Project 1: Global Weather

1 Understand

1.1 Know the Objectives

After completing this project, you should have a solid grasp of these topics and how to code them:

- ArrayList
- Binary Search
- Linear Regression (Least Squares)

1.2 Understand the Problem

Assume you were interested in global weather and were scouring the internet for a good dataset. You happen upon one that looks great, found at this link, listing average daily temperature for major cities of the world. The data in the downloaded file (provided) is ample, consisting of almost 3 million rows. You decide to use an ArrayList to store and manipulate the data.

After reading in the data, create objects for each row and populate the ArrayList, then sort the ArrayList data to ensure it is in the desired order. WeatherReading objects will need natural ordering to support this ordering, sorting by country, state, city, year, month, and day (*note the omission of region*).

Core methods will take advantage of the sorting. One linchpin method will search for data for a given city, retrieving key list data like the starting index of that city's data. Because the data is sorted, it can use a *binary search* which will be much faster than the linear search coded into ArrayList (O(log n) versus O(n)).

2 Design

Before you code, create appropriate design documentation and obtain feedback. Update designs per feedback, then use them during the rest of the development process and submit them as part of your project. Recommended tool: Violet, which creates the diagram types we need, along with others.

2.1 Draw a Class Diagram

Draw and submit a class diagram for this project. Show interfaces and all classes (except Main, maybe). Since we're learning ArrayList, include a simple box for that class, its private inner classes, interfaces, etc.

2.2 Draw an Object Diagram

Draw and submit an object diagram for this project. Start with Main. Capture the moment at which several elements have been added to the collection, including some duplicates.

3 Code

3.1 Do Use These

- ArrayList (Java's version, though it's good to know the author's code as a knowledge baseline)
- Binary search (author code, Ch. 13, Search.java, binarySearch method)
- Iterators

3.2 Don't Use These

- For loading file data: no parallel arrays or any other data structures or collections; such use will cause work to be rejected and returned for rework.
- For other methods, create no additional ArrayLists except perhaps in getCityListStats where it might be convenient for working with years.
- Stream operations (including stream(), asList(), and similar methods). These are inefficient (object overhead) and unnecessary when we're coding inside a collection and have direct access to data.

3.3 Write Preconditions

Write preconditions where they make sense, especially where Java won't throw a reasonable exception. In the future, you should always think about these, even if the project doesn't specifically mention it. You don't need them for the record holding file data; let's trust that data for this project.

3.4 Implement Other Requirements[1]

Carefully read the interface's JavaDoc and other comments; there is a lot of good background and implementation information there. FYI: that JavaDoc can be brought into the implementing classes.

3.4.1 Records

3.4.1.1 WeatherReading

- Create a *record* (not a class) to hold one row of data from the file. Include these fields, in this order: region, country, state, city (all String), month, day, year (all int), and avgTemperature (double).
- Add natural ordering using a multi-level approach using these fields, in ascending order: country, state, city, year, month, day.
- Add a compareTo-type method called that compares only country, state, and city. You'll need it.
- Add a full/proper equals override that is harmonious with natural ordering[2]. Ask if you are unsure.

3.4.1.2 CityListStats

Create a record (not a class) to hold list statistics for a city, useful for method returns. Include these fields, *in this order*, with *exactly these data types*: startingIndex (int), count (int), years (int[]).

3.4.2 GlobalWeatherManager

Make this class iterable; it should iterate over WeatherReading data. Test this in your unit tests.

3.4.2.1 Constructor

The constructor should take a single parameter, a File reference for the file to be read. It is expected that the constructor will throw a FileNotFoundException if there are file issues[3].

3.4.2.2 getCityListStats

This is a challenging method. I suggest you implement incrementally and check your work at each point. Remember than when verifying/testing list content, we want to be aware of position, e.g., the thing of

[1] Failure to follow exact naming and data types will result in code that will not compile with the instructor's test code. Such code will be returned to you for rework, wasting valuable time (yours and mine).
[2] We should not have conflicts or confusion between the way these methods work; they should focus on the same data. Failing to do so means that in sorting and comparison situations, unexpected results will occur when the equals method result isn't the same as compareTo method result of 0 (which clients are reasonable to expect to match).
[3] Failure do to this will cause a failure to compile with the instructor's test code; your work will be rejected/returned.

interest at the start of the list, in the middle somewhere, and at the end of the list. Algorithms often still fail at one or more of those positions after a successful quick test.

Notes:

- Binary search should be integrated into this method. Read the author's code carefully, and perhaps do some debugging or print output to make sure you understand what is happening. Look carefully at where the "found it" portion of the code sits; most of your work goes there.
- Remember that binary search will find a match, in this case for the right country/state/city. But that doesn't mean it finds the *first occurrence* of that data; do a linear "back up" (backward search) from the found location to ensure you return the first occurrence of the data. Good testing is needed here.
- Once you find the start of that city, you'll need to *count* the pertinent rows and collect information on the *non-duplicated years* that are covered by the data (e.g., we only care to see 2017 one time).

3.4.2.3 getTemperatureLinearRegressionSlope

Learn about linear regression, best-fit lines, and the Least Squares method here, or via other internet searches. The math is straightforward. Here, we want to use years as x coordinates and average temperatures as y coordinates, finding the slope of the best-fit line.

How would this be used? While our data doesn't cover a huge number of years, one could do some analysis to see whether, over time, the temperatures seem to be changing in a predictable way for a given month and day. A positive slope would indicate an increasing temperature, negative a decreasing one.

3.4.2.4 calcLinearRegressionSlope

This is a support method for the method mentioned just above. The least-squares math will live here.

3.5 Use the Interface(s)

Use the provided interface(s) for the project. Your class(es) should implement these and must exactly follow the signatures in the interfaces. Add helper methods to your classes, of course, but do not alter the interface in any way unless instructed to do so.

4 Document

4.1 Follow the Style Guide

Follow the Course Style Guide, which is linked in the Reference section of the Modules list in Canvas. Failure to do so will result in the loss of points.

4.2 Write JavaDoc

Write complete JavaDoc notation for the GlobalWeatherManager class, along with records WeatherReading and CityListStats. *Complete* means that an -Xdoclint:all comes back with no errors or warnings. If there are errors or warnings, you will lose points. Check this the correct way: Build \ Rebuild Project, then check the Build Tab (near the bottom left near the status bar) and expand each class.

5 Test

5.1 Write Unit Tests

Write unit tests for the GlobalWeatherManager class. Create a JUnit5 test class (named GlobalWeatherManagerTest, per convention). Ensure that all methods and state are fully tested. Run your tests with code coverage; for full credit, tests must cover 100% of code statements in the production class.

Tests on the full file (which mine are, BTW) will run super slowly; let's talk about why that is. For full credit, fix the speed issue; let's discuss some approaches. Clever work here means the first test will run slowly, while the rest will run quickly.

5.2 Verify Code Meets Acceptance Criteria

Work will be returned to you for rework (with a deduction for late work) if it doesn't meet the acceptance criteria listed here:

- …uses provided interfaces and data files with no modifications to either.
- …compiles with all required Xlint and Xdoclint command line additions.
- …runs without throwing any exceptions.
- …has no failing unit tests.
- …loads and sorts data correctly.
- …has a Main class with a runnable main method that demonstrates some broad/deep capabilities of GlobalWeatherManager.

6 Demonstrate

Write a Main class with a runnable main method that passes a File argument to the constructor of GlobalWeatherManager, handling file issues gracefully should they occur (i.e., a message and graceful termination). Demonstrate the class's capabilities. Think of a handful of demonstrations that tell your instructor that the major functionality is in place and working (deeper/broader methods, not trivial ones). Your output shouldn't be voluminous; a small bit of well-labeled output is what we want here.

7 Extra Credit (1%)

For extra credit, in addition to making your tests hit 100% *line* coverage (code statements) in your production code, also make them hit 100% *branch* coverage.

(continues…)

8 Measure Success

Area	Value	Evaluation
Design docs – initial	5%	Did you submit initial design documents, and were they in reasonable shape to begin coding?
Design docs – final	5%	Did you submit final design documents, and were they a good representation of the final version of the project?
WeatherReading record	10%	Was this record created per specifications? Is natural ordering coded properly? Is the equals method a proper/full override, well coded?
CityListStats record	5%	Was this record created per specifications?
GlobalWeatherManager – general	10%	Does the constructor do its job, reading in data from the file, creating objects, adding them to the list, and sorting the result?
GlobalWeatherManager – getCityListStats	15%	Is the binary search properly integrated? Does it return the requested information in all cases? Does it pass the instructor's unit tests?
GlobalWeatherManager – get methods	10%	Do get__ methods function as requested? Do they pass the instructor's unit tests?
GlobalWeatherManager – iterator	5%	Is a valid iterator returned? Is it usable to loop through data?
GlobalWeatherManager – linear regression	10%	Are linear regression methods working properly, returning the right data? Are they well coded, with code redundancy reduced?
Main	5%	Does this class demonstrate the depth and breadth of the model and its most interesting code?
Unit tests	5%	Were test classes created for requested classes? Were all methods tested? Do all tests run and pass? Are proper verification methods in place?
Test line coverage	5%	Do tests cover 100% of the lines in the production class?
JavaDoc	5%	Did you use JavaDoc notation, and use it properly? Is the Xdoclint run clean?
Style/internal documentation	5%	Were elements of style (including the Style Guide) followed?
Extra credit	1%	Was 100% branch coverage achieved?
Total	**100**%	

9 Submit Work

Follow the course's Submission Guide when submitting your project. *But please don't submit huge data files that were supplied with the project; there's no reason for Canvas to be burdened with that.*

10 Suggested Plan of Attack

Abbreviations: WR = WeatherReading GWM = GlobalWeatherManager

1. Create WR record.
2. In GWM constructor, read in data, instantiate WR objects, put them into the ArrayList.
3. In GWM, write most methods (except for getCityListStats, discussed below).
4. Implement comparison methods and code the multi-level sort in WR. Call for sorting of the ArrayList in GWM constructor.
5. Create CityListStats record. Code getCityListStats method in GWM (which requires integration of binary search code into the method).
6. Write tests, fix speed issues. Write/check JavaDoc. Do style check and a general code cleanup pass.