HW4: Analyzing Country Data

Goals

To build a non-trivial program with Java that involves reading texts from a file, and to practice using the List interface in JCL. This is a pair-programming assignment, and only one member needs to submit the work on Blackboard.

General Tips

1. Compile your code often, so that you could catch and fix immediate errors before they build up.
2. Check your `imports` (learn to read error messages).
3. You may choose to search some of your error messages on the Internet.

Setup

In the folder you downloaded, you will see `CountryDataset.csv`, a dataset containing information about countries, created using data from the World Bank. Your program will read this file in Step 3 of Programming Steps and Tips (described below). You may ignore the following three files in the folder for now, because they are only for the bonus part:

1. BarChart.java
2. javadoc (folder)
3. jfreechart-1.5.0.jar

To assist with this assignmnet, it may be useful to refer to the code from class; the appendices of our book may also be useful. The official Javadoc of List will likely be helpful for seeing what methods are available.

This Assignment

For this assignment, you will write a program that reads from a text file (`CountryDataset.csv`) with one line for each country. Your program will need to represent each country as an instance of a class `Country`. It will need to convert the text file data into a `List<Country>` and display the data as a text list of all countries, in sorted order for a given indicator.

This will give you practice working with the List ADT, `String`, `File`, `Scanner`, loops, and simple classes. It's also a chance to see how your programming skills might be used to analyze large datasets.

1. The input file format

The input file is a text file that contains 0 or more lines of countries, with an initial line that gives the meaning of each item in the line. After that first line, each line of the input file will represent a single country via a comma-delimited list. Here's an example of the representation of Afghanistan, the alphabetically first country:

```
Afghanistan,0.31647873497721196,18168.86,82.60891342163086,17.011405995013824,0.103138023195208,3.00660949046542
```

Each of the numbers represents some information about the country. Specifically, the meaning of each of the items is as follows:

1. Country Name
2. CO2 emissions (metric tons per capita)
3. Total greenhouse gas emissions (kt of CO2 equivalent)
4. Access to electricity (% of population)
5. Renewable energy consumption (% of total final energy consumption)
6. Terrestrial protected areas (% of total land area)
7. Population growth (annual %)
8. Population (total)
9. Urban population growth (annual %)

Open up the CSV file with a **text editor** to take a look at all of the data. (Don't open it with Excel.)

The information in CountryDataset.csv is based off of the World Bank Indicators dataset. The original dataset contains information about each country for each indicator (item) and year from 1960-2019 (although some years are missing for some countries). For each country, the dataset I've given you contains a single number for each indicator. That number is the average value of the indicator from 2012-2019, skipping any years where there wasn't data for that country.

For your testing, you can use the entire data file; but you may also test on smaller subsets of the data to make sure that your output is correct, and that your program doesn't crash if there are no countries (—an edge case).

2. The command-line syntax

Your `main` method should be structured so that the program expects two command-line arguments. The meaning of the command line arguments is as follows:

1. The path to the file containing the country data. This includes the name of the file, and any directory information if the file is not in the same directory as the java program you're making. E.g., if the country file is in a subdirectory of the directory containing the java program and that subdirectory is named "data", this argument would be data/CountryDataset.csv. (Note: there's nothing special you have to do to handle “CountryDataset.csv” versus “data/CountryDataset.csv”. `File` will do the right thing in both without you having to think about it.) If the file is in the same directory as your java programs, then this argument should just be CountryDataset.csv.
2. The name of the indicator that we should sort the data on. This should be one of: CO2Emissions, TotalGreenhouseGasEmissions, AccessToElectricity, RenewableEnergy, ProtectedAreas, PopulationGrowth, PopulationTotal, or UrbanPopulationGrowth.

3. The expected output

The program should print to standard output (i.e. `System.out`) one country per line in sorted order. The sorted order should be based on the indicator passed in as the second command-line arguments. For example, if "PopulationTotal" was the second command line argument, then the list of countries should begin with the the country with the largest population total, followed by the one with the second to largest population total, and so on. Here's an example of the first few lines of my output when I use the following command line:
`java CountryDisplayer CountryDataset.csv PopulationGrowth`

```
Oman,16.1347536949426,62201.55080448151,100.0,0.0,2.5718221209005327,5.6534794468626615,4218868.428571428,7.1402
Qatar,41.0352002847597,103155.12072137301,100.0,0.0,9.636156954527268,4.459003515508291,2531190.1428571427,4.536
Maldives,2.83161983745105,NaN,99.86328582763669,0.9983738251675678,1.1327304513667067,4.34349154240501,455767.28
Equatorial Guinea,4.840438109455877,6374.17105425022,66.46417236328125,6.6940880271428025,19.267839302429465,4.0
Lebanon,3.91594244914353,NaN,99.99894460042317,4.385490965120395,2.5948724159837133,3.9283229231561343,6374323.1
...
Eritrea,NaN,4977.88803281947,45.26542599995931,80.04016597659202,4.8715939593562325,NaN,NaN,NaN
```

(The "countries" we're using are whatever the WorldBank data listed as a country. That means that some territories, like Puerto Rico and Greenland, have their own data, and even some groups of countries, like "Euro area", are included.)

For sorting, you can break ties however you like, but NaN values for the category you're sorting on should always appear at the very end. NaN stands for not a number—it means there was no data present in the WorldBank file for that country and indicator. You don't need to pay special attention to "NaN"s in the CSV file, as `Double.parseDouble("NaN")` will conveniently return `Double.NaN` (a constant of the Double class)—see the provided `Country` constructor.

4. [Optional] Examining your results

Take a look at what your program tells you about the country dataset. Discuss whether you're surprised by any of the results, and whether you have hypotheses about the reason for any of these results (e.g., why particular countries have higher use of renewable energy). Brainstorm several questions that you'd like to investigate further based on these data. In a document named "AnalysisDiscussion.txt", write up some notes about your discussion, making sure to include information about what results surprised you, your hypotheses about the reasons for these trends (these can be trends that were surprising or unsurprising to you), and your questions that you'd like to investigate further. You may also want to do this using the tools developed in the bonus part.

Programming Steps and Tips

Step 1: Writing the `Country` class

I provide the following starter code for the `Country` class. All you need to do is add the getter and setter for the `countryName` data field, and finish implementing `getFeature`, `setFeature`, and `toString`. Note that both `getFeature` and `setFeature` have a parameter that specify the "feature" to get or set—this conveniently avoids writing eight separate getters/setters for each of the eight features, and allows clients to get a feature by passing a String varialbe (later in Step 5). (Optional: You may be interested in using a switch statement in your getFeature/setFeature to deal with multiple string options. You can read about switch statements in Supplement 1.51 in your book.)

```java
public class Country {
    private String countryName;
    private double co2Emissions;
    private double totalGreenhouseGasEmissions;
    private double accessToElectricity;
    private double renewableEnergy;
    private double protectedAreas;
    private double populationGrowth;
    private double populationTotal;
    private double urbanPopulationGrowth;

    public Country(String[] countryData) {
        countryName = countryData[0];
        co2Emissions = Double.parseDouble(countryData[1]);
        totalGreenhouseGasEmissions = Double.parseDouble(countryData[2]);
        accessToElectricity = Double.parseDouble(countryData[3]);
        renewableEnergy = Double.parseDouble(countryData[4]);
        protectedAreas = Double.parseDouble(countryData[5]);
        populationGrowth = Double.parseDouble(countryData[6]);
        populationTotal = Double.parseDouble(countryData[7]);
        urbanPopulationGrowth = Double.parseDouble(countryData[8]);
    }

    // add the getter and the setter for countryName here...
    // IMPLEMENT getCountryName
    // IMPLEMENT setCountryName

    // return the corresponding value of the given featureName. The passed in featureName
    // should start with an upper-case letter: "CO2Emissions", "TotalGreenhouseGasEmissions", etc.
    // Please use equals (not ==) to compare strings.
    public double getFeature(String featureName) {
        return 0; // CHANGE ME
    }

    // set the corresponding value of the given featureName as newValue. The passed in featureName
    // should start with an upper-case letter: "CO2Emissions", "TotalGreenhouseGasEmissions", etc.
    // Please use equals (not ==) to compare strings
    public void setFeature(String featureName, double newValue) {
        // IMPLEMENT ME
    }

    public String toString() {
        return null; // CHANGE ME
    }

    // Testing public methods:
    public static void main (String[] args) {
        String[] data = {"Country1", "1", "2", "3", "4", "5", "6", "7", "8"};
        Country country1 = new Country(data);
        // Testing toString:
        System.out.println(country1);
        // Or use one line:
        Country country2 = new Country(new String[] {"Country2", "9", "8", "NaN", "6", "5", "4", "3", "2"});
        // Testing toString:
        System.out.println(country2);
        // Testing getFeature (after it's been implemented)
        // System.out.println(country1.getFeature("CO2Emissions")) // should show 1
        // System.out.println(country1.getFeature("PopulationTotal")) // should show 7
    }
}
```

The parameter `String[] countryData` of the constructor is an array of `String` s. Refer to Supplement 1.86-96 for *arrays*. Make sure you understand the constructor: search the `Double.parseDouble()` method in the official Javadoc; `Double` is the *wrapper class* for `double` (see S1.97-105).

[Important] The `Country` class does *not* deal with files at all (the `CountryDisplayer` class will take care of reading files later). All it needs to create a `Country` instance is an array of `String` s passed to the constructor (and this array will be supplied by its client, `CountryDisplay`).

Please make sure this file compiles and passes the tests before moving to the next step. For example, to test the `toString` method using the existing code in `main` :

```
java Country
```

shoud display

```
Country1,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0
Country2,9.0,8.0,NaN,6.0,5.0,4.0,3.0,2.0
```

Uncomment the last two lines in `main` to test `getFeature`; please test other methods as well, and you may leave your testing code in `main`.

Step 2: Structure of the `CountryDisplayer` class

`CountryDisplayer` class has one data field which stores a list of countries. You should make sure that you put essential self-contained operations into separate methods (`sortCountryList` and `displayTextCountries`). This way, the main method reads like a high-level outline of the process you're implementing, rather than being a mess of all the code in a single method. Please write (private) helper methods for these two major (public) methods (Step 5).

```
public class CountryDisplayer {

    private List<Country> countries;

    public CountryDisplayer(String filePath) {
        // Read the country file and load the countries.
        // IMPLEMENT ME
    }

    public ?? sortCountryList(??) {
        // IMPLEMENT ME
    }

    public ?? displayTextCountries(??) {
        // IMPLEMENT ME
    }

    public static void main(String[] args) {
        // IMPLEMENT ME
        // If there's no command-line argument, print a usage statement
        // and exit. Otherwise, use args[0] as the input file path.
        if (args.length == 0) {
           // ...
        }

        // Call the methods you defined above to load, process (sort), and display the countries.
    }
}
```

Step 3: Implementing the constructor of `CountryDisplayer`

The constructor has one parameter: a string with the location of the file with the countries (e.g., `"CountryDataset.csv"`). Make sure to test that your CountryDisplayer properly loads different files of country data. Your program should never throw an exception or error message if given a valid filename that has a properly formatted country dataset (containing 0 or more countries). However, you don't need to worry about recovering if the user gives you a file that doesn't have the same format as the example one (e.g., values not separated by commas).

You will find the `LineReader.java` from HW1 useful as an example of reading a file line by line. Always use `try-catch` when dealing with files—Supplement 2 provides more details about reading Files.

You will also find the `split` method of the `String` class helpful. Suppose `line` is a `String` variable representing one line of text read from the CSV file (data separated by commas), then `line.split(",")` would return a String array (`String[]`) where each element is one data entry. Then, you can use this returned String array as the argument for the `Country` constructor:

```
Country country = new Country(line.split(","));
```

Add some testing code in `main` (e.g., print out the 2nd country in the list), and then compile and run it by:

```
javac CountryDisplayer.java
java CountryDisplayer CountryDataset.csv
```

Note that your program shouldn't try to create a country out of the first line, as the first line is the header.

Step 4: Displaying the countries

Next, implement the method `displayTextCountries()` that prints out the country list, from the first country to the last country. Calling this method will also help you see the result of sorting in Step 5 (so to help you test Step 5).

Modify `main` so that you can have the option of testing this functionality by passing in **two** command-line arguments. Then, compile and run it by:

```
javac CountryDisplayer.java
java CountryDisplayer CountryDataset.csv PopulationGrowth
```

(You could replace PopulationGrowth with any one of the eight feature names.)

The countries should appear in the same order as in the CSV file for now, because we haven't sorted them.

Step 5: Helper methods for `sortCountryList`

Modularity is especially important for sorting—you don't want to have completely different sorting methods for all of the different possible indicators (feature names), and we want to avoid duplicating code. Please implement the following two (private) helper methods (implement them first!) to make your `sortCountryList` method shorter and more readable.

Helper 1: getDataEntry

```
/**
 * returns the data entry of the Country at the specified index in the list, and with the specified column
 */
private double getDataEntry(int index, String columnName) {
}
```

For example, `getDataEntry(10, "PopulationGrowth")` would return the population growth of the 11th country in the list (indexing starts with 0 for lists in the Java Class Library). Here's the place where you want to call the `getFeature` method of `Country`. You can implement this method using only one or two lines. I made this method "private" because I don't intend to allow clients of this class to use it—it's for internal use only.

Helper 2: findMaxIndex

```
/**
 * returns the index of the country in the list with the maximum value (in terms of columnName).
 */
private int findMaxIndex(String columnName) {
}
```

Depending on the logic design, you may or may not use `Double.isNaN()`, and `-Double.MAX_VALUE`, the smallest value `double` can represent. You also may or may not need to be very careful when comparing NaNs (here's a resource in case you need to use it for debugging).

Step 6: Implementing `sortCountryList`

This method should modify the data field (it has side effects). Here's the pseudocode:

```
eclare and "new" a list, called sortedCountryList. It is currently empty.
hile countries (data field) is not empty:
    2.1 find the index of the country in countries that has the maximum value in terms of columnName (call findMa
    2.2 add this country to sortedCountryList
    2.3 remove this country from countries
ssign sortedCountryList to countries
```

Call and test this method in `main`; please test your code thoroughly.

Bonus Part

Description

For the bonus part, the code will also use 1) a library for producing graphs called JFreeChart, and 2) one additional class that I provided to make it easy for you to interact with the graphing library:

1. jfreechart-1.5.0.jar
2. BarChart.java

In the same folder, you also have the Javadoc for the `BarChart` class under `/javadoc/BarChart.html`. **You should not make any changes to these files.**

Your program (`CountryDisplayer`) should produce output based on whether two or three command-line arguments are present. If two arguments are present, it should behave just like in the above assignment; if three arguments are present, the program should display a graph (see more below). The third argument should be an additional indicator (feature name) that we also want to display; this should be from the same list of indicators represented above.

Special Notes

We're using a separate Java library to do the graphing, which means that we'll need to tell the compiler and the Java virtual machine (JVM) about that library. Specifically, we have to tell it to include those classes on the `classpath`—the set of classes that the program has access to (see this StackOverflow question and answer for a little more info). You'll do this by adding a classpath argument when you compile and run the program. So, when you want to compile (on a Mac), navigate to the directory that contains your code, which should also contain `BarChart.java` and `jfreechart-1.5.0.jar`, and type:

```
javac -classpath .:jfreechart-1.5.0.jar *.java
```

That tells the compiler to include the current directory (.) and the classes in the jar file when compiling. `*` is a wildcard symbol, and `*.java` means to compile all the files with the extension `.java` in this directory.

Then to run your program, type:

```
java -classpath .:jfreechart-1.5.0.jar CountryDisplayer
```

and **put your other command-line arguments after CountryDisplayer**. For example,
`java -classpath .:jfreechart-1.5.0.jar CountryDisplayer CountryDataset.csv PopulationGrowth` will display countries as texts, sorted by PopulatiionGrowth, and
`java -classpath .:jfreechart-1.5.0.jar CountryDisplayer CountryDataset.csv PopulationGrowth UrbanPopulationGrowth` will display a graph.

Note: If you're on Windows, replace ".:" with ".;". If on your own machine (any operating system), you get a message about Unsupported Major/Minor Version error, it means you're running a different version of Java than the jar was compiled with. Use the lab machines or change to the newest JDK.

Graph Display

If there are three command line arguments, your program should provide a graph of the top 10 countries based on the sorting by the first provided indicator (the second argument). Sorting should follow the same rules as for the text display. Your graph will include two series: one has the data for the first indicator you sorted by, and the other has the data for the second indictor that was provided on the command line (as the third command line argument). For example, the graph below is what gets displayed when I run the following line:
`java -classpath .:jfreechart-1.5.0.jar CountryDisplayer CountryDataset.csv PopulationGrowth UrbanPopulationGrowth`
(If the graph is not shown here, please see the attached graph called HW4_GraphOutputExample.png)

Step 7 (bonus): Implementing `displayCountryGraph` method

Add a public method `displayCountryGraph`, which will be called in `main` if three command line arguments are present.

```
    public ?? displayCountryGraph(??) {
        // ...
    }
```

My `displayCountryGraph` has fewer than ten lines of code. You'll create an instance of `BarChart` for your graph, and you'll add data to a particular series using `addValue(String country, double value, String series)` method of `BarChart`. Look at the Javadoc (`/javadoc/BarChart.html`) or the code to find out more about how to create and display a bar chart. **Hint:** Bar labels are the country names (e.g., Oman, Qatar, ..., Angola), and "series" is the legend items (e.g., PopulationGrowth, UrbanPopulationGrowth). While the `BarChart` class might look complicated at first, you should only need to use a few methods: a constructor, the `addValue` method, and `displayChart`. Take a look at the javadoc to learn how to use these methods, and experiment if you're not sure! As mentioned before, you'll create an instance of `BarChart` for your graph, and you'll add data to a particular series using `addValue(String country, double value, String series)` method of `BarChart`.

Modify `main` so that you can have the option of testing this functionality by passing in **three** command-line arguments.

As noted above, to compile all of the files in a folder at one time and include the jar file with the graphing classes, you can run `javac -classpath .:jfreechart-1.5.0.jar *.java`. The "*.java" means anything that ends with ".java".

If the user doesn't properly type in an indicator as the second command-line argument, or if they have a third command-line argument that isn't one of the indicators, display a helpful message for them about how to use your class—this is called a usage statement. See the last line of `GuessingGameSolution.java` as an example of displaying a usage statement.

Submission Guidelines

Your program must consist of these two Java files `CountryDisplayer.java` and `Country.java`. Optionally, include `AnalysisDiscussion.txt`, describing your reflections about the results. Do *not* submit any `.class` files for the classes that you write, and also do not submit `BarChart.java` or the `jar` file that was provided.

To prepare to submit all your files, put the above two (or three) files in a directory named `HW04`, and then zip up that directory. Double-check your zip file: copy it to somewhere else, unzip it, copy over the files that I provided to you, and try to compile and run your code. Also, make sure to not include extraneous files or directories. When you're satisfied that your submission is ready, upload this zip file and `Collaborations.txt` to Blackboard.

Start early, ask lots of questions, and have fun!

Assignment requirements

This is a partial list of the things that we'll be looking for when evaluating your work:

- Correct command-line syntax and behavior (including naming the main class `CountryDisplayer`)
- Program compiles
- Program runs correctly on test input
- CSV File parsing
- Sorting according to criteria described above
- Correct formatting of output
- Correct `Country` class implementation
- Code uses the ADT implementation correctly: the name of the class implementation list (e.g., `ArrayList`) appears only in the initialization of `List` variables, and then only on the right-hand side (just as in the example code from class).
- Two @author tags each with a member's full name, in both Java files
- Good coding style
- (optional bonus) Use of the `BarChart` class to display a graph of your analyses

This assignment is worth 30 points (5 of which is for good style). The bonus part is worth an additional 5 points.