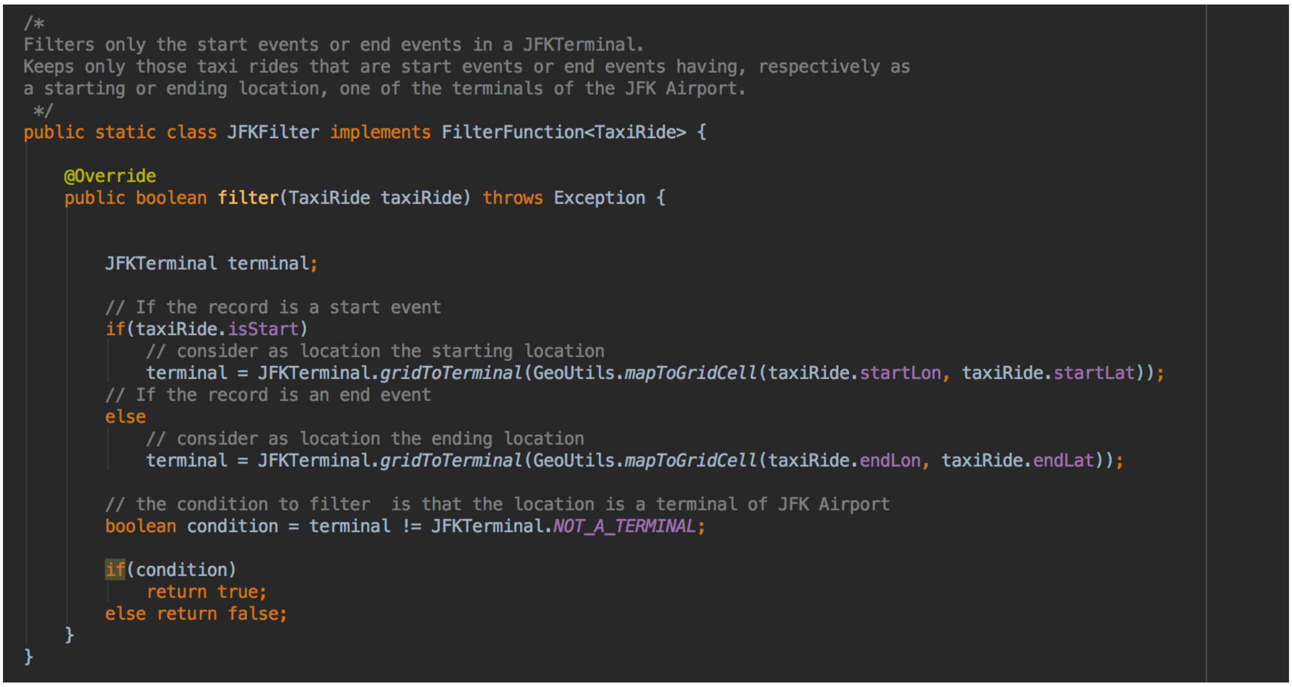
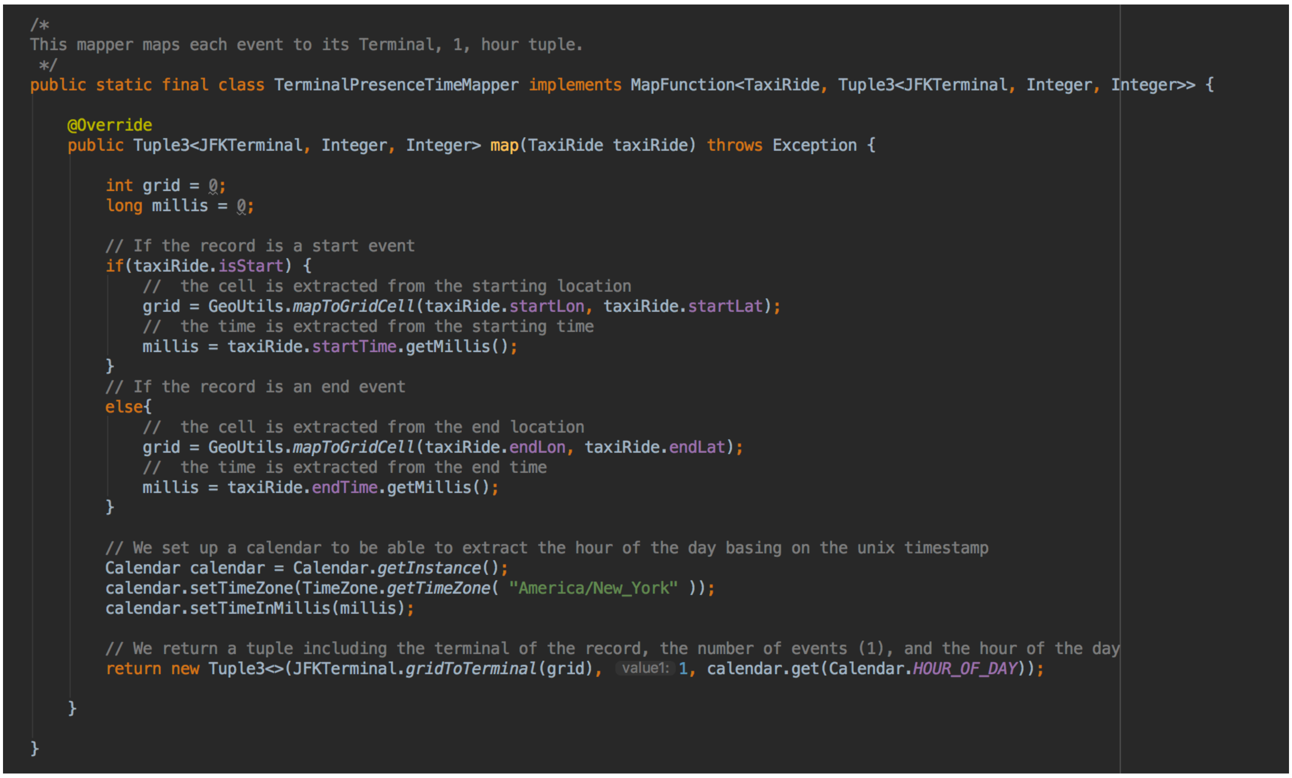
Lab Assignment 3

Author: Vendramin Nicolò

In order to solve the problem presented in task one I had to develop a filter function and a map function.
The filter function is required to only keep in the streams those ride events that are starting (or ending, depending wether it is a start or end event) in one of the terminals of the JFK airport of New York.
The mapper instead is used to put the stream from a sequence of TaxiRides to a sequences of tuples showing <Terminal_String, Cardinality, Hour of the day>. Below I attach the code of the two classes:

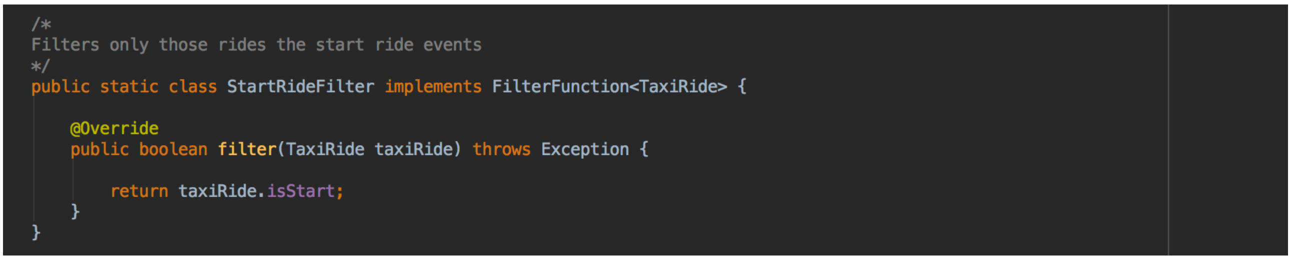


```
/*
Filters only the start events or end events in a JFKTerminal.
Keeps only those taxi rides that are start events or end events having, respectively as
a starting or ending location, one of the terminals of the JFK Airport.
 */
public static class JFKFilter implements FilterFunction<TaxiRide> {

    @Override
    public boolean filter(TaxiRide taxiRide) throws Exception {


        JFKTerminal terminal;

        // If the record is a start event
        if(taxiRide.isStart)
            // consider as location the starting location
            terminal = JFKTerminal.gridToTerminal(GeoUtils.mapToGridCell(taxiRide.startLon, taxiRide.startLat));
        // If the record is an end event
        else
            // consider as location the ending location
            terminal = JFKTerminal.gridToTerminal(GeoUtils.mapToGridCell(taxiRide.endLon, taxiRide.endLat));

        // the condition to filter  is that the location is a terminal of JFK Airport
        boolean condition = terminal != JFKTerminal.NOT_A_TERMINAL;

        if(condition)
            return true;
        else return false;
    }
}
```



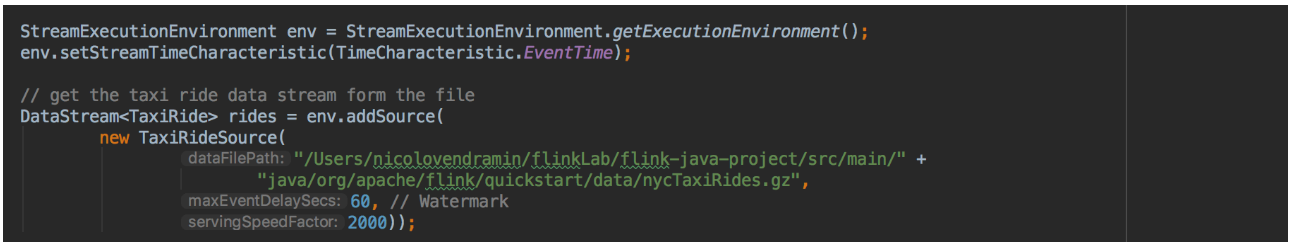
```
/*
This mapper maps each event to its Terminal, 1, hour tuple.
 */
public static final class TerminalPresenceTimeMapper implements MapFunction<TaxiRide, Tuple3<JFKTerminal, Integer, Integer>> {

    @Override
    public Tuple3<JFKTerminal, Integer, Integer> map(TaxiRide taxiRide) throws Exception {

        int grid = 0;
        long millis = 0;

        // If the record is a start event
        if(taxiRide.isStart) {
            //  the cell is extracted from the starting location
            grid = GeoUtils.mapToGridCell(taxiRide.startLon, taxiRide.startLat);
            //  the time is extracted from the starting time
            millis = taxiRide.startTime.getMillis();
        }
        // If the record is an end event
        else{
            //  the cell is extracted from the end location
            grid = GeoUtils.mapToGridCell(taxiRide.endLon, taxiRide.endLat);
            //  the time is extracted from the end time
            millis = taxiRide.endTime.getMillis();
        }

        // We set up a calendar to be able to extract the hour of the day basing on the unix timestamp
        Calendar calendar = Calendar.getInstance();
        calendar.setTimeZone(TimeZone.getTimeZone( "America/New_York" ));
        calendar.setTimeInMillis(millis);

        // We return a tuple including the terminal of the record, the number of events (1), and the hour of the day
        return new Tuple3<>(JFKTerminal.gridToTerminal(grid),  value1: 1, calendar.get(Calendar.HOUR_OF_DAY));

    }

}
```

For what the second task is concerned I was having two possible interpretations. For one of them I needed to implement a second filter that only keeps those ride records corresponding to start event. Below I attach the code:



```
/*
Filters only those rides the start ride events
*/
public static class StartRideFilter implements FilterFunction<TaxiRide> {

    @Override
    public boolean filter(TaxiRide taxiRide) throws Exception {

        return taxiRide.isStart;
    }
}
```

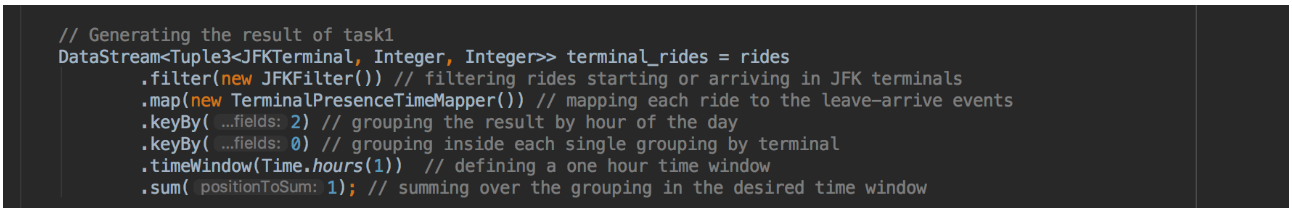
In the main() I proceed as follow:
At first the execution environment is set up, and the file containing the data is added as a source to the execution environment.
The watermark is set to 60 seconds, considering possible disconnections or delay in the communication of the events from the car to the system storing the data.



```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

// get the taxi ride data stream form the file
DataStream<TaxiRide> rides = env.addSource(
        new TaxiRideSource(
                dataFilePath: "/Users/nicolovendramin/flinkLab/flink-java-project/src/main/" +
                        "java/org/apache/flink/quickstart/data/nycTaxiRides.gz",
                maxEventDelaySecs: 60, // Watermark
                servingSpeedFactor: 2000));
```

After initialising the data stream we can start applying the transformations for each task.

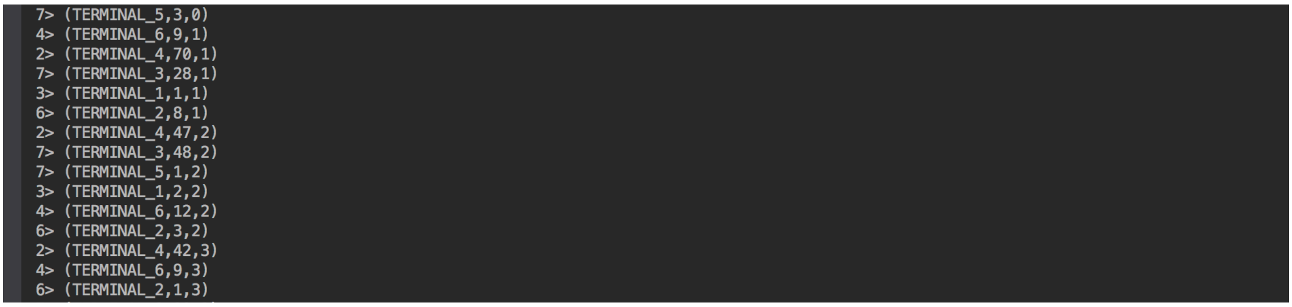
Task1: For each hour progressively, for each terminal count the number of events (both start and end events).



```
    // Generating the result of task1
    DataStream<Tuple3<JFKTerminal, Integer, Integer>> terminal_rides = rides
            .filter(new JFKFilter()) // filtering rides starting or arriving in JFK terminals
            .map(new TerminalPresenceTimeMapper()) // mapping each ride to the leave-arrive events
            .keyBy( ...fields: 2) // grouping the result by hour of the day
            .keyBy( ...fields: 0) // grouping inside each single grouping by terminal
            .timeWindow(Time.hours(1))  // defining a one hour time window
            .sum( positionToSum: 1); // summing over the grouping in the desired time window
```

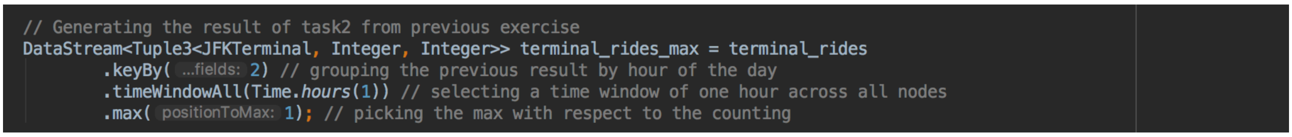
Simply we filter only the rides starting or ending at JFK airport, we map them to the <Terminal_String, Cardinality, Hour of the day> format using the map class reported above, we group them by hour of the day and by terminal (in this order), we define a timeWindow of 1 hour and we perform a sum over the elements of the grouping in the required time window.

Output(Task1):



```
7> (TERMINAL_5,3,0)
4> (TERMINAL_6,9,1)
2> (TERMINAL_4,70,1)
7> (TERMINAL_3,28,1)
3> (TERMINAL_1,1,1)
6> (TERMINAL_2,8,1)
2> (TERMINAL_4,47,2)
7> (TERMINAL_3,48,2)
7> (TERMINAL_5,1,2)
3> (TERMINAL_1,2,2)
4> (TERMINAL_6,12,2)
6> (TERMINAL_2,3,2)
2> (TERMINAL_4,42,3)
4> (TERMINAL_6,9,3)
6> (TERMINAL_2,1,3)
```

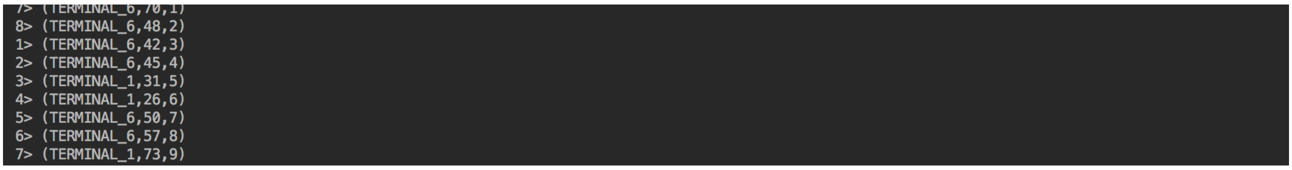
Task2(alpha): For each hour of the day we pick the busiest terminal with the number of events (both start and end events).



```
// Generating the result of task2 from previous exercise
DataStream<Tuple3<JFKTerminal, Integer, Integer>> terminal_rides_max = terminal_rides
        .keyBy( ...fields: 2) // grouping the previous result by hour of the day
        .timeWindowAll(Time.hours(1)) // selecting a time window of one hour across all nodes
        .max( positionToMax: 1); // picking the max with respect to the counting
```

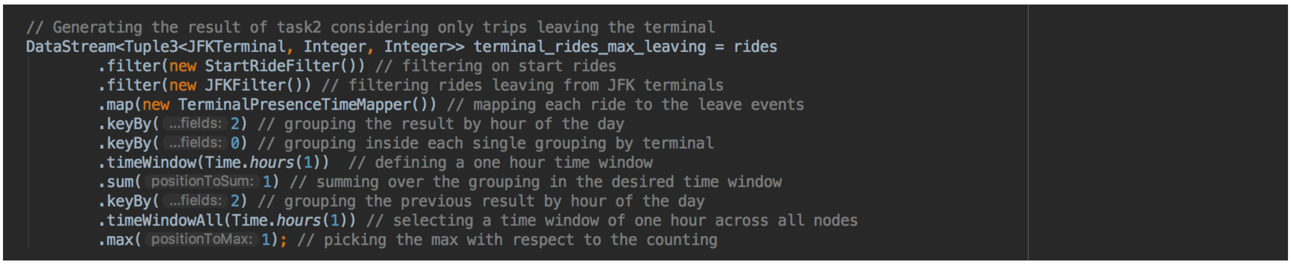
We just start from the previously obtained stream, we apply a further grouping by hour of the day, we set a time window across all the nodes and we extract the maximum over the different nodes.

Output(Task2(alpha)):



```
7> (TERMINAL_6,70,1)
8> (TERMINAL_6,48,2)
1> (TERMINAL_6,42,3)
2> (TERMINAL_6,45,4)
3> (TERMINAL_1,31,5)
4> (TERMINAL_1,26,6)
5> (TERMINAL_6,50,7)
6> (TERMINAL_6,57,8)
7> (TERMINAL_1,73,9)
```

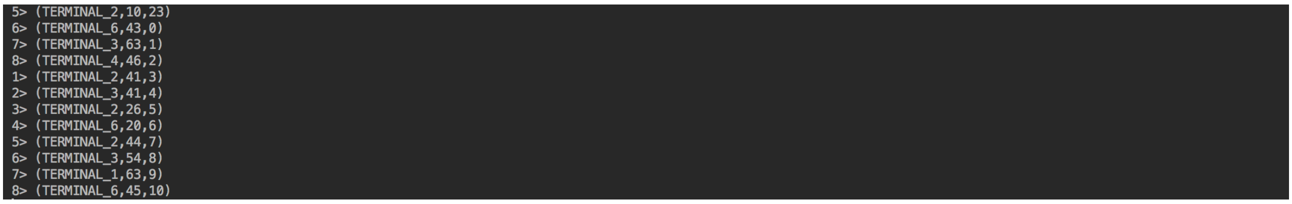
Task2(beta): For each hour of the day we pick the terminal from which more people has to leave (only start events).



```
// Generating the result of task2 considering only trips leaving the terminal
DataStream<Tuple3<JFKTerminal, Integer, Integer>> terminal_rides_max_leaving = rides
        .filter(new StartRideFilter()) // filtering on start rides
        .filter(new JFKFilter()) // filtering rides leaving from JFK terminals
        .map(new TerminalPresenceTimeMapper()) // mapping each ride to the leave events
        .keyBy( ...fields: 2) // grouping the result by hour of the day
        .keyBy( ...fields: 0) // grouping inside each single grouping by terminal
        .timeWindow(Time.hours(1))  // defining a one hour time window
        .sum( positionToSum: 1) // summing over the grouping in the desired time window
        .keyBy( ...fields: 2) // grouping the previous result by hour of the day
        .timeWindowAll(Time.hours(1)) // selecting a time window of one hour across all nodes
        .max( positionToMax: 1); // picking the max with respect to the counting
```

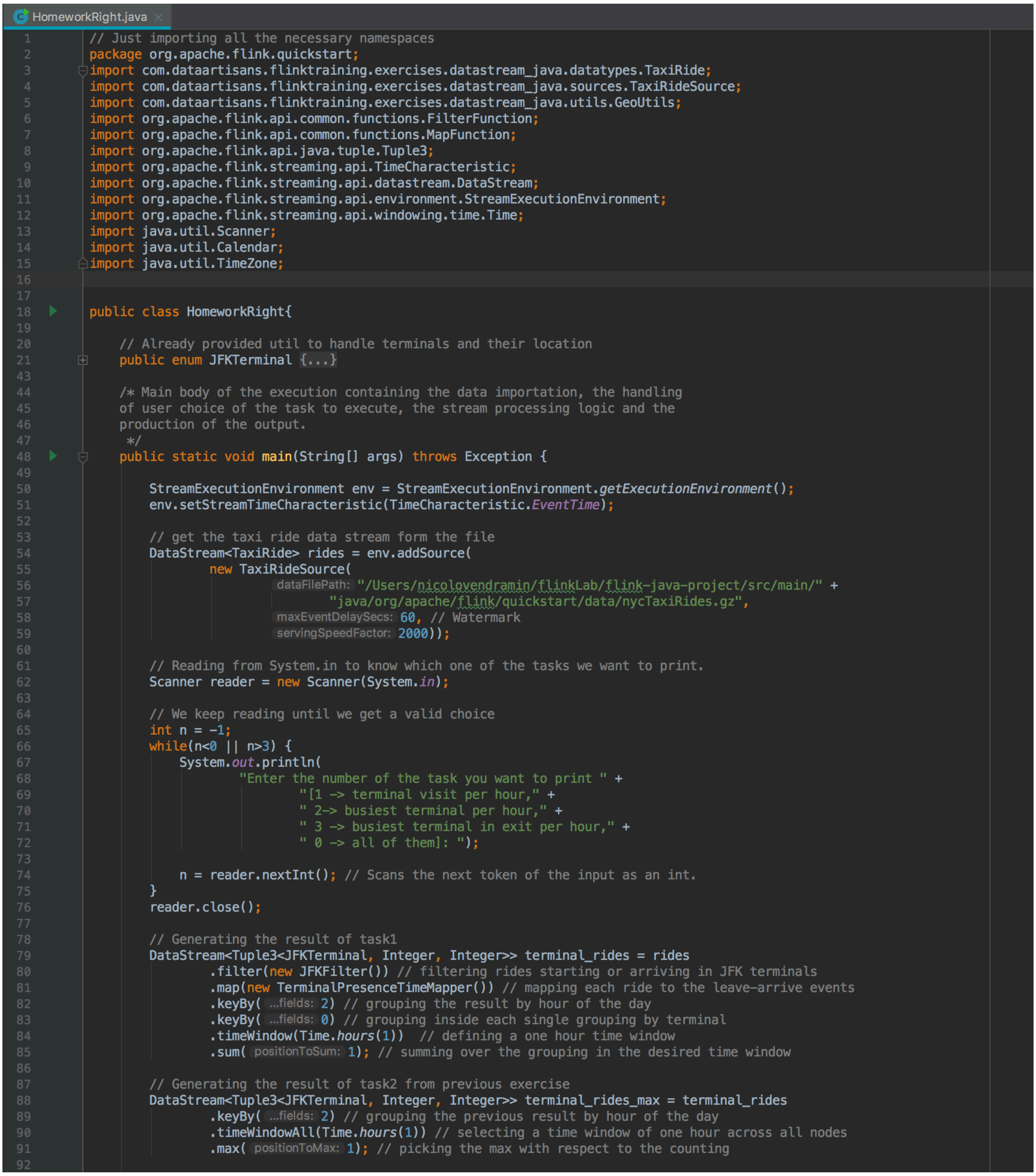
Here we have to restart from the clean. (imported from file) data stream. We apply a first filter to keep only the start event records. After that we perform all the operations that have been described for task 1 and task2(alpha) in the same order.

Output(Task2(beta)):

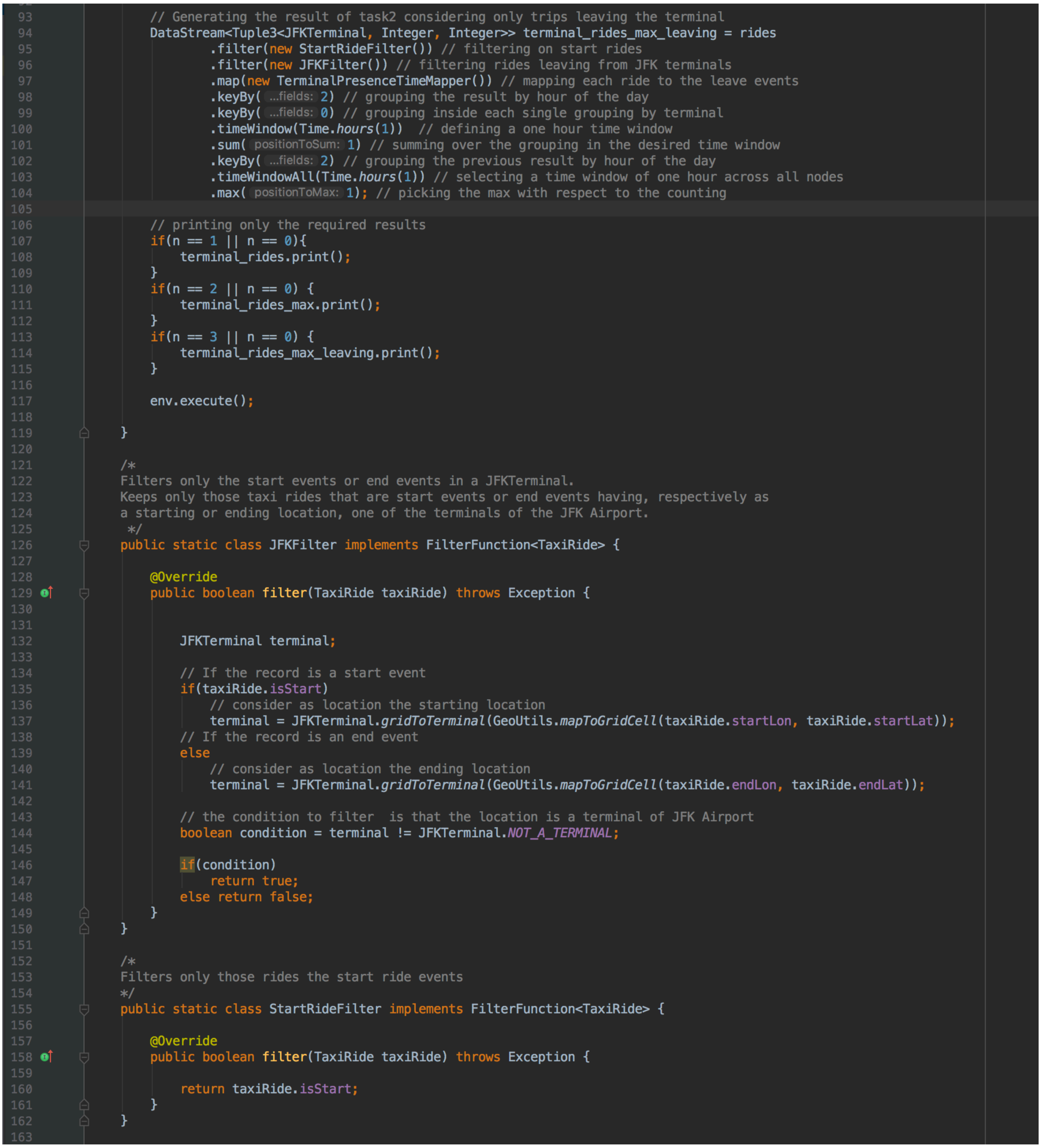


```
5> (TERMINAL_2,10,23)
6> (TERMINAL_6,43,0)
7> (TERMINAL_3,63,1)
8> (TERMINAL_4,46,2)
1> (TERMINAL_2,41,3)
2> (TERMINAL_3,41,4)
3> (TERMINAL_2,26,5)
4> (TERMINAL_6,20,6)
5> (TERMINAL_2,44,7)
6> (TERMINAL_3,54,8)
7> (TERMINAL_1,63,9)
8> (TERMINAL_6,45,10)
```

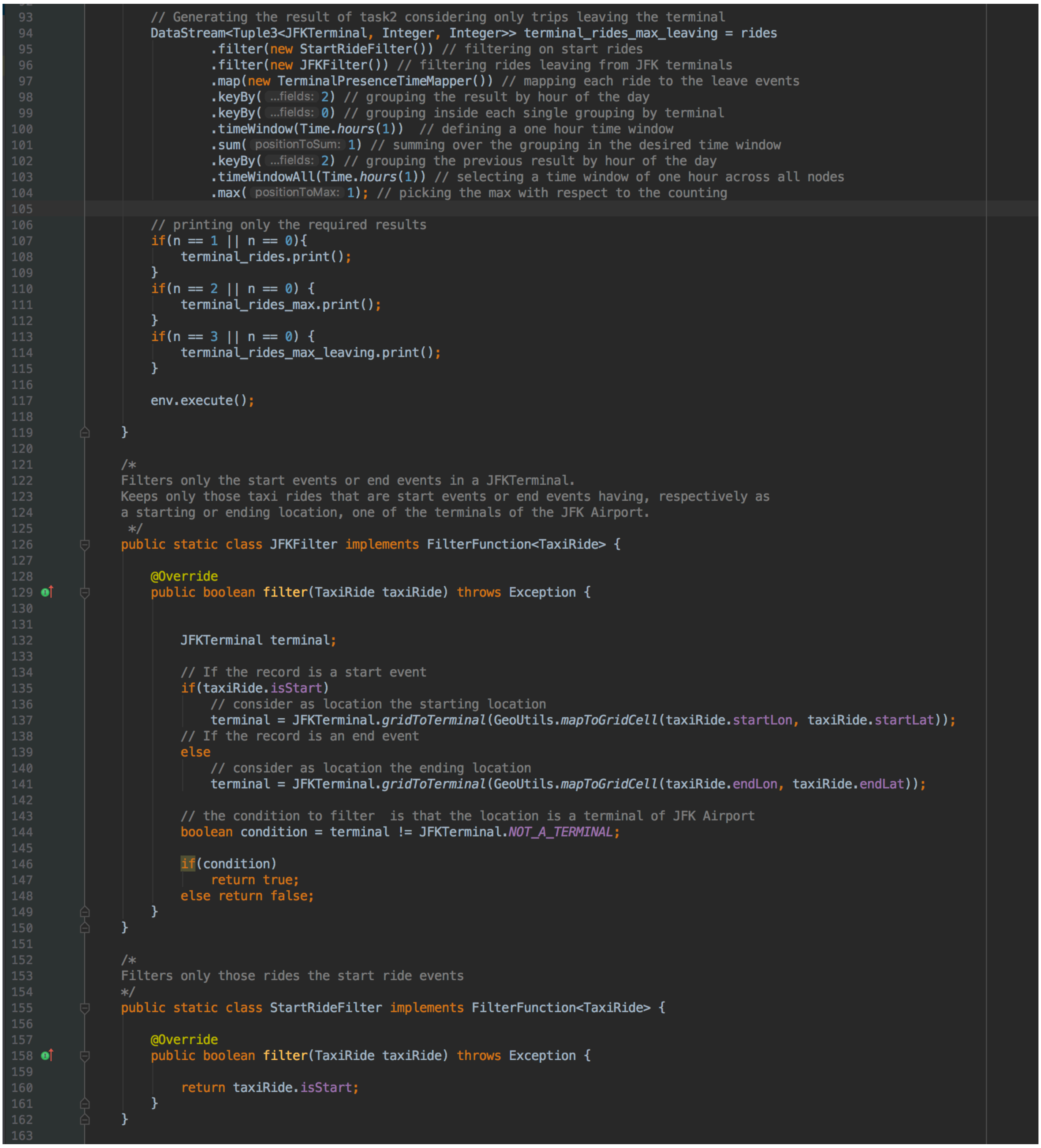
The full code is attached at the end of this document in case of any doubt about how the different pieces are plugged.



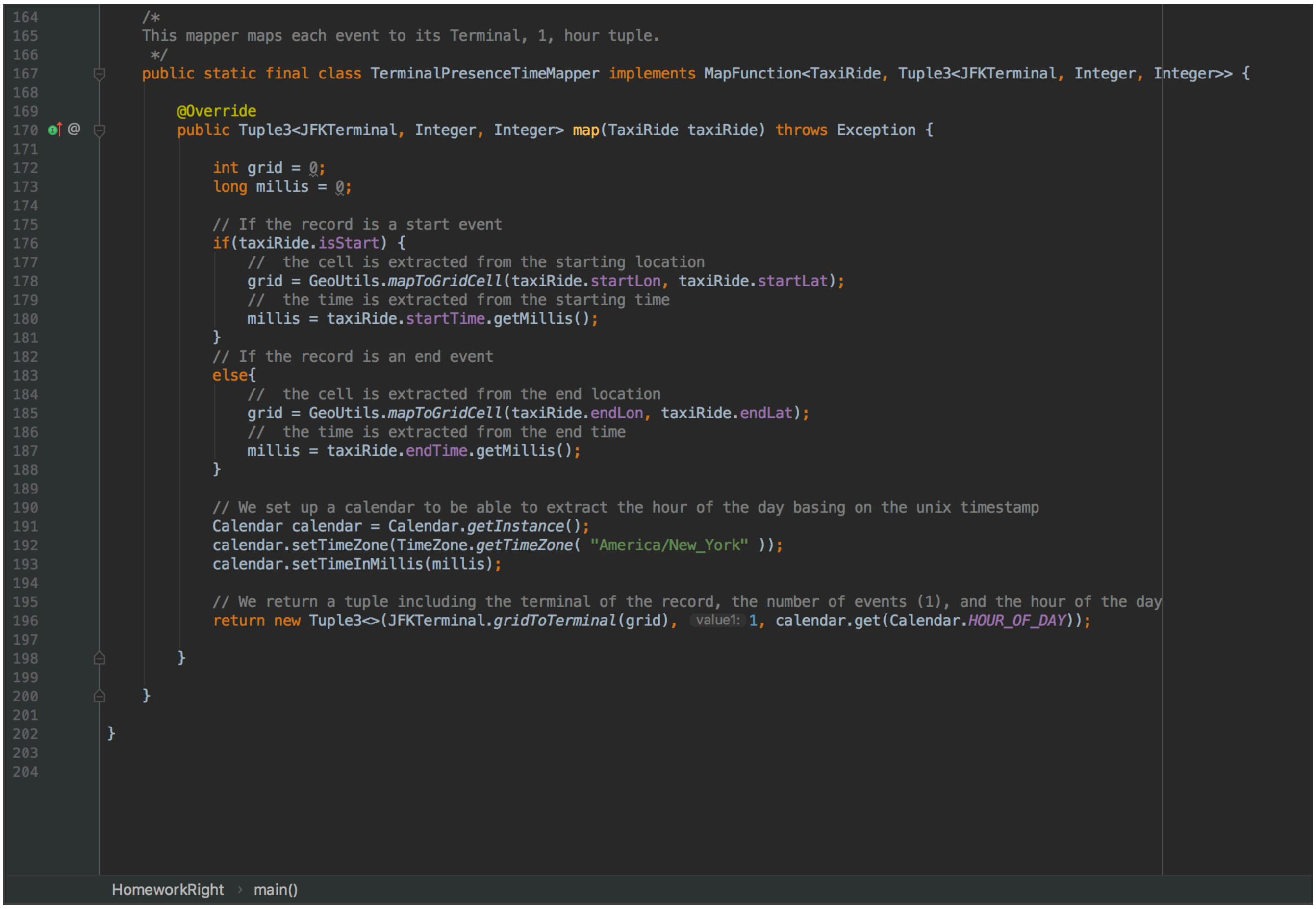
```java
// Just importing all the necessary namespaces
package org.apache.flink.quickstart;
import com.dataartisans.flinktraining.exercises.datastream_java.datatypes.TaxiRide;
import com.dataartisans.flinktraining.exercises.datastream_java.sources.TaxiRideSource;
import com.dataartisans.flinktraining.exercises.datastream_java.utils.GeoUtils;
import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.TimeCharacteristic;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import java.util.Scanner;
import java.util.Calendar;
import java.util.TimeZone;


public class HomeworkRight{

    // Already provided util to handle terminals and their location
    public enum JFKTerminal {...}

    /* Main body of the execution containing the data importation, the handling
    of user choice of the task to execute, the stream processing logic and the
    production of the output.
     */
    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        // get the taxi ride data stream form the file
        DataStream<TaxiRide> rides = env.addSource(
                new TaxiRideSource(
                        dataFilePath: "/Users/nicolovendramin/flinkLab/flink-java-project/src/main/" +
                                "java/org/apache/flink/quickstart/data/nycTaxiRides.gz",
                        maxEventDelaySecs: 60, // Watermark
                        servingSpeedFactor: 2000));

        // Reading from System.in to know which one of the tasks we want to print.
        Scanner reader = new Scanner(System.in);

        // We keep reading until we get a valid choice
        int n = -1;
        while(n<0 || n>3) {
            System.out.println(
                    "Enter the number of the task you want to print " +
                            "[1 -> terminal visit per hour," +
                            " 2-> busiest terminal per hour," +
                            " 3 -> busiest terminal in exit per hour," +
                            " 0 -> all of them]: ");

            n = reader.nextInt(); // Scans the next token of the input as an int.
        }
        reader.close();

        // Generating the result of task1
        DataStream<Tuple3<JFKTerminal, Integer, Integer>> terminal_rides = rides
                .filter(new JFKFilter()) // filtering rides starting or arriving in JFK terminals
                .map(new TerminalPresenceTimeMapper()) // mapping each ride to the leave-arrive events
                .keyBy( ...fields: 2) // grouping the result by hour of the day
                .keyBy( ...fields: 0) // grouping inside each single grouping by terminal
                .timeWindow(Time.hours(1))  // defining a one hour time window
                .sum( positionToSum: 1); // summing over the grouping in the desired time window

        // Generating the result of task2 from previous exercise
        DataStream<Tuple3<JFKTerminal, Integer, Integer>> terminal_rides_max = terminal_rides
                .keyBy( ...fields: 2) // grouping the previous result by hour of the day
                .timeWindowAll(Time.hours(1)) // selecting a time window of one hour across all nodes
                .max( positionToMax: 1); // picking the max with respect to the counting

```



```java
        // Generating the result of task2 considering only trips leaving the terminal
        DataStream<Tuple3<JFKTerminal, Integer, Integer>> terminal_rides_max_leaving = rides
                .filter(new StartRideFilter()) // filtering on start rides
                .filter(new JFKFilter()) // filtering rides leaving from JFK terminals
                .map(new TerminalPresenceTimeMapper()) // mapping each ride to the leave events
                .keyBy( ...fields: 2) // grouping the result by hour of the day
                .keyBy( ...fields: 0) // grouping inside each single grouping by terminal
                .timeWindow(Time.hours(1))  // defining a one hour time window
                .sum( positionToSum: 1) // summing over the grouping in the desired time window
                .keyBy( ...fields: 2) // grouping the previous result by hour of the day
                .timeWindowAll(Time.hours(1)) // selecting a time window of one hour across all nodes
                .max( positionToMax: 1); // picking the max with respect to the counting

        // printing only the required results
        if(n == 1 || n == 0){
            terminal_rides.print();
        }
        if(n == 2 || n == 0) {
            terminal_rides_max.print();
        }
        if(n == 3 || n == 0) {
            terminal_rides_max_leaving.print();
        }

        env.execute();

    }

    /*
    Filters only the start events or end events in a JFKTerminal.
    Keeps only those taxi rides that are start events or end events having, respectively as
    a starting or ending location, one of the terminals of the JFK Airport.
     */
    public static class JFKFilter implements FilterFunction<TaxiRide> {

        @Override
        public boolean filter(TaxiRide taxiRide) throws Exception {


            JFKTerminal terminal;

            // If the record is a start event
            if(taxiRide.isStart)
                // consider as location the starting location
                terminal = JFKTerminal.gridToTerminal(GeoUtils.mapToGridCell(taxiRide.startLon, taxiRide.startLat));
            // If the record is an end event
            else
                // consider as location the ending location
                terminal = JFKTerminal.gridToTerminal(GeoUtils.mapToGridCell(taxiRide.endLon, taxiRide.endLat));

            // the condition to filter  is that the location is a terminal of JFK Airport
            boolean condition = terminal != JFKTerminal.NOT_A_TERMINAL;

            if(condition)
                return true;
            else return false;
        }
    }

    /*
    Filters only those rides the start ride events
    */
    public static class StartRideFilter implements FilterFunction<TaxiRide> {

        @Override
        public boolean filter(TaxiRide taxiRide) throws Exception {

            return taxiRide.isStart;
        }
    }
```



```
        // Generating the result of task2 considering only trips leaving the terminal
        DataStream<Tuple3<JFKTerminal, Integer, Integer>> terminal_rides_max_leaving = rides
                .filter(new StartRideFilter()) // filtering on start rides
                .filter(new JFKFilter()) // filtering rides leaving from JFK terminals
                .map(new TerminalPresenceTimeMapper()) // mapping each ride to the leave events
                .keyBy( ...fields: 2) // grouping the result by hour of the day
                .keyBy( ...fields: 0) // grouping inside each single grouping by terminal
                .timeWindow(Time.hours(1))  // defining a one hour time window
                .sum( positionToSum: 1) // summing over the grouping in the desired time window
                .keyBy( ...fields: 2) // grouping the previous result by hour of the day
                .timeWindowAll(Time.hours(1)) // selecting a time window of one hour across all nodes
                .max( positionToMax: 1); // picking the max with respect to the counting

        // printing only the required results
        if(n == 1 || n == 0){
            terminal_rides.print();
        }
        if(n == 2 || n == 0) {
            terminal_rides_max.print();
        }
        if(n == 3 || n == 0) {
            terminal_rides_max_leaving.print();
        }

        env.execute();

    }

    /*
    Filters only the start events or end events in a JFKTerminal.
    Keeps only those taxi rides that are start events or end events having, respectively as
    a starting or ending location, one of the terminals of the JFK Airport.
     */
    public static class JFKFilter implements FilterFunction<TaxiRide> {

        @Override
        public boolean filter(TaxiRide taxiRide) throws Exception {


            JFKTerminal terminal;

            // If the record is a start event
            if(taxiRide.isStart)
                // consider as location the starting location
                terminal = JFKTerminal.gridToTerminal(GeoUtils.mapToGridCell(taxiRide.startLon, taxiRide.startLat));
            // If the record is an end event
            else
                // consider as location the ending location
                terminal = JFKTerminal.gridToTerminal(GeoUtils.mapToGridCell(taxiRide.endLon, taxiRide.endLat));

            // the condition to filter  is that the location is a terminal of JFK Airport
            boolean condition = terminal != JFKTerminal.NOT_A_TERMINAL;

            if(condition)
                return true;
            else return false;
        }
    }

    /*
    Filters only those rides the start ride events
    */
    public static class StartRideFilter implements FilterFunction<TaxiRide> {

        @Override
        public boolean filter(TaxiRide taxiRide) throws Exception {

            return taxiRide.isStart;
        }
    }
```



```java
    /*
    This mapper maps each event to its Terminal, 1, hour tuple.
     */
    public static final class TerminalPresenceTimeMapper implements MapFunction<TaxiRide, Tuple3<JFKTerminal, Integer, Integer>> {

        @Override
        public Tuple3<JFKTerminal, Integer, Integer> map(TaxiRide taxiRide) throws Exception {

            int grid = 0;
            long millis = 0;

            // If the record is a start event
            if(taxiRide.isStart) {
                //  the cell is extracted from the starting location
                grid = GeoUtils.mapToGridCell(taxiRide.startLon, taxiRide.startLat);
                //  the time is extracted from the starting time
                millis = taxiRide.startTime.getMillis();
            }
            // If the record is an end event
            else{
                //  the cell is extracted from the end location
                grid = GeoUtils.mapToGridCell(taxiRide.endLon, taxiRide.endLat);
                //  the time is extracted from the end time
                millis = taxiRide.endTime.getMillis();
            }

            // We set up a calendar to be able to extract the hour of the day basing on the unix timestamp
            Calendar calendar = Calendar.getInstance();
            calendar.setTimeZone(TimeZone.getTimeZone( "America/New_York" ));
            calendar.setTimeInMillis(millis);

            // We return a tuple including the terminal of the record, the number of events (1), and the hour of the day
            return new Tuple3<>(JFKTerminal.gridToTerminal(grid),  value1: 1, calendar.get(Calendar.HOUR_OF_DAY));

        }

    }

}
```