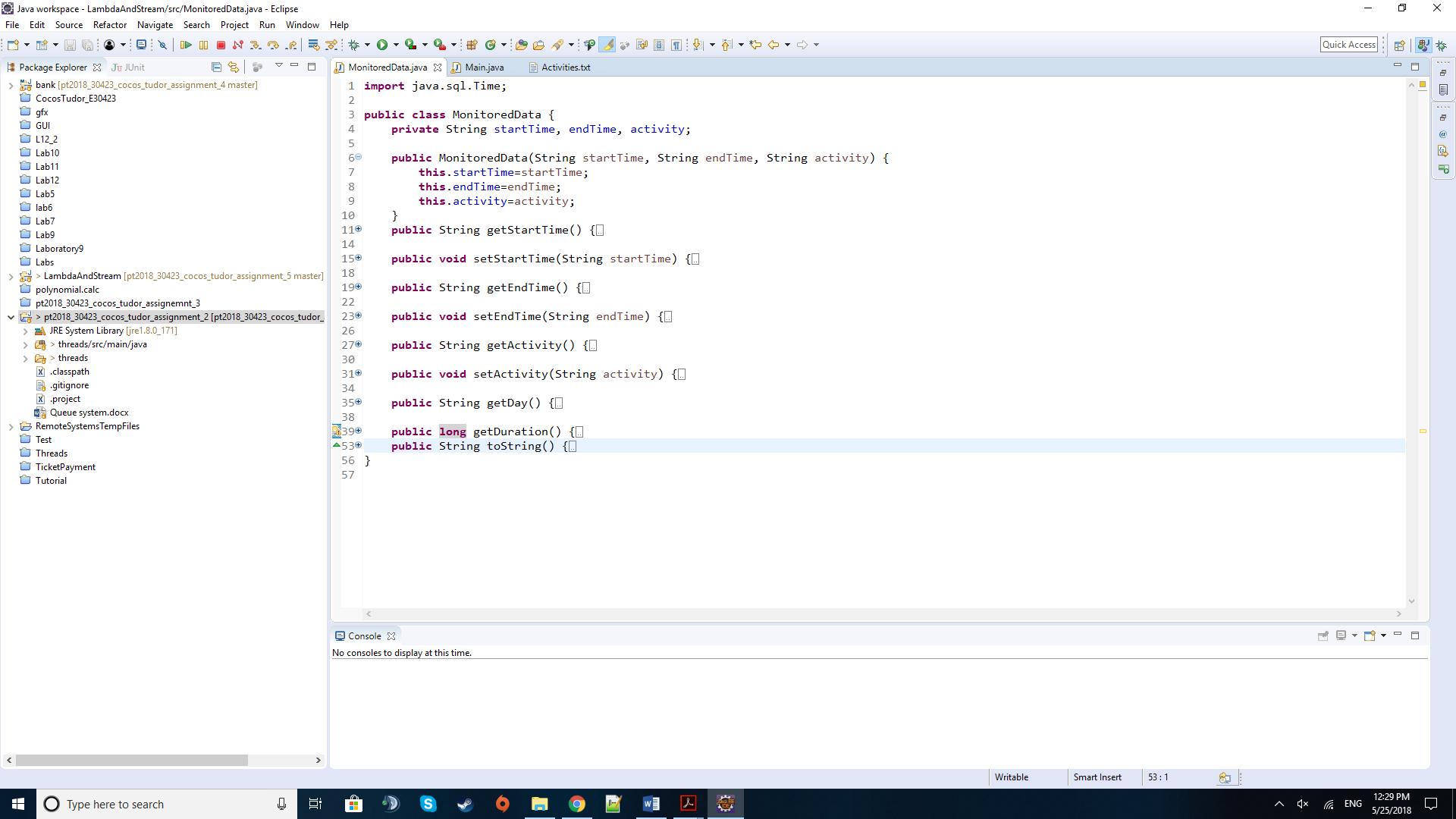
Streams support plenty of different operations: from intermediate like map or filter to final ones such as collect for forEach. Collect is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a List, Set or Map. Collect accepts a Collector which consists of four different operations: a supplier, an accumulator, a combiner and a finisher. If for example, you need a Set instead of List we can just use Collectors.toSet(). Collectors are extremely versatile: aggregations on the elements of the stream can also be created, e.g. determining the time of an activity during the whole time span, on the 14 days. In order to transform the stream elements into a map, we have to specify how both the keys and the values should be mapped.

*b) UML Diagrams*

*c) Class design*



***Main class***

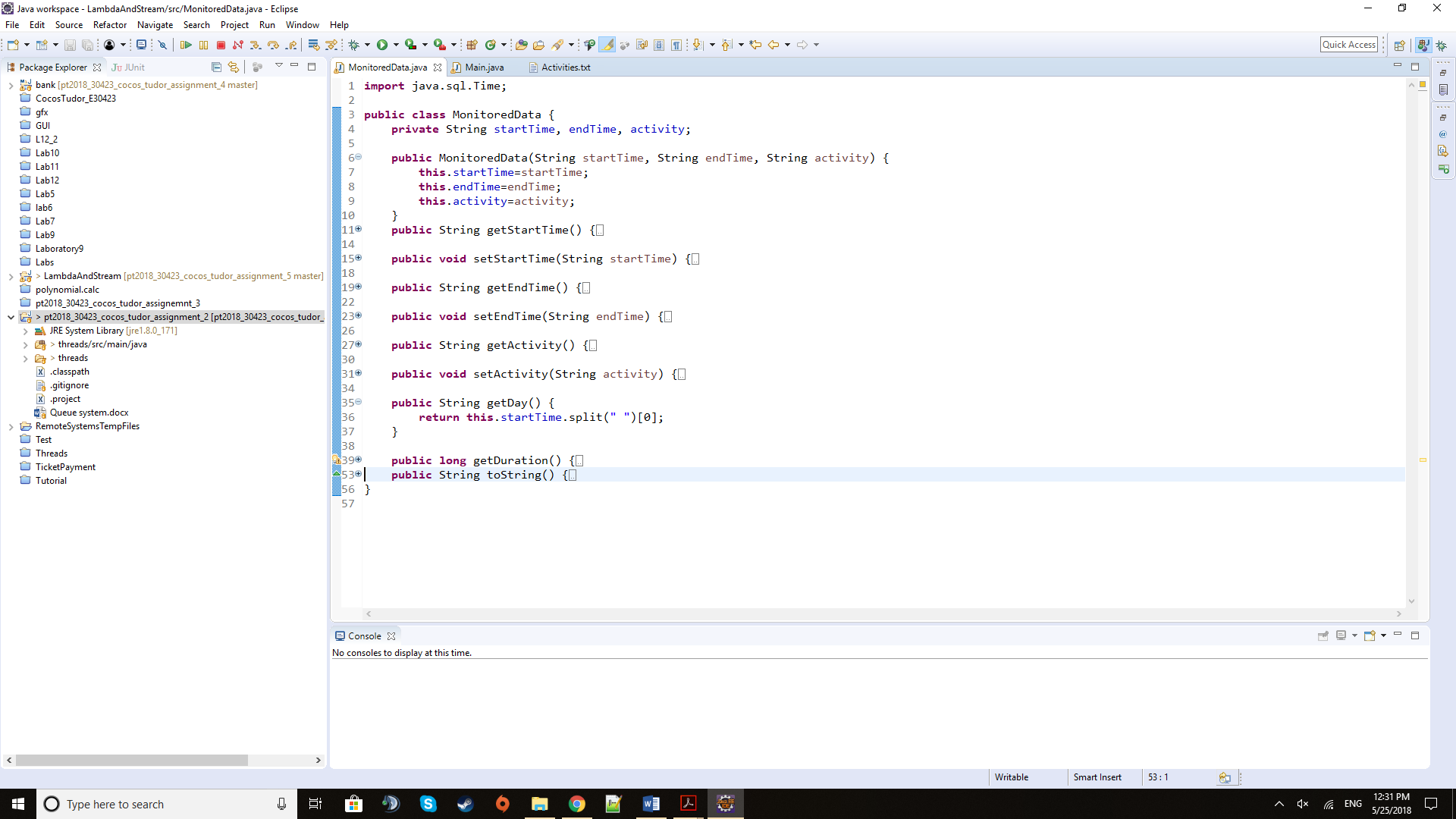


***MonitoredData class***

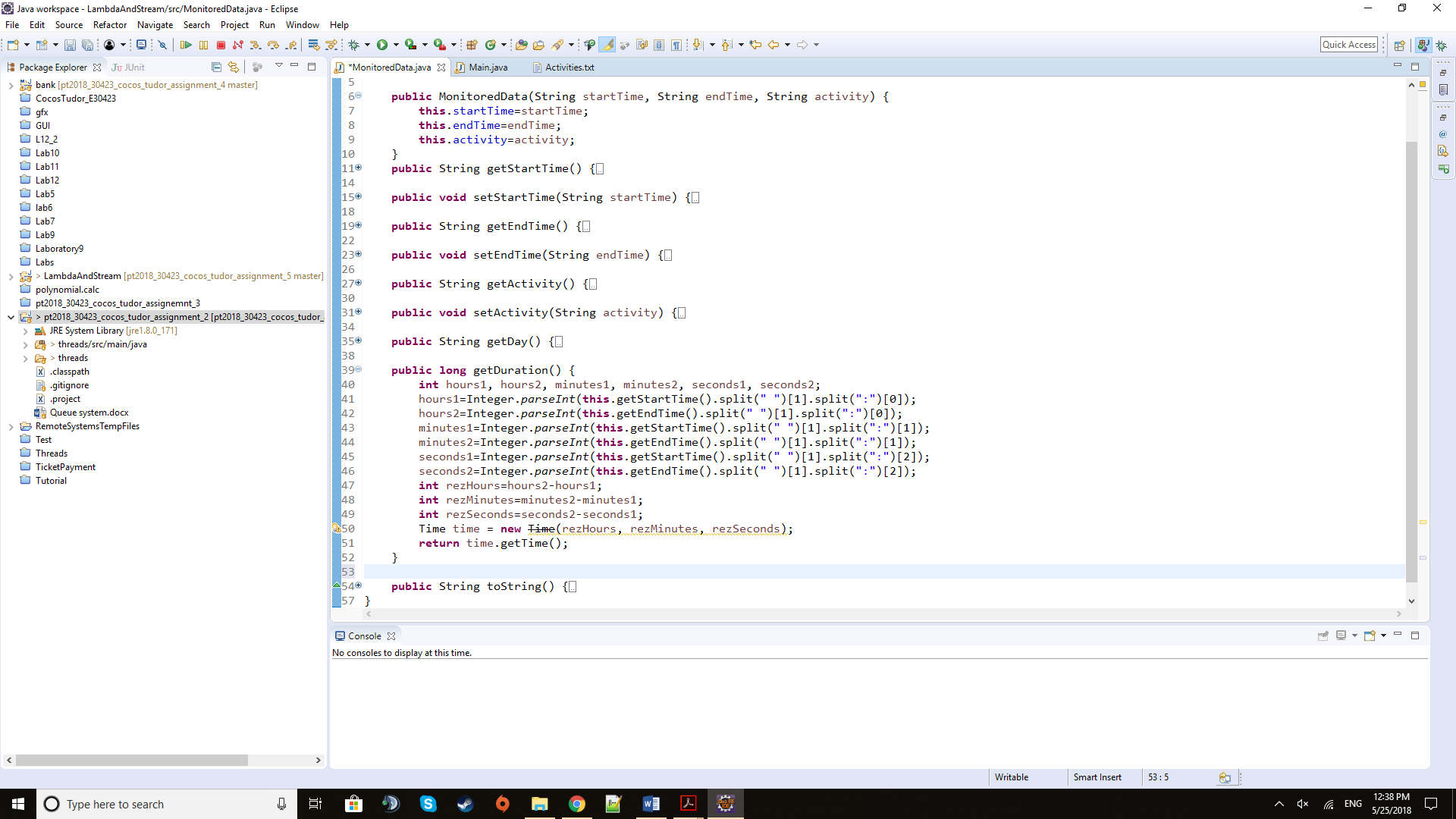
*d) Algorithms*

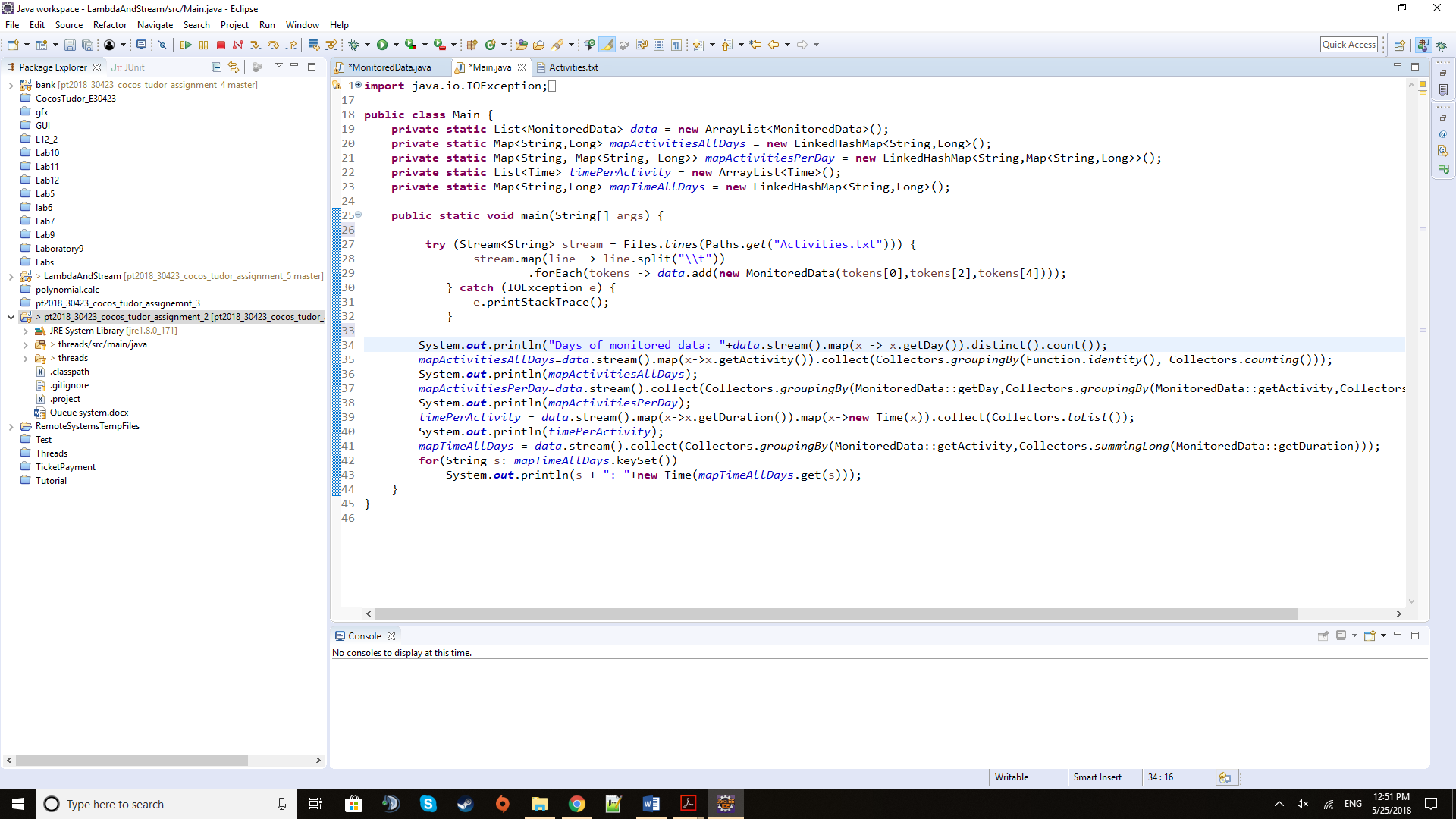
As for algorithms there are no special mathematical-based algorithms like in the polynomial processing assignment. Instead, we rely heavily on the lambda expressions and stream operations to obtain the wanted results. Their implementation and the reasoning behind their use will be described in the section below.

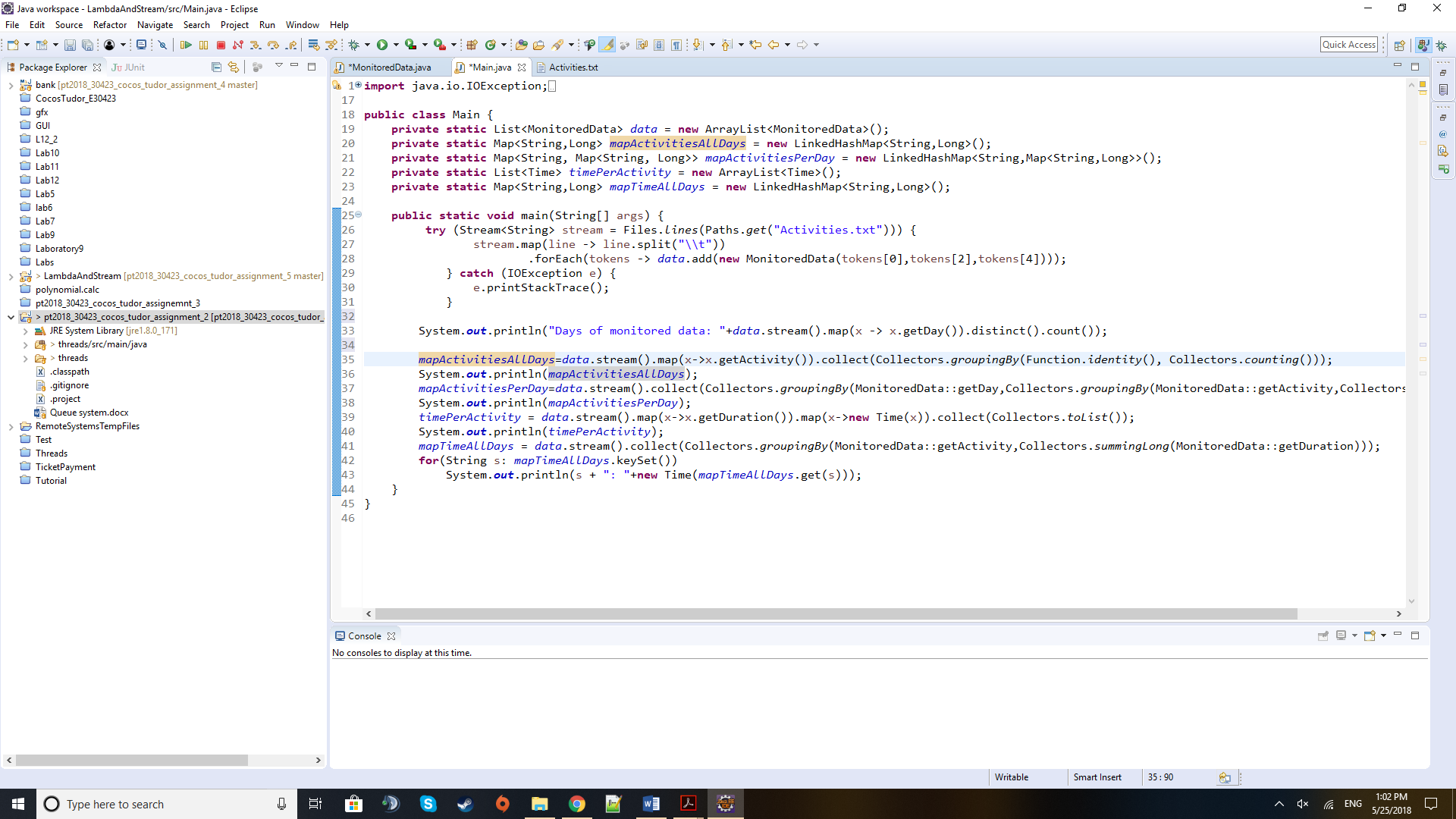
**4) Implementation**

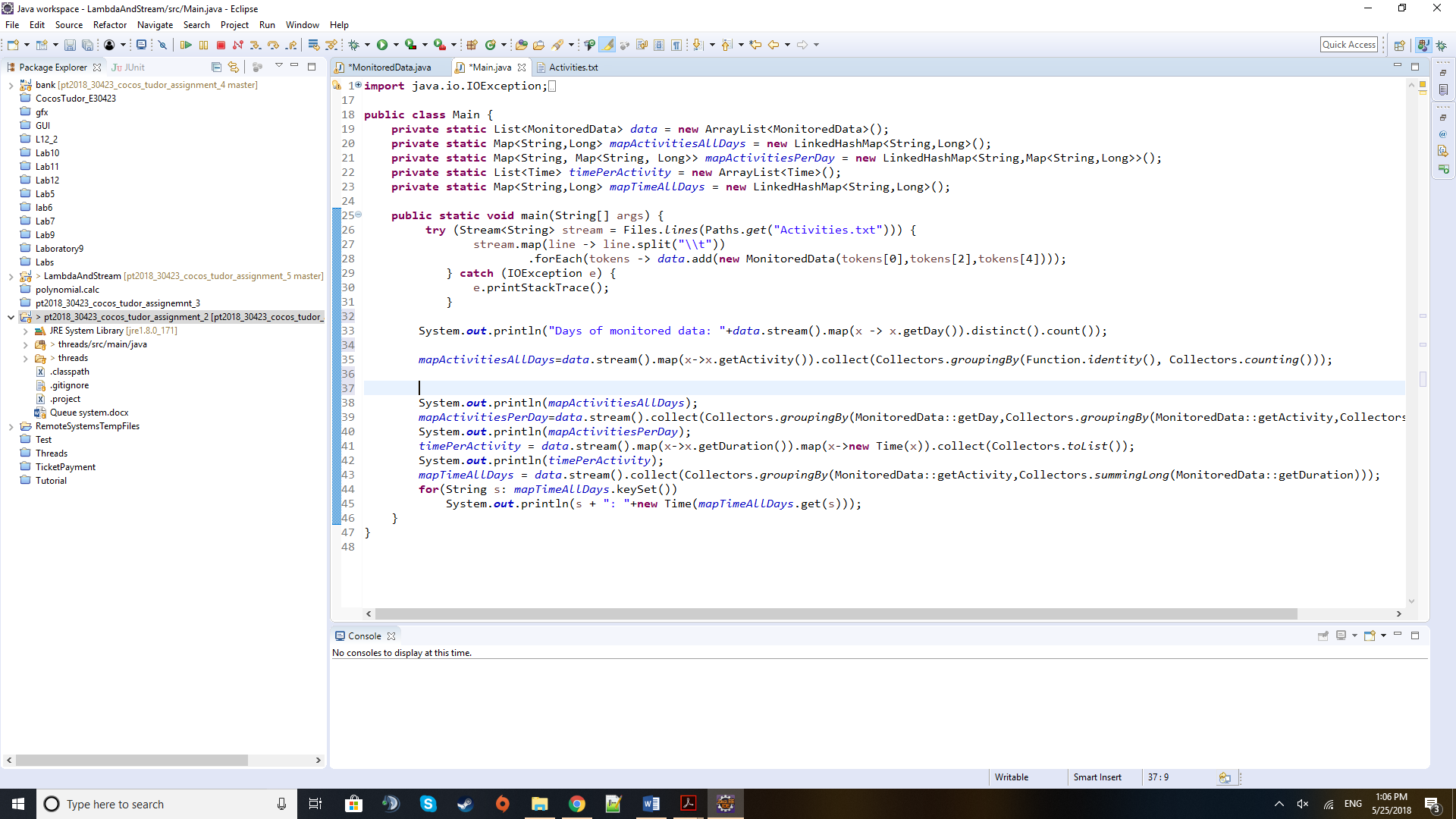
 I will describe here the most relevant pieces of code and especially those that are not that common and straight-forward (unlike getters, setters, and some basic constructors).

The method above may be simple, but it saved a lot of trouble when using the mapping operation or the methods of the Collectors. It takes the string that is a combination of date and time of type year month day and hours minutes seconds and splits it by the blank space in the middle to get the date defined by only the day, month and year.

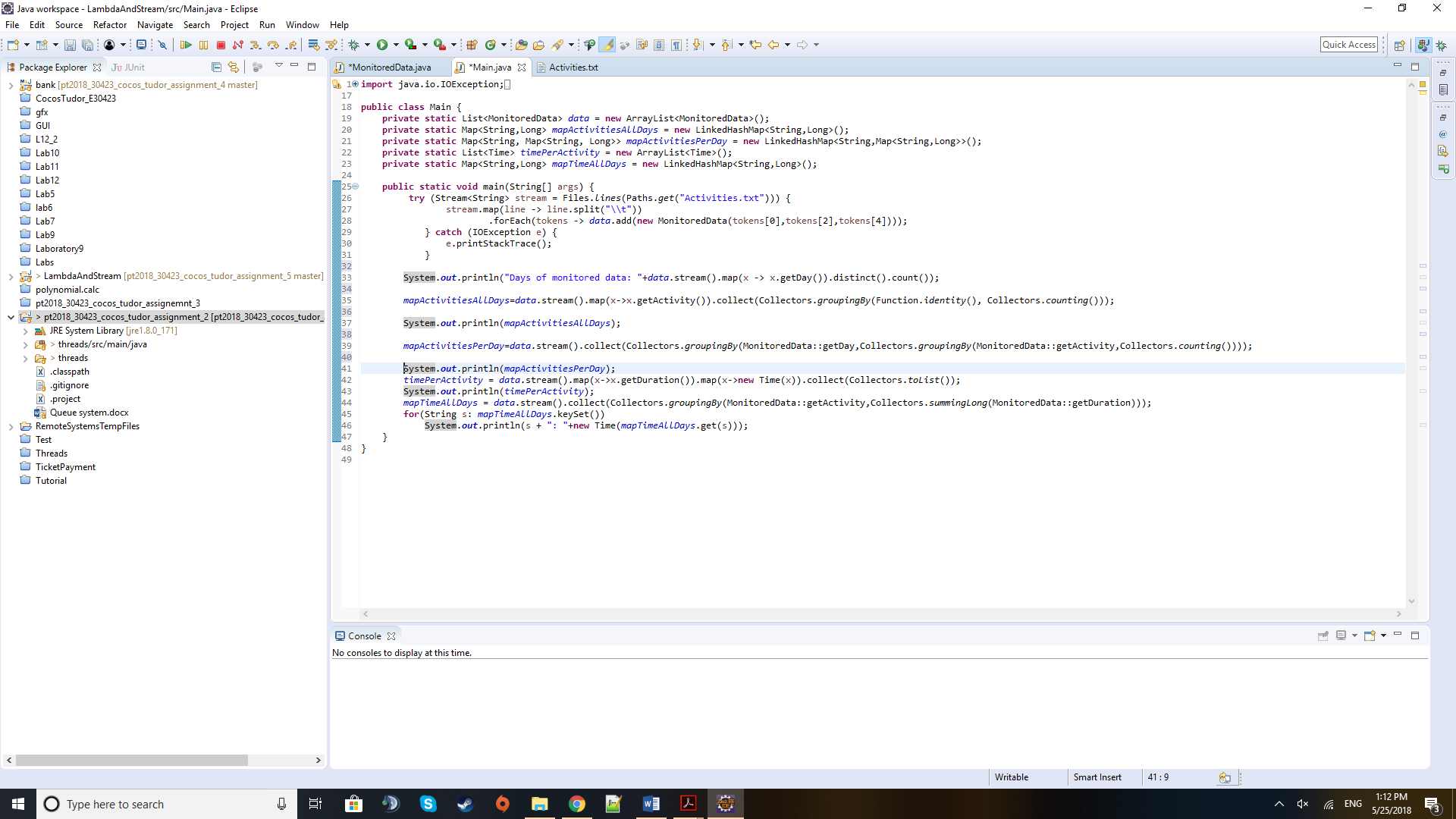


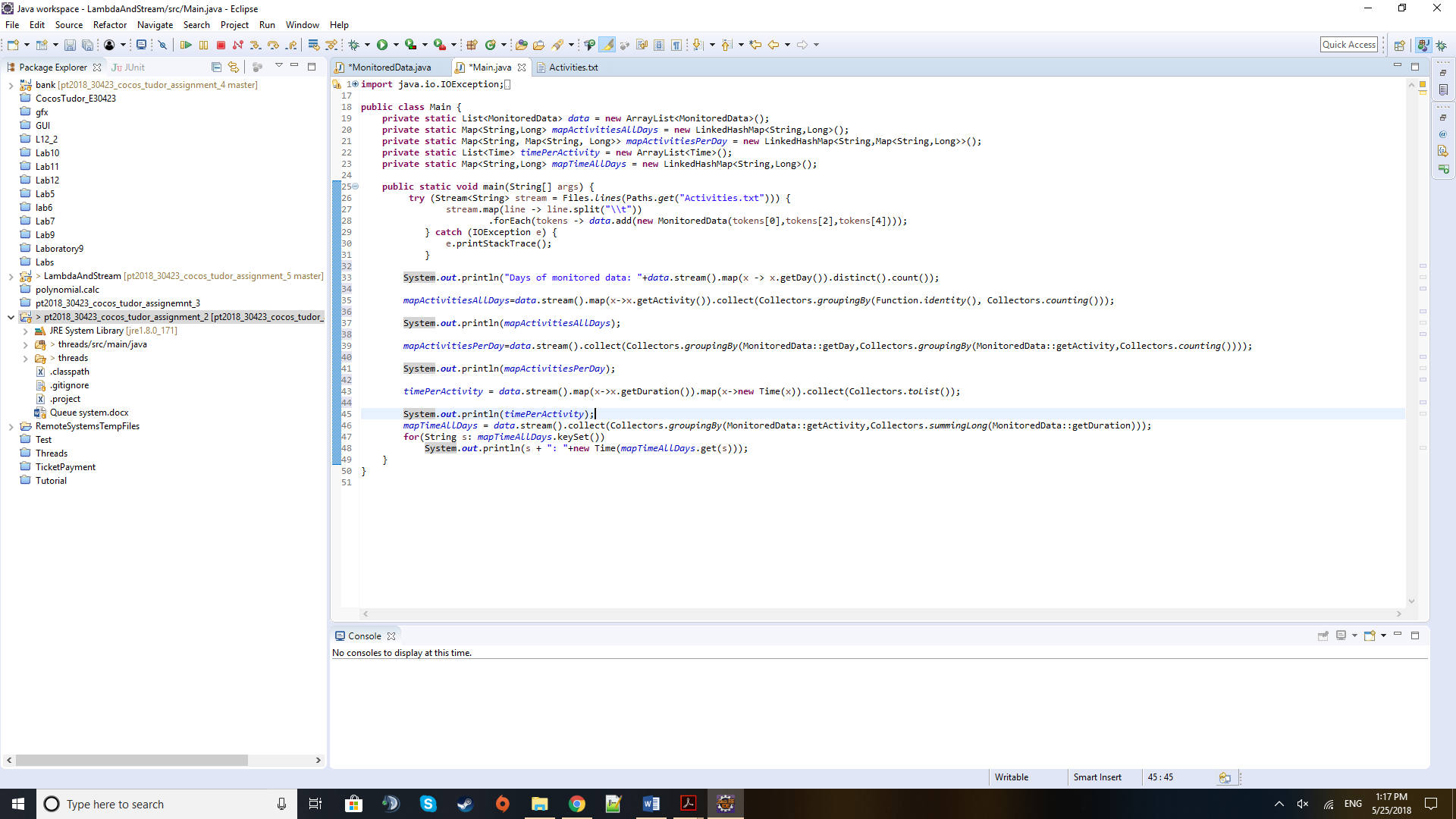
The method above returns in a long format the value of the duration of the activity stored in the given data file. It basically takes the values of the hours and minutes and seconds from the string fields of start and end time and makes three new values by subtracting the end values from the start values. These are later used to construct a Time entity that will provide the long value of that time interval translated into long. To get the appropriate strings to be parsed, we need to split both the start and end times, firstly, by taking the time part of the date and time combination and then splitting the time string by the full column character in 3 parts: hours, minutes and seconds.

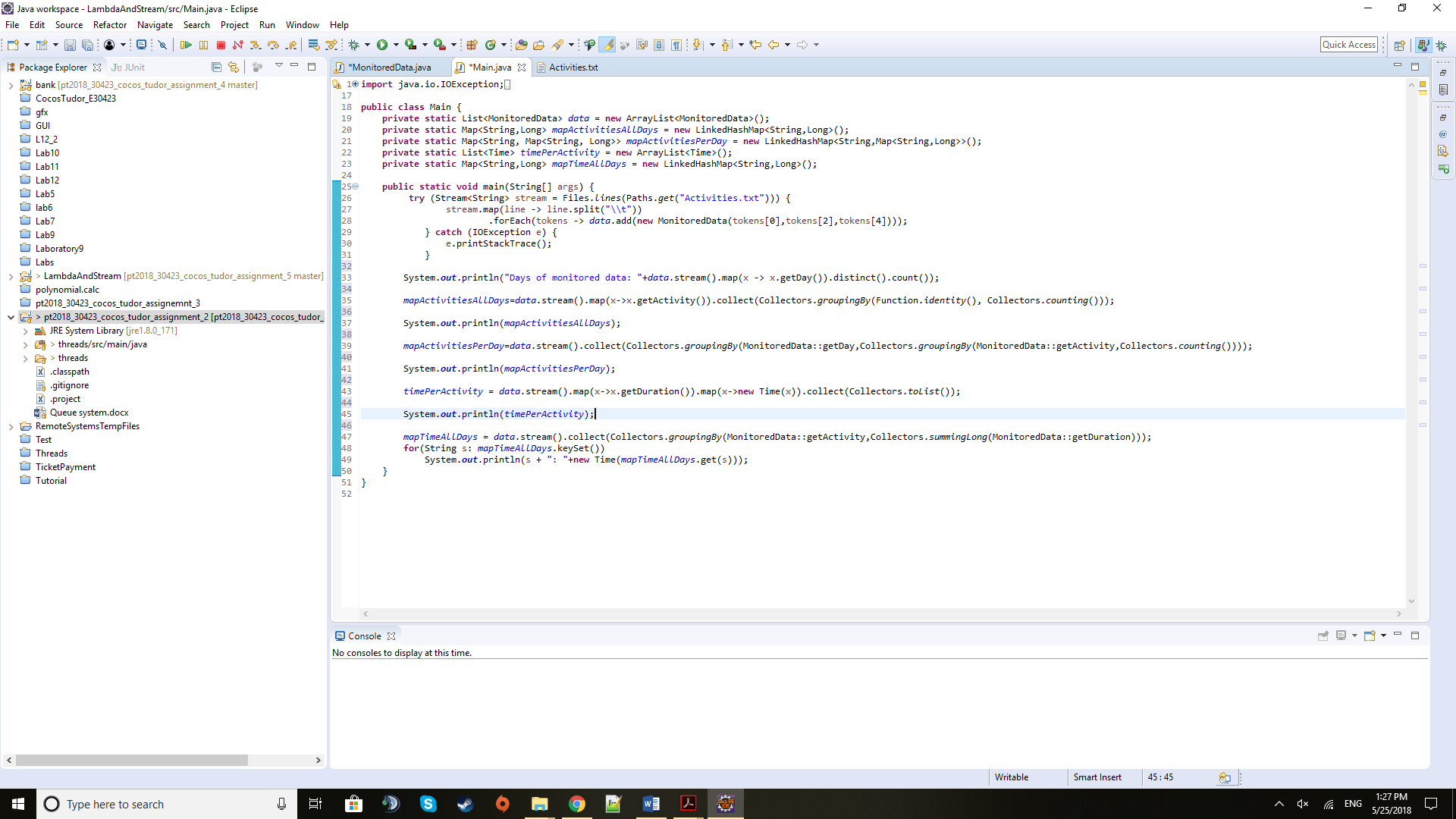
 The inspiration for the method of reading from a file came from a code snippet I saw online. After the file is opened and a stream of objects, that represent the lines of the text file, is started, we split the whole line by the tab character into 5 tokens: start time, a NULL value (the fields in the given text file are divided by a space and tab so the space between the tab and space is again found and forms an empty String), the end time, NULL values and finally, the activity label. That is why we build a new instance of MonitoredData with the splitted values at 0, 2 and 4, ignoring the NULL Strings.

 This line takes the stream of MonitoredData objects and transforms them into their respective day in order to be later able to count the distinct days that show up, thus giving us the result for the task of telling how many days have been recorded.

The code snippet above details how the map of activity labels and their count is made by the means of grouping by method of the Collectors class.



 To get the map of type Map<String, Map<String, Long>> we need to use the grouping by method two times. Firstly, we need to group the data by the day and then form the next map by the activity and its count only for that day.

To get the list of the time intervals of all the activities we use the to List method of the Collectors class.

Here we collect into a Map defined by String and Long, the name of the activity and a sum of all the time intervals by using the summing Long method. To display the information from the map nicely we need to transform the time value in long to a Time format by constructing a Time entity.

**5) Conclusions**

This assignment proved extremely useful because it introduced new concepts that are widely in use and that bring a lot of advantages especially when it comes to parallel processing via multicore processors. To the advantage of streams, come, of course, the lambda expressions that are handy in performing certain tasks in a neat and discrete way, being written in a single line of code.

Further development of this application may be to use this in parsing different types of filed, not a predefined one with a fixed structure.

**6) Bibliography**

Tutorials from: <http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

<https://caveofprogramming.teachable.com/p/java8>

<http://www.mkyong.com/tutorials/java-8-tutorials/>

For any issues regarding exceptions: <https://stackoverflow.com/>

For details about the methods and fields of predefined classes: <https://docs.oracle.com/javase/8/docs/api/overview-summary.html>

For the UML diagram I used an application online: <https://creately.com/>