```
package it.polito.bigdata.spark.exercise2;

import org.apache.spark.api.java.*;

import org.apache.spark.SparkConf;

public class SparkDriver {


      public static void main(String[] args) {

            String inputPathPM10readings;
            String outputPathMonthlyStatistics;
            String outputPathCriticalPeriods;

            double PM10threshold;

            inputPathPM10readings=args[0];
            PM10threshold=Double.parseDouble(args[1]);
            outputPathMonthlyStatistics=args[2];
            outputPathCriticalPeriods=args[3];

            // Create a configuration object and set the name of the application
            SparkConf conf=new SparkConf().setAppName("Spark Exam 2 - Exercise
#2");

            // Create a Spark Context object
            JavaSparkContext sc = new JavaSparkContext(conf);

            // ****************************************
            // Exercise 2 - Part A
            // ****************************************
            // Compute, for each pair (sensorid, month),
            // the number of days with a critical PM10 value and store the result
in an HDFS folder.
            // Each line of the output file contains a
            // pair (month, number of critical days for that month).

            // Read the content of PM10Readings.txt
            JavaRDD<String> PM10readingsRDD = sc.textFile(inputPathPM10readings);

            // Select the critical readings/lines
            JavaRDD<String> criticalPM10readingsRDD = PM10readingsRDD.filter(new
CriticalPM10(PM10threshold)).cache();

            // Count the number of critical days (lines) for each pair
(sensorid,month)
            // Define a JavaPairRDD with sensorid_month as key and 1 as value
            JavaPairRDD<String,Integer>
sensorMonthdatesCriticalPM10=criticalPM10readingsRDD.mapToPair(new
SensorMonthOne());

            // Use reduce by key to compute the number of critical days for each
pair (sensorid,month)
            JavaPairRDD<String,Integer>
sensorMonthdatesNumDays=sensorMonthdatesCriticalPM10.reduceByKey(new Sum());

            sensorMonthdatesNumDays.saveAsTextFile(outputPathMonthlyStatistics);
```

```
            // ****************************************
            // Exercise 2 - Part B
            // ****************************************
            // Select, for each sensor, the dates associated with critical PM10
values
            // The critical readings are already available in
criticalPM10readingsRDD.
            // Each line associated with a critical PM10 value can be part of
            // three critical time periods (composed of three consecutive dates).
            // Suppose dateCurrent is the date of the current line.
            // The three potential critical time periods containing dateCurrent
are:
            // - dateCurrent, dateCurrent+1, dateCurrent+2
            // - dateCurrent-1, dateCurrent, dateCurrent+1
            // - dateCurrent-2, dateCurrent-1, dateCurrent
            // For each line emits three pairs:
            // - dateCurrent_sensorid, 1,
            // - dateCurrent-1_sensorid, 1
            // - dateCurrent-2_sensorid, 1
            // The sensorid is needed because the same date can be associated with
many sensors
            JavaPairRDD<String,Integer>
sensorDateInitialSequenceRDD=criticalPM10readingsRDD.flatMapToPair(new
SensorDateInitialSequenceOne());

            // Count the number of ones for each key. If the sum is 3 it means
            // that the dates key, key+1, and key+2, for the sensor sensorid, are
all associated
            // with a critical PM10 value. Hence, key, key+2 is a critical time
period for sensor sensordid
            JavaPairRDD<String,Integer>
DateInitialSequenceCountRDD=sensorDateInitialSequenceRDD.reduceByKey(new Sum());

            // Select the critical time periods (i.e., the ones with count = 3
            JavaPairRDD<String,Integer>
DateInitialCriticalSequenceCountRDD=DateInitialSequenceCountRDD.filter(new
ThreeValues());

            // Get only the dates and store them
            JavaRDD<String>
DateInitialCriticalSequenceRDD=DateInitialCriticalSequenceCountRDD.keys();


      DateInitialCriticalSequenceRDD.saveAsTextFile(outputPathCriticalPeriods);

            // Close the Spark context
            sc.close();
      }
}
```

```
package it.polito.bigdata.spark.exercise2;

import org.apache.spark.api.java.function.Function;

@SuppressWarnings("serial")
public class CriticalPM10 implements Function<String, Boolean> {

	private double PM10threshold;

	public CriticalPM10(double PM10th) {
		PM10threshold=PM10th;
	}

	public Boolean call(String line)  {

		// fields[2] = PM10 value
		String[] fields=line.split(",");
		if (Double.parseDouble(fields[2])>PM10threshold)
			return true;
		else
			return false;
	}

}
```

```
package it.polito.bigdata.spark.exercise2;

import org.apache.spark.api.java.function.PairFunction;

import scala.Tuple2;

@SuppressWarnings("serial")
public class SensorMonthOne implements PairFunction<String, String, Integer> {

	@Override
	public Tuple2<String, Integer> call(String PM10reading)  {

		// sensor#1,1-10-2014,15.3
		String sensorid;
		String month;
		String year;

		// fields[0] = sensorid
		// fields[1] = date
		String[] fields=PM10reading.split(",");

		sensorid=fields[0];
		month=fields[1].split("-")[0];
		year=fields[1].split("-")[2];

		return new Tuple2<String, Integer>(new
String(sensorid+"_"+month+"-"+year),new Integer(1));
	}

}
```

```
package it.polito.bigdata.spark.exercise2;

import org.apache.spark.api.java.function.Function2;

@SuppressWarnings("serial")
public class Sum implements Function2<Integer, Integer, Integer> {

	public Integer call(Integer num1, Integer num2) {
		return num1+num2;
	}

}
```

```
package it.polito.bigdata.spark.exercise2;

import java.util.ArrayList;

import org.apache.spark.api.java.function.PairFlatMapFunction;

import scala.Tuple2;

@SuppressWarnings("serial")
public class SensorDateInitialSequenceOne implements PairFlatMapFunction<String,
String, Integer> {

      public Iterable<Tuple2<String, Integer>> call(String PM10reading)  {

            ArrayList<Tuple2<String, Integer>> initialDates=new
ArrayList<Tuple2<String, Integer>>();

            // sensor#1,January-10-2014,15.3
            String[] fields = PM10reading.split(",");

            // fields[0] = sensordid
            // fields[1] = date
            // Return
            // - date,1
            // - date-1,1
            // - date-2,1
            String sensorid=fields[0];
            String date=fields[1];
            initialDates.add(new Tuple2<String, Integer>(date+"_"+sensorid,new
Integer(1)));

            initialDates.add(new Tuple2<String, Integer>(DateTool.dateOffset(date,
-1)+"_"+sensorid,new Integer(1)));

            initialDates.add(new Tuple2<String, Integer>(DateTool.dateOffset(date,
-2)+"_"+sensorid,new Integer(1)));

            return initialDates;
      }

}
```

```
package it.polito.bigdata.spark.exercise2;

import org.apache.spark.api.java.function.Function;

import scala.Tuple2;

@SuppressWarnings("serial")
public class ThreeValues implements Function<Tuple2<String, Integer>, Boolean> {

	public Boolean call(Tuple2<String, Integer> InitiaDateCount) {
		if (InitiaDateCount._2()==3)
			return true;
		else
			return false;
	}

}
```

```
package it.polito.bigdata.spark.exercise2;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

public class DateTool {

	public static String dateOffset(String date, int deltaDays) {
		String newDate;

		Date d=new Date();


		SimpleDateFormat format = new SimpleDateFormat("MM-dd-yyyy");
		try {
			d = format.parse(date);
		} catch (ParseException e) {
			e.printStackTrace();
		}
		System.out.println(d);


		Calendar cal = Calendar.getInstance();
		cal.setTime(d);
		cal.add(Calendar.DATE, deltaDays);


		newDate=format.format(cal.getTime());

		return newDate;

	}

}
```