# Assignment 5 Documentation

# Working with streams and lambda expressions

Dreghici Popa Vlad

Group 30422

1. *Problem description*

Consider the task of analyzing the behavior of a person recorded by a set of sensors. The historical log of the person’s activity is stored as tuples (start\_time, end\_time, activity\_label), where start\_time and end\_time 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 data is spread over several days as many entries in the log Activities.txt, taken from [1,2] and downloadable from the file Activities.txt located in this folder. Write a Java 1.8 program using lambda expressions and stream processing to do the tasks defined below.

1. *Problem analysis*

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions take advantage of parallel process capabilities of multi-core environments as seen with the support of pipeline operations on data in the Stream API.

Lambda expressions introduce the new arrow operator -> into Java. It divides the lambda expressions in two parts The left side specifies the parameters required by the expression, which could also be empty if no parameters are required. The right side is the lambda body which specifies the actions of the lambda expression. It might be helpful to think about this operator as “becomes”. For example, “n becomes n\*n”, or “n becomes n squared”. With functional interface and arrow operator concepts in mind, you can put together a simple lambda expression. They are anonymous methods (methods without names) used to implement a method defined by a functional interface. It’s important know what a functional interface is before getting your hands dirty with lambda expressions. Lambda expressions enable you to do this, to treat functionality as method argument, or code as data. The previous section, Anonymous Classes, shows you how to implement a base class without giving it a name. Although this is often more concise than a named class, for classes with only one method, even an anonymous class seems a bit excessive and cumbersome. Lambda expressions let you express instances of single-method classes more compactly.

Using lambda expression, you can refer to any final variable or effectively final variable (which is assigned only once). Lambda expression throws a compilation error, if a variable is assigned a value the second time.

Stream is a new abstract layer introduced in Java 8. Using stream, you can process data in a declarative way similar to SQL statements. Stream represents a sequence of objects from a source, which supports aggregate operations. The following are characteristics of a stream:

-Sequene of elements:  A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.

Source − Stream takes Collections, Arrays, or I/O resources as input source.

Pipelining − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.

Automatic iterations − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

There are two methods to generate a stream:

Stream() – returns a sequential stream considering collection as its source.

parallelStream() – returns a parallel stream considering collection as its source.

Streams provide a number of methods for easier accessing data from them:

forEach : Stream has provided a new method ‘forEach’ to iterate each element of the stream. The following code segment shows how to print 10 random numbers using forEach.

Map: The ‘map’ method is used to map each element to its corresponding result. The following code segment prints unique squares of numbers using map.

Filter: The ‘filter’ method is used to eliminate elements based on a criteria. The following code segment prints a count of empty strings using filter.

Collectors: Collectors are used to combine the result of processing on the elements of a stream. Collectors can be used to return a list or a string.

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. In the above example filter, map and sorted are intermediate operations whereas forEach is a terminal operation. For a full list of all available stream operations see the [Stream Javadoc](http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html). Such a chain of stream operations as seen in the example above is also known as operation pipeline.

Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both non-interfering and stateless. What does that mean?

See all data

Count days

Count activities

Count each day



Get duration



Filter



Run application

Go back

Review Result

User

Select option

1. *Design*

*Classes*

The classes is a blueprint that describes the behavior of the object it supports. The classes are organized in packages, which basically are a collection of classes, interfaces. In our case, the classes are organized in a MVC manner.

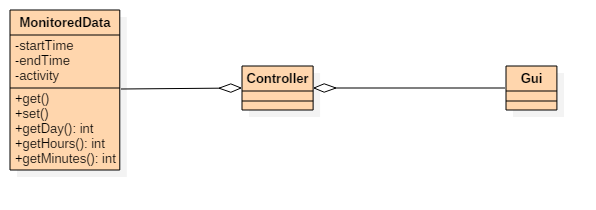
An object instantiated from a class has states and behaviors declared in the respective class.

The project will be split in a certain manner. There will be 3 packages: Model, View (gui classes) and finally Controller which controls all the classes that execute operations on the file we access. The model package will contain all the back end processes, like the algorithms, person, accounts. The view package will contain the front end classes. In this case we have one class: gui, where we have the main options to do. This is the visualization of our application.

**Controller class:** The controller class is exactly what the name says, a controller. It will implement the control of our application. The controller will also link the presentation part with the model data. First of all it will connect to the file. It will execute certain reads one a particular button is pressed. For example the first button in the gui represents the show all option. When this button is pressed there will be all the logs of the person from the file displayed inside the textArea. Once this button is pressed the controller will make the link between the file and the JTextArea, perform the requested action and insert the data in the table. This is the procedure for most of the algorithms implemented.

**GUI Class:** is the main gui class we see when we start the app. It will display the JTextArea with all the data from the file. You will be able to select all the options requested from this window and after pressing the corresponding button you will see the result inside the JTextArea.

**Monitored Data:** has 3 fields: start time, end time and activity as string. Read the data from the file Activity.txt using streams and create a list of objects of type MonitoredData. It implements a few methods in order to help us write the algorithms like: getDay which will return the day, getHours, returns total hours and getMinutes, returns the minutes of a activity.



1. *Implementation*

MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

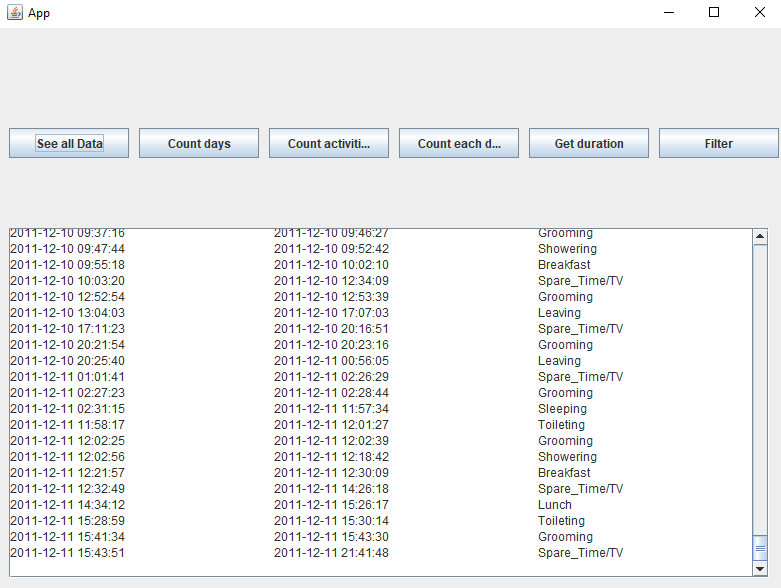
* **Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
* **View** - View represents the visualization of the data that model contains.
* **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.
  1. *Monitored Data class*

It has 3 fields: start time, end time and activity as string. Read the data from the file Activity.txt using streams and create a list of objects of type MonitoredData. It implements a few methods in order to help us write the algorithms like: getDay which will return the day, getHours, returns total hours and getMinutes, returns the minutes of a activity.

* 1. *Controller Class*

The controller class is exactly what the name says, a controller. It will implement the control of our application. The controller will also link the presentation part with the model data. First of all it will connect to the file. It will execute certain reads one a particular button is pressed. For example the first button in the gui represents the show all option. When this button is pressed there will be all the logs of the person from the file displayed inside the textArea. Once this button is pressed the controller will make the link between the file and the JTextArea, perform the requested action and insert the data in the table. This is the procedure for most of the algorithms implemented. The mostly used functions for streams in the ontroller are the following:

* Filter: returns a stream consisting of the elements of this stream that match the specified predicate.
* Map: returns a stream consisting of the result of applying the specified function to the elements of this stream. Performs one to one mapping.
* forEach: applies an action for each elements in the stream.
* Distinct: returns a stream consisting of the distinc elements of this stream. Elements are considered equal if e1.equals(e2) returns truel.
  1. *Gui Class*

**

The gui classes are the front end of our applications. They will display all the data taken from the tables and will make the interface user friendly. The guis is only one window. It basically has 7 buttons, each button executes a requested function and after the function is executed it will display the result inside the JTextArea below them. It has been built by extending the class JFrame and after that adding certain attributes in the controller, building it from scratch withous using the Window builder feature from eclipse. It contains a couple of functions for the JTextArea which helps us to easily append or set text of the JText Area. All of this are used inside the controller class, which basically controls all our application.

1. *Testing*

The testing part will be done by repeteadly using the app and testing all of the algorithms for each button repeatedly.

There is no point in using a JUnit testing unit because there are no ways to hardcode values. All the input we get is from the activities.txt file which we are given.

1. *Results*

During the usage of the application the user will be able to execute certain operations file given. First of them is to show all the data inside the file, second is to count the total number of days for which the activities of the person have been logged. Third is to count the total number of the activities, fourth is to count the same activities but per day. Fifth is to the the duration of the activities that last more than 10 hours over the log period and the last one is to filter the activities which 90% of the time were not longer than 5 minutes.

1. *Conclusions*

By working on this assignment I acquired more knowledge about how to work with lambda expressions. Another thing I learned is how to use the stream and work around with, even more how to successfully work with the functions provided and obtain result much more easier just by using a number of rows less than I used until now.

As a future feature of this application I thought of implementing a way to be more picky with results you want to display. For example filter over more options like name or appearences and so on. Over 50 % or less than 40% and so on. Those would give the user much more freedom in choosing which results he wants to see.

1. *Bibliography*

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

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

<http://www.tutorialspoint.com/java/java_serialization.htm>

<http://stackoverflow.com/>

<https://tutorialspoint.com/>

<https://www.javatpoint.com/>

<https://geeksforgeeks.com/>