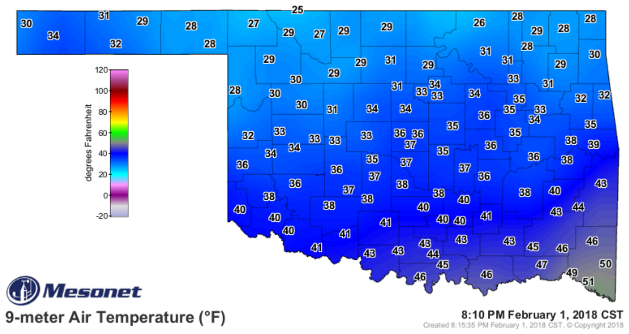
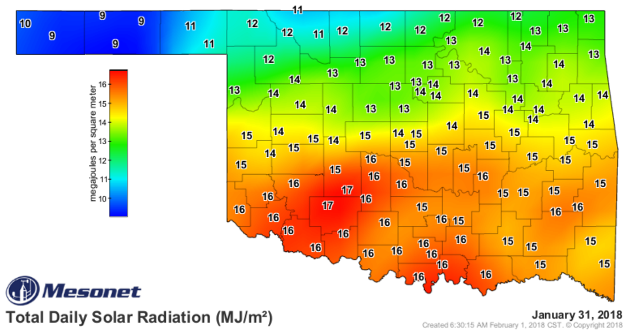
Project 1: Reading data from files
Computer Science 2234

Due: 11:59pm February 15, 2018

Overview


degrees Fahrenheit
Mesonet
9-meter Air Temperature (°F)
8:10 PM February 1, 2018 CST
Created 8:15:35 PM February 1, 2018 CST. © Copyright 2018

(a) Temperatures on February 1, 2018


megajoules per square meter
Mesonet
Total Daily Solar Radiation (MJ/m²)
January 31, 2018
Created 6:30:15 AM February 1, 2018 CST. © Copyright 2018

(b) Total Solar radiation for January 31, 2018

Figure 1: Example mesonet data

You will be using data from the Oklahoma Mesonet[1] for all of your projects this semester. This network of observing stations is unique to the State of Oklahoma. It provides a variety of weather observations for every county in Oklahoma every five minutes. It has been in existence for 20 years and is an invaluable resource for understanding our weather and climate.

Project 1 Details

For this project, you will focus on a single station (Norman) and temperature at 1.5 and 9 meters above the ground and solar radiation data. Later projects will expand to other stations and additional weather data. The data that we provide to you is a comma separated file with the daily summary information for Norman.

[1] http://www.mesonet.org

We have provided one day of weather data for Norman and your job is to read in the data file, parse the data, create appropriate objects from this data, and summarize the data using maximum, minimum, and average mathematical functions. More details are below in the Milestones section.

Note, due to unforeseen circumstances such as extended power outages, sometimes data is unavailable for a day at a particular station. These values are represented using **-900** and beyond. Make sure you don't accidentally use these values in your statistical summaries.

Learning Objectives

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

- Parse structured data from a file
- Create objects using data parsed from a file
- Use mathematical transformations on strings
- Implement mathematical functions in code
- Understand why unit testing is useful
- Implement unit tests to verify code is working
- Provide proper documentation in JavaDoc format

Academic honesty

You may write your program from scratch or may start from programs for which the source code is freely available on the web or through other sources (such as friends or student organizations). If you do not start from scratch, you must give a complete and accurate accounting of where all of your code came from and indicate which parts are original or changed, and which you got from which other source. Failure to give credit where credit is due is academic fraud and will be dealt with accordingly.

Milestones

A milestone is a significant point in the development of your project. Milestones serve to guide you in the design and development of your project. Listed below are a set of milestones for this project along with a brief description of each.

1. Understand and use a design given by a UML class diagram
   - UML will help you to organize your program. This is likely the largest program you have written. We will begin the UML in class when we hand out the lab. This should help to get you started. We will post the class UML to Canvas. You must implement all of the methods given in the UML class diagram.
2. Use proper documentation and formatting (javadoc and in-line documentation)
   - This is important for debugging and for communication with users of your program and your future self. You may re-use your project code in future projects this semester so don't make it obfuscated.
3. Create a class called **Project1** that will contain your **main** method and run the project.
4. Create a class called **DayData** that will read in a day of data from a text file, parse it, and create the daily data.
   - Look up String.split() in the Java API to find a quick and clean way to break up your strings
5. Create a class called **TimeData** that will hold measurements for a station every 5 minutes.
6. Create a class called **Measurement** to hold a single measurement and flag it if is bad (so you don't use it in the statistics).
7. Create directory named **data** in your project, and place **20180101nrmn.csv** file there.
8. Use simple File I/O to read the weather data from a file and create the appropriate objects from the data. File name follows the following format: yyyymmddstid.csv

- yyyy - year, here it is 2018
- mm - month, here it is 01
- dd - day, here it is 01
- stid - station id, here it is **nrmn**

20180101nrmn.csv

9. Implement unit tests to verify that your parsing is done correctly.
   - Make a file with data that you know is correct (just do a single day) and verify the max/min and averages that are read in. You will submit this file as well.
10. Implement unit tests to verify that average, maximum, and minimum are computed correctly
    - Setup data that you know what the max/min/averages will be and ensure that these are the computed values.
11. Implement a program that outputs the minimum, average, and maximum temperatures at both heights and minimum, average, and maximum solar radiation, followed by the accumulated solar radiation for a day.

```
2018-01-01, nrmn:
 Air Temperature[1.5m] = [-14.9000, -10.8931, -6.5000],
 Air Temperature[9m] = [-14.9000, -6.5000, -10.9726],
 Solar Radiation = [0.0000, 139.8264, 564.0000, 40270.0000]
```

12. **Bonus** Find a difference between 1.5m and 9m air temperatures, that is, TAIR and TA9M. This difference will be TAIR - TA9M. See if this difference is negative or positive in relation to solar radiation, since solar radiation should be close to zero for the night hours. Write output to a file named *tempDiff.txt* in the following way:

```
<date> <difference: positive or negative> <daytime: true or false> \n
```

Example:

```
Date_time           Difference Daytime
2018-01-01 00:00    negative   false
```

Resources

If you are confused or need help, after you have read the entire Project description, realize that the following resources are available to you. Please come to office hours or email the TAs or Dr. Jabrzemski.

- Java JDK 8 API http://docs.oracle.com/javase/8/docs/api/

Project (Class) Documentation

Below the import statements at the top of every java source file (and above the class declaration), you must include a documentation block at the top of the file that includes the following:

```
/**
   @author <your name(s)>
   @version <today's date>
   Lab <number>

   <A short, abstract description of the file>
*/
```

Note that the notation:

```
<some text>
```

indicates that you should provide some information and not actually write the less-than and greater than symbols. Also, the double splats (stars) at the beginning of the comment are necessary to be recognized by Javadoc as being important.

Instance/Class Variable Documentation

As we start to implement classes that contain either *instance* or *class* variables (more on the difference in later labs), you must have a line of documentation for each of these variables. For example, consider a *Person* class. Here are some example documentation lines with variable declarations:

```
/** First name of the Person */
String firstName;

/** Last name of the Person */
String lastName;

/** Unique identifier */
int idNumber;
```

Note that this Javadoc format for documentation (with the double splat) **is not** required for variables defined locally within a method. Nevertheless, you still need to document the meaning of your variables.

Method-Level Documentation

Every method must include documentation above the method prototype using standard Javadoc markup. This documentation should be sufficient for another programmer to understand *what* the method does, but not the details of *how* the method performs its task. For example, consider a method that will test whether a value is within a range and whose prototype is declared as follows:

```
public static boolean isInRange(double min, double max, double value)
```

The documentation for this method will be placed above the prototype in your java file and might look like this:

```
/**
   Indicate whether a value is within a range of values

   @param min Minimum value in the range
   @param max Maximum value in the range
   @param value The value being tested
   @return True if value is between min and max.  False if outside
this range.

*/
```

Note that this example includes all the required documentation elements:

- A short, intuitive description of what the method does (not how it does it),
- A list of the parameter names and a short description of the *meaning* of the parameters, and
- A short description of the return value.

Inline Documentation

Inside of each method, you must also include documentation that describes *how* a method is performing its task. This documentation should be detailed enough for another programmer to understand what you have done and to make modifications to your code. Typically, this documentation is preceded by “//” and occupies a line by itself ahead of the code that is being documented. While each line of code could be documented with its own documentation line, it is typically not necessary or appropriate. Instead, we typically use a single documentation line to capture what a small number of code lines is doing.

In addition, inline documentation should be done at a logical level and should not simply repeat what the line of code says.

Here is an example:

```
public static boolean isInRange(double min, double max, double value)
{
    // Check lower bound
    if (value < min)
    {
        return false;
    }

    // Check upper bound
    if (value > max)
    {
        return false;
    }

    // Within the boundaries
    return true;
}
```

Program Formatting

Our Web-Cat server will enforce a particular program formatting standard. In addition to the project/class- and method-level documentation, it will also check for several other items, including:

- Proper indentation. Indents must be done with spaces only (and not tabs).
- A space between a conditional keyword (such as *if* or *while* and the following open parenthesis.
- Curly brackets on their own lines.

- Code blocks surrounded by curly brackets. For example, an *if (...)* statement must be followed on the next line with an open curly bracket.
- *if* and *while* must be followed by a space, then an open parenthesis.
- Lines with at most 120 characters.
- Use of all declared variables.
- Use of all import statements.

Note that Eclipse can be configured to automatically do much of this formatting for you (see our Web-Cat page).

Turning in the project

To turn in the project, go to your eclipse project directory (mine was called Project1), create a zip file of the entire directory, and turn the zip file into Canvas in the Project1 Dropbox. This zip file will include your code and your java doc.

Grading the project

Your project will be graded in an interactive session with the TAs.

Grading Rubric

- 50 points for correctly implementing the different classes.
  - 50 points if there is no mistake. The files are opened and read correctly, the data is parsed, the objects are created, and the statistical functions are implemented and values created.
  - 45 points if there are several minor mistakes. For example, the value column is offset by one (remember java counts from 0!).
  - 30 points if you have one major mistake. An example of a major mistake would be failing to correctly parse the data or failing to properly implement the statistical functions.
  - 20 if there are several major mistakes.

  - 10 points if you have nothing working but have at least tried something.

- 20 points for your broad design: Your code should follow OOP principles and be easy to read. Points will be deducted for obfuscated code.

  - 20 points for code the correctly implements the UML diagram we did in class for the project and follows OOP principles
  - 15 points if your code if not particularly pretty because of technical difficulties but it is obvious that efforts have been made
  - 10 points if your code if not particularly pretty because of technical difficulties and the developer (you) are not able to to justify convincingly your own choices of implementation.

- 20 points for fully documented code, with java docs produced

  - 20 points if the code is well documented and you have produced (and turned in) the java docs along with your code
  - 15 points if your code is well documented but you forgot to make java docs or if you did java docs but your comments are lacking in a minor way
  - 10 points if your documentation or java doc is lacking in a major way (e.g. major functions not commented)

- 10 points will be awarded for the quality of your Unit Testing.

- Up to 10 points can be earned for the Bonus described above. To earn all 10 points, this must be done in a OOP manner and be code that can be re-used (e.g. if Project 2 requires statistics but on additional data, hint hint).