# CS 2334
# Project 3: Java Collections Framework

October 12, 2016

**Due: 1:29 pm on Wednesday, October 26, 2016**

## Introduction

For the last two projects, you have been using data that are well-structured. In particular, you could assume ahead of time that you knew which stations were included in the data set and, for each station, you could assume that you knew which data elements were being recorded. In this project, we will break both of these assumptions. At run time, your program will load a pair of configuration files that will inform it of 1) the set of stations that are included in the data set, and 2) the set of measurements that are made at each station. Given this information, your program will create the data structures necessary to load in the data for the set of stations and to compute statistics over the individual measurements.

Specifically, your final product will:

1. Load in a pair of configuration files that describe the set of measures taken (the variables) at the stations, and the set of stations
2. Load in one or more year data files (each data file contains information about all of the stations)
3. Report the minimum, maximum and average of specified statistics for specific stations

# Learning Objectives

By the end of this project, you should be able to:

1. Make use of **HashMaps** and **TreeMaps** to flexibly store data in a structure that is efficient to access
2. Compute statistics over the stored data in a manner that does not rely on *a priori* knowledge of the specifics of the data
3. Continue to exercise good coding practices for Javadoc and for unit testing

# Proper Academic Conduct

This project is to be done in the groups of two that we have assigned. You are to work together to design the data structures and solution, and to implement and test this design. You will turn in a single copy of your solution. Do not look at or discuss solutions with anyone other than the instructor, TAs or your assigned team. Do not copy or look at specific solutions from the net.

# Strategies for Success

- The UML specification constitutes the interface that we will rely on during our testing. Do not make changes to this interface.
- When you are implementing a class or a method, focus on just what that class/method should be doing. Try your best to put the larger problem out of your mind.
- We encourage you to work closely with your other team member, meeting in person when possible.
- Start this project early. In most cases, it cannot be completed in a day or two.
- Implement and test your project components incrementally. Don't wait until your entire implementation is done to start the testing process.
- Write your documentation as you go. Don't wait until the end of the implementation process to add documentation. It is often a good strategy to write your documentation **before** you begin your implementation.

# Preparation

Copy your project 2 implementation into a new *project3* project.

Import the data for project3 into your workspace:

http://www.cs.ou.edu/~fagg/classes/cs2334/projects/project3/project3-data.zip

# Example Output

Below are several examples of output generated by our program (using the *Driver.reportStation()* method. Your implementation should behave in a similar way. Keep in mind that we will be testing many other cases when we evaluate your code.

For the 2015 data file and the Tishomingo station, the ATOT, WSMX, WSPD and WSMN variables report the following Max, Avg and Min values. Note that for WSMN, we are indeed computing the Min, Max and average

```
Station: TISH, Tishomingo, Tishomingo
ATOT, Total Solar Radiation (mega Joules per square meter)
Max: 2015-06-09, TISH: 30.3500
Avg: 15.7975
Min: 2015-12-27, TISH: 0.4000
WSMX, Maximum Wind Speed (miles per hour)
Max: 2015-12-27, TISH: 40.5300
Avg: 16.4307
Min: 2015-06-09, TISH: 6.9100
WSPD, Average Wind Speed (miles per hour)
Max: 2015-12-27, TISH: 25.5100
Avg: 7.8934
Min: 2015-03-22, TISH: 3.0100
WSMN, Minimum Wind Speed (miles per hour)
Max: 2015-12-27, TISH: 11.7300
Avg: 1.6287
Min: 2015-01-13, TISH: 0.0000
```

Note that while these values are correct, the form of this particular String output will not be tested in our unit tests.

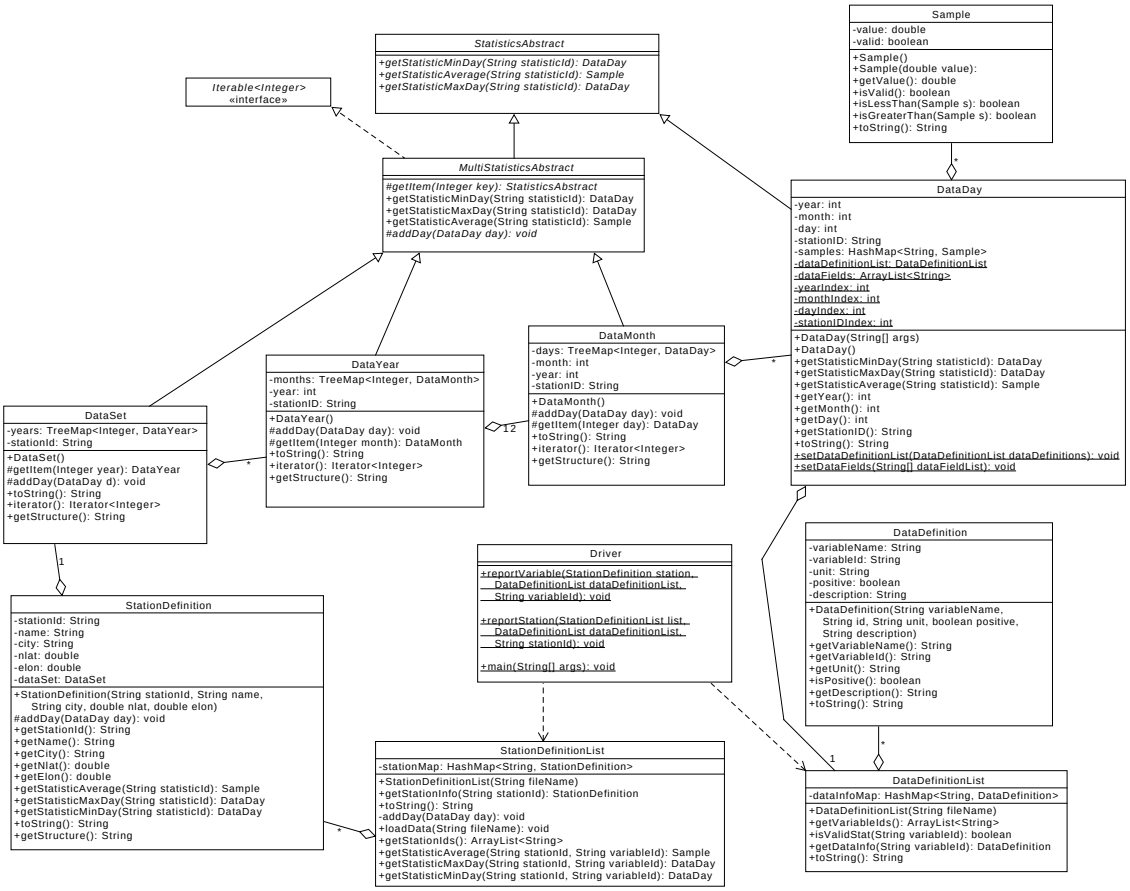
For the 2013, 2014 and 2015 data files, the Fittstown station reports the following summary statistics:

```
Station: FITT, Fittstown, Fittstown
ATOT, Total Solar Radiation (mega Joules per square meter)
Max: 2013-06-07, FITT: 31.4000
Avg: 16.1567
Min: 2015-12-27, FITT: 0.3600
WSMX, Maximum Wind Speed (miles per hour)
Max: 2013-05-29, FITT: 41.0300
Avg: 18.2879
Min: 2013-02-05, FITT: 6.7300
WSPD, Average Wind Speed (miles per hour)
Max: 2014-04-13, FITT: 21.6000
Avg: 9.8574
Min: 2015-01-01, FITT: 2.9900
WSMN, Minimum Wind Speed (miles per hour)
Max: 2013-04-08, FITT: 13.2700
Avg: 3.0825
Min: 2013-01-18, FITT: 0.0000
```

# Class Design

Because your program will not know at compile time what the set of variables will be, we must fundamentally change the way that we are representing the data Samples. Specifically, variables will be identified using a String *variableId*. These IDs will then be used to look-up the associated Sample value, as well compute the minimum, maximum and average of the variable. Likewise, your program will not know ahead of time what the set of stations will be. This set will be loaded at run time.

Below is a complete UML diagram for our key classes.




Implementation notes:

- The **DataDefinition** class will represent the information about a single variable, including its name, ID, a text description and the variable's units, and whether the variable encodes information positively or negatively (the latter is a Boolean). The toString() method should return information in the form of:

```
<ID>, <name> (<units>)
```

  Note that there is no newline character at the end of this String.

- The **DataDefinitionList** class will represent all of the possible variables that are stored for a single station. The constructor will load a configuration file

(DataTranslation.csv); each line of this file encodes a single variable. The class' **HashMap** structure allows for a quick translation between the variableId and the corresponding **DataDefinition** object. The toString() method should return a multi-line String that contains one DataDefinition String per line. Note that there should be a trailing newline at the end of the String, but the order of the individual lines is arbitrary.

- The **StationDefinition** class will represent the information about a station, including its ID, name and location (longitude and latitude), as well as a text description of the station (the city that it is located in).

  A single **StationDefinition** object has exactly one **DataSet**. The *addDay()* method is responsible for inserting a given day into this DataSet.

- The **StationDefinitionList** class will represent the information about all of the stations using a **HashMap**. The constructor will load a configuration file (geoinfo.csv); each line of this file encodes one station. You may safely ignore the *datc* and *datd* columns of this file.

  This class provides a helper **addDay()** method that should be used by the **loadData()** method. This method is responsible for adding a given day to the appropriate station.

- Many classes implement an *addDay(DataDay day)* method. In all cases, this is about adding a new day to the data structure. In each class, the method must decide how to handle this day. For the case of a YearlyData object, it must decide which month to add the day to and then add the day to that month.

- Many classes implement a *getStatisticAverage(String variableId)* method that returns a **Sample**. The **MultiStatisticsAbstract** class provides this implementation for all of its children. The remaining classes (**DataDay**, **StationDefinition** and **StationDefinitionList**) each provide their own implementations.

- *getStatisticMinDay()* and *getStatisticMaxDay()* behave the same way that Average does.

- The **MultiStatisticAbstract** child classes all represent their sub-objects using a **TreeMap**. Keys for the **TreeMap** are Integers (e.g., corresponding to the year in the case of a **DataSet** object, the month in the case of a **Year** object). We use **TreeMap** because we want to preserve the order of the keys

when we perform searches for the minimum and maximum values of a variable. Furthermore, the Map allows us to use meaningful keys (e.g., the actual years).

- The subclasses of **MultiStatisticsAbstract** will each implement **Iterable<Integer>**, enabling MSA to access an Iterator over the subclass' keys. This iterator must produce the keys in numerical order. Hint: look carefully at the **TreeMap** API.

- The **DataDay** class will no longer explicitly represent variables for every possible **Sample** type. Instead, this class will use a HashMap to map a variableId to an instance of a **Sample** for that variable.

- The **DataDay** class is informed of the **DataDefinitionList** and the set of field names through a pair of static methods (*setDataDefinitionList()* and *setDataFields()*. The latter stores the list of field names contained at the top of a data file. In addition, this latter method sets the *yearIndex*, *monthIndex*, *dayIndex* and *stationIdIndex* class variables.

- The **DataDay** constructor will use the **DataDefinitionList** to determine which fields correspond to proper variables that must be represented as Samples.

- Any class that extends the **MultiStatisticsAbstract** class will not explicitly represent specific variables. Instead, the variableId will be used to query value of the variable and to compute the minimum, maximum and average of the variable.

- **DataSet** will contain all of the years associated with a specific station.

- The *toString()* method DataSet should return a String that encodes the stationId:

```
Data Set: stationId
```

The *toString()* methods for Days, Months and Years should return a String that encodes the date and the stationId:

```
YYYY-MM-DD, stationId
```

where MM and DD are dropped if the month and day values are not contained in the object.

Note that there is no trailing newline character for any of these *toString()* methods.

- The *getStructure()* methods should return a String that describes the entire contents of the object (this is useful for debugging). *DataSet.getStructure()* will have the format:

```
TS\n
```

where TS is the String returned by *DataSet.toString()*. Appended to this String is the set of Strings for each of the component *DataYear.getStructure()* methods (in year order).

Year will have the format:

```
\tYear: TS\n
```

where TS is the String returned by *DataYear.toString()*. This line will be followed by a set of Strings containing the Strings returned by *Month.getStructure()*. Each of these will have the following format:

```
\t\tMonth: TS\n
```

where TS is the String returned by *DataMonth.toString()*. This line will be followed by a set of Strings (one for each day) of the format:

```
\t\t\tTS\n
```

where TS is the String returned by *DataDay.toString()*.

*getStructure()* is for your testing purposes only (it is useful to visually confirm that your data structure is being loaded correctly). We will not perform unit tests on the output of this method.

# Project Checklist

Start your project 3 implementation from your project 2. Modify or add classes as follows:

1. Implement your **DataDefinition** and **DataDefinitionList** classes. Use the *DataTranslation.csv* file as a guide to how these should be implemented.
2. Implement the **DataDefinitionListTest** class.
3. Implement your **StationDefinition** and **StationDefinitionList** classes. Use the *geoinfo.csv* file as a guide to how these should be implemented.
4. Implement the **StationDefinitionListTest** class.
5. Update your **StatisticsAbstract** class according to the UML.
6. Update your **DataDay** implementation to match the requirements of the UML.
7. Update your **DataDayTest** class to test the new implementation of the class.
8. Update the **MultiStatisticsAbstract** to match the UML specification.
9. Modify your classes called **DataMonth**, **DataYear** and **DataSet** classes.
10. Create a unit test that loads the variable list, station list and data, then carefully tests the minimum, maximum and average statistics computations for multiple stations and variables.
11. Your *Driver.main()* method will only be used for testing purposes only. This method should load the configuration files, one or more data files and print a set of useful Strings.

# Assumptions

You may make the following simplifying assumptions:

1. All stations in the data set will be included in the geoinfo.csv file. The columns in this file are pre-defined (i.e., they won't change).
2. All files exist and are properly-formatted csv files.

3. The data file will include the following columns: YEAR, MONTH, DAY, STID. However, they may occur at any column.

4. All variable values in the data set file can be interpreted as doubles.

# Notes

1. The first line of the data file determines the names of each of the fields in this file (each column is a field). While most of these columns are variables that are measured by the stations, not all are. In your **DataDay** constructor, you should only include Samples for fields that correspond to variables, as determined by the *isValidStat()* method.

2. The *Boolean* class provides methods for parsing Boolean values from text.

# Final Steps

1. Generate Javadoc using Eclipse for all of your classes.

2. Open the *project3/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that all of your classes are listed (five primary classes plus four JUnit test classes) and that all of your documented methods have the necessary documentation.

# Submission Instructions

- All required components (source code and compiled documentation) are due at 1:29 pm on Wednesday, October 26 (i.e, before class begins)

- Submit your project to Web-Cat using one of the two procedures documented in the Lab 2 specification.

# Grading: Code Review

All groups must attend a code review session in order to receive a grade for your project. The procedure is as follows:

- Submit your project for grading to the Web-Cat server.

- Any time following the submission, you may do the code review with the instructor or one of the TAs. For this, you have two options:

    1. Schedule a 15-minute time slot in which to do the code review. We will use Doodle to schedule these (a link will be posted on Canvas). You must attend the code review during your scheduled time. Failure to do so will leave you only with option 2 (no rescheduling of code reviews is permitted). Note that schedule code review time **may not** be used for help with a lab or a project
    2. "Walk-in" during an unscheduled office hour time. However, priority will be given to those needing assistance in the labs and project

- Both group members must be present for the code review

- During the code review, we will discuss all aspects of the rubric, including:

    1. The results of the tests that we have executed against your code
    2. The documentation that has been provided (all three levels of documentation will be examined)
    3. The implementation. Note that both group members must be able to answer questions about the entire solution that the group has produced

- If you complete your code review before the deadline, you have the option of going back to make changes and resubmitting (by the deadline). If you do this, you may need to return for another code review, as determined by the grader conducting the current code review

- The code review must be completed by Wednesday, October 19th to receive credit for the project

# Notes

- There are multiple ways to define the *average* wind speed over a year. For this project, we will define a year's average as the average over the months that belong to that year.

- Remember that arrays are zero-indexed and months are one-indexed.

# References

- The Java API: https://docs.oracle.com/javase/8/docs/api/

- The Oklahoma Mesonet: http://www.mesonet.org

- The API of the *Assert* class can be found at:
  http://junit.sourceforge.net/javadoc/org/junit/Assert.html

- JUnit tutorial in Eclipse:
  https://dzone.com/articles/junit-tutorial-beginners

# Rubric

The project will be graded out of 100 points. The distribution is as follows:

**Correctness/Testing: 45 points**

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against our set of tests. These unit tests will not be visible to you, but the Web-Cat server will inform you as to how many tests your code passed/failed. This grade component is proportional to the fraction of tests that your code passes (so 22.5 points means that your code passed half of the tests).

**Style/Coding: 20 points**

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

**Design/Readability: 35 points**

This element will be assessed by a grader during the code review. Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation
- Missing or inappropriate inline documentation
- Inappropriate choice of variable or method names
- Inefficient implementation of an algorithm
- Incorrect implementation of an algorithm
- Incomplete coverage of your Unit Tests. We expect that your unit tests will test all lines of your code

If you do not submit compiled Javadoc for your project, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions(where points may be deducted).

**Bonus: up to 5 points**

You will earn one bonus point for every twelve hours that your assignment is submitted early.

**Penalties: up to 100 points**

You will lose five points for every twelve hours that your assignment is submitted late (up to 48 hours). Submissions will not be accepted more than 48 hours after the deadline.