Assignment 3 -- Container Classes

Reminder: Project Hand in Rules

Read and follow the instructions!

1. All project due dates are specified on Brightspace, it is the final say on when the project is due.
2. All projects handed in must be NetBeans projects or include a README for instructions on how to run you project from the command line
3. If you hand in ANYTHING (even a blank corrupted file) before the due date you are eligible to take advantage of the late policy as outlined on the syllabus
4. For projects, you will hand in what is specified by individual assignments. Most of the time this will be your entire project as a .zip archive (other compression formats may be accepted, but slow down the grading process). Name the archive with ONLY your last name, as an example I would submit `Droppo.zip` to Brightspace as my submission.
5. All work is subject to the collaboration policy from the syllabus. As a reminder, violation of the collaboration policy may result in an automatic failure of the course.

Background

Publicly traded stocks are one of the many investment instruments available to people who have the ability to spend money on it or have received them as part of their contracted work or bonuses during their careers. Although they can be volatile in a short period, over periods of many years (historically) the stock market outperforms many other alternative investments[1] as measured by the percentage increase in value.

For this project, you will be reading in a data file that contains a number of stock price quotations for a collection of companies, calculate the statistics for the data read in, and display the results in a formatted output resembling a table.

Project Kit:

You Provide

- A new blank java project

I Provide

- PortfolioManagementKit
  - out#.txt -- Example outputs for each test
  - StockPortfolioManagement*.class.violet.html -- The UML diagrams of a complete project with a B Grade variant
  - test#.txt -- The testing inputs that correspond with similarly numbered out#.txt files
  - UnitTests.txt -- Main method unit tests for the `Stock` and `Portfolio` classes
  - implementationNotes.txt -- Implementation notes that might help with parsing and calculating the longest upward trend. (This is an artifact from the base assignment)
  - instructions.txt -- A short (optional) breakdown of broad steps that can be taken to solve this project.
  - ReadMeFirst.txt -- A short README .txt file outlining the parts of the project (similar to how I've done here).

The out files and the test files should be put into the root directory of your project and you should copy the unit tests from UnitTests.txt into new java classes for `Stock` and `Portfolio`

Input Data File Format

Each filename MUST be passed in as a system argument.

The input file format is a sequence of paired lines, with the first line holding the name of the company and the second line representing the share price quotations (pq_n) over some number of periods[2]. For example from `test1.txt`:

```
A
15 25 35 45 55 65 75 85 95 105
```

In general this means that the structure will be:

```
<company name>
<pq0> <pq1> ... <pqn-1>
```

Given n price quotes.

NOTE: AN EMPTY FILE IS VALID INPUT[3]

Rules Defining the Price Quotes

- Each company will have at least one price quote
- Each company inside the same file will have the same number of price quotes
- Different files might have different amounts of price quotes

- All price quotes will be positive

Rules Defining the `Stock` and `Portfolio` Classes

- `Stock` models a single stock[4].
- `Portfolio` contains an `ArrayList<Stock>` called `stocks` and models the file being read in.
- The methods required for an `A` grade are done in `Portfolio` and not `Stock`.
- Anything you find yourself calculating multiple times should be made a field with the appropriate access modifier.
- Each statistic should be stored as a field of the appropriate class.
- Each statistic that takes more than one line to calculate should be done in a `private` method called by the appropriate constructor.

Your Task

In the files provided you should find a UML diagram of each class' fields and methods[5]. It is your job to implement these methods and classes. Below I've outlined a portion of the project that should serve as a minimum viable product. I go into more detail about specific portions of the assignment that might provide difficulty in Hints and Notes.

`Stock.java`

1. **Low Price**: The minimum price given for the company's shares
2. **High Price**: The maximum price given for the company's shares
3. **Net Change**: The difference between the last price and the first price.
4. **Average Price**: The average price for the whole sequence. Calculated by summing all prices and dividing by the number of prices.
5. **Standard Deviation of Prices**: The square root of the average difference between each price squared and the average price squared. This is calculated $\sqrt{\Sigma_{i=0}^{n}((pq_i - pq_a)^2)/n}$ where n is the number of prices and pq is each price quote.
6. `ToString()` : A string representation of the statistics for the stock that follows the rules listed in Output Table Rules ***THERE SHOULD NOT BE A NEWLINE AT THE END OF THE STRING RETURNED BY THIS METHOD***
7. **Unit Test** `Stock` : Everything up to this point, if implemented correctly, earns a `C` grade for this assignment. The final two tasks in `Stock` are only required for an `A`
8. **Longest Upward Trend (LUT)**: The longest string of trends such that $pq_{n-1} \leq pq_n$ holds for all price quotes in the trend.[6] More details can be found in Hints and Notes.

9. **Best Growth Rate**: The difference between the last and first price quote in the LUT divided by the length of the LUT.

`Portfolio.java`

1. `toString()` : Returns a string representation of the table outlined in Output Display.
2. **Unit test** `Portfolio` : Everything above should be implemented and unit tested for a `B` grade. You may wish to complete this before returning to `Stock.java` 's 8th and 9th tasks.
3. **Best Rate**: The maximum value of pq_{i+1}/pq_i across all price quotes and all stocks.
4. **Period of Best Rate**: The first index where the maximum value of pq_{i+1}/pq_i occurs.

Output Display

After calculating the values for each company, we want to display them as a table with labeled columns for the company name, and each of the statistics described in both `Stock.java` and `Portfolio.java`.

Output Table Rules:

- Each column should be a fixed minimum width of either 20 or 10 characters.
- Company names are left justified.
- Numerical values should be right justified.
- Double values should be rounded to the hundredth's place (two decimal points).
- If a statistic is undefined or if a statistic can't be calculated[7] then you must instead print `n/a` in place of the undefined or incalculable value.

This can be achieved exclusively through the use of `String.format()` , and you can find its documentation here. As a hint/help for how wide each column should be: each company name shouldn't exceed 20 characters and all double values will be at most 9 digits for a total character count of 10 characters.

The example headers in the out files are suggestions and you do not need to follow them exactly, but if you deviate from them you must still have a maximum column width of either 20 or 10 characters and wrap any header title to be multiple lines. For example "Best Upward Trend Rate" could become[8]:

```
Best Upward
Trend Rate
```

Grading Criteria:

Each requirement below is graded as all or nothing. Grading for this project will be done in the order listed in the table below:

Grade Earned	Requirements
D	`main` method(s) are the only methods with the `static` modifier in any class. `main` method in the driver checks for command line arguments. Usage message is correct for the number of required arguments. `main` method in the driver creates an object of the driver class and calls a run method. All code handed in is syntactically correct -- the driver compiles and runs.
C	Everything above plus: `Stock` class is correctly written with methods in IPO[9] order. Correctly calculates the first 5 statistics. `Stock` class `toString` returns the strings formatted according to the rules in Output Table Rules and does NOT end with a linefeed. `Stock` passes all unit tests provided in `UnitTests.txt` for the `Stock` class.
B	Everything above plus: `Portfolio` class is correctly written with methods in IPO order. `Portfolio.toString()` returns the formatted output of all stocks in the portfolio and the table's headers in accordance to the stipulations in Output Display such that each stock receives its own line in the output table.
A	Everything above plus: `Stock` class correctly calculates the Longest Upward Trend. `Stock` correctly calculates the best growth rate. `Portfolio` correctly calculates the period of best rate.

What to Hand In

Your zipped project archive following the stipulations in Reminder Project Hand in Rules.

Hints and Notes

In this section I've collected sections I think should be highlighted from the .txt files in the .zip archive I provided[10] as well as suggested hints for implementing different aspects of this project. These may involve breaking away from the outlined design of the project, but do not fundamentally change the output or skills being exercised.

Hints for `Stock`

- `trends[]` is an array of doubles according to the UML. I've found it's much easier if `trends[]` holds the indexes for the start of each upward trend, making this actually an

`int[]` and not a `double[]`.

- `heightMultiplier` is where we want to store the maximum value found in the ratio across price quotes. As a mathematical expression this is: $\max(pq_{n+1}/pq_n)$ where pq_n is the n-th price quote.
- `highPeriod` is where I ended up storing the value of n for my `heightMultiplier`, making it an `int` and not a `double`.
- The calculation for `netChange` does not need to be done in `calcMinMaxNetAndAverage()` [11] and can be done in the constructor instead, but `minimum`, `maximum`, and `average` should be contained in a method.
- `getTrendAt` should return the pq_{n+1}/pq_n ratio for the given index

How to Populate the `trends` Array

I've pulled this from `implementationNotes.txt`, and made attempts to make this more clear; but due to the limits of text-based communication it may require you to learn some extra math symbology. I will do my best to explain with code equivalents or the footnotes immediately after each math block.

To make the LUT and Best rate calculations easier we are going to want an array of starting indices for each upward trend. Every element in this trend must satisfy the pattern $pq_n \leq pq_{n+1}$ or in english *a price quote that is less than or equal to the next price quote in the series*.[12] This calculation is easiest done if we use an auxiliary array of upward trend start indices.[13] I will be calling this array `trends`. Since there are at most n possible starting indexes where n is the number of price quotes, we can initialize `trends` to be the same size as our `prices` array.

We can then start filling our array with the starting indexes for each LUT. We know that the first trend ***MUST*** start at 0[14] we can initialize our zeroth index in `trends` as such:

```
trends[0] = 0;
```

From here on we can generalize an algorithm to get the k-th upward trend, as $k = 0$ is our only special case. The algorithm is as follows:

Suppose we have already seen the starts of upward trends $0, 1, \ldots, k-1$ and those slots of `trends` are loaded correctly and we are scanning for the end of trend $k-1$. If we look at the current value we are scanning `prices[i]` and compare it to `prices[i-1]` [15] we can know that one of two possibilities is the case:

$$\text{prices}[i] \geq \text{prices}[i-1]$$
$$\textbf{OR}$$
$$\text{prices}[i] < \text{prices}[i-1]$$

If the first case is true `prices[i]` is part of trend $k-1$ and we can continue our scan. In the second case is true then we've found the end of trend $k-1$ and i is the index at the start of upward trend k. In code this would be:

```
if (prices[i] < prices[i-1]){
        trends[k] = i-1;
        k++; // to advance to the next slot in trends
}
```

By maintaining k's value as we go, when we complete our scan of `prices` we can guarantee that k will be the total number of upward trends and `trends` will contain the start indices for each trend in the indices $[0, k-1]$.

How to Calculate LUT

Once we have the indices that start each upward trend stored in `trends` we can calculate its length by finding the value of

$$\text{trends}[i+1] - \text{trends}[i] - 1$$

For all values of i such that $0 \leq i < k-1$ and

$$n - \text{trends}[k-1] - 1$$

For trend $k-1$[16] where n is the length of the `prices`[17] array. The `calcLUT()` method should return the maximum length found in this way.

How to Calculate Best Growth

Once we have the indices that start each upward trend stored in `trends` we can find the amount of upward growth for that trend by finding the maximum ratio of net change to length of trend. As a mathematical expression this would look like:

$$\forall k \in \text{trends} : \max(\frac{pq_{k_{end}} - pq_{k_{start}}}{k_{end} - k_{start}})$$

Which might look scary, but in reality is simply saying:

```
maxVal = -infinity
for(each valid index in trends):
    upwardGrowth = prices[trends[index + 1]] - prices[trends[index]]
    upwardGrowth = upwardGrowth / (trends[index + 1] - trends[index])
    maxVal = max(upwardGrowth, maxVal)
```

A special exception must be made for the last trend, similar to the exception for the LUT calculation. Instead of using the next UT index from `trends` we will use n giving us instead:

```
upwardGrowth = prices[prices.length - 1] - prices[trends[index]]
upwardGrowth = upwardGrowth / (prices.length - 1 - trends[index])
maxVal = max(upwardGrowth, maxVal)
```

Hints for `Portfolio`

- `addStock()` and `getStockAt()` effectively wrap the `ArrayList` methods of similar purpose.
- If you've separated the statistics calculations correctly `calculateBestStocksByPeriod()` should run a maximum function for each period.

Hints for `PortfolioManagement`

- This is the driver class, `main` should either:
  - call `driver.readFile(args[0])` and then `driver.run()` without arguments[18]
  - call `driver.run(args[0])` and then inside `run` you call `this.readFile(filename)`[19].

---

1. citation needed ↩
2. Each period can be treated as one day for this assignment. ↩
3. See `test0.txt` ↩
4. The two line pair of company name and price quotes ↩
5. Both for a partial project and the complete project labeled as "extra credit" ↩
6. In the case where $n = 0$ the trend is pq_0 ↩
7. This may be the case for any statistics that divide by zero or don't have upwards trends ↩
8. I recognize that there are 11 characters in the first line, I won't dock points for improperly formatting your header unless it is ambiguous what column each header is referring to. ↩
9. Input Processing Output ↩
10. See I Provide ↩
11. It can be done in one line: `net = prices[prices.length-1] - prices[0]` ↩
12. **NOTE:** this means that if every price quote fails this inequality ***each*** price quote is it's own UT of size 0. If every price quote satisfies this relationship then we have a single UT of size $n - 1$ where n is the total number of price quotes. ↩
13. Hence my suggestion at the beginning of Hints for `Stock` . ↩
14. See the footnote in `Stock.java` 's requirement for LUT. ↩

15. This is partially why the 0th trend is a special case. It helps us enforce that $i \neq 0$ ↩
16. This is only noted this way because we're zero-indexing the start of each trend. ↩
17. or `trends` ↩
18. Yes, this breaks from the UML provided but this is an equally valid solution that would prevent file input errors from interfering with the `run` method. ↩
19. Where filename is the name of the `String` parameter for `run` ↩