STOCK LAB

(Working with object arrays)

Richard Bourquard
April 2018

Lab 1: Reading Data

DISCUSSION:

This lab will analyze a stock that has the symbol QQQ. Note that QQQ is a real stock that you could buy!

Actually QQQ is a fund, which means it follows a "basket" of stocks. QQQ's basket contains mostly high-tech stocks. When the stocks in the basket gain value, QQQ gains value. When they lose value, QQQ loses value.

This lab includes a data file that contains 19 years of QQQ's daily closing prices (that is, the price at the end of the day). There were 2 big "crashes" that occurred during this time period. The first (often called the "dot.com crash") happened in 2001 when stocks lost around 78% of their value. The second (often called the "Credit Crisis") happened in 2008 when stocks lost around 57% of their value. The market recovered both times, but it was very scary for investors each time.

QQQ prices are delivered from the internet in descending order (that is, the most recent date is first). Consequently the data file lists them in that order.

PROBLEM:

Write a program that reads the 19-year data file and loads its values into daily price objects that can be manipulated in the later labs.

As a data check, have the program print the number of dates it read (thus excluding the 1st row) and print the sum of the prices for the first 2,000 rows.

PROCEDURE:

The system you will build consists of 3 classes:

1. The *Main* Class contains the client program.
2. The *Stock* Class will have one object that holds information about QQQ.
3. The *DailyPrice* Class will have thousands of objects; each holding one date and the associated closing price.

The *Main* program will read the data file and create a QQQ *Stock* object. Then it will have that *Stock* object create over 4,000 *DailyPrice* objects, one for each of the daily prices. Thus, *Main* will have-a *Stock* object, and *Stock* will have-a bunch of *DailyPrices*.

PROGRAMMING STEPS:
Let's look at each Class in turn…

DailyPrice Class

Each object of this class stores the closing price for one day. It also stores a version of the date that is much easier to manipulate called a "hashDate".

- Write a constructor that will accept a *date* (String), *price* (double), and *hashDate* (int) – and store them in private fields. *Date* will have the String format of either:
  "dd-mmm-yy", or "d-mmm-yy".

Stock Class

An object of this class represents one stock (for this lab, that stock is QQQ). This object retains pointers to all the *DailyPrice* objects so it can manipulate them.

- Write a constructor that will accept a stock *symbol* (String) and store it in a private field.
- Write a *addDailyPrice* method that will accept a date and price, and create a new *DailyPrice* object using its constructor. It will call the *createHashDate* method described below to create the *hashDate* required by the constructor. The new *DailyPrice* object that is created should be added to an internal ArrayList. Note that this will be an array of objects!
- Write the private *createHashDate* method, which will convert the date (which is a String) into a hash number (an int). Use the formula: *hashDate = 10,000*year + 100*month + day*

  WHY ARE WE DOING THIS? While you could sort and search the DailyPrice *objects by using the date, it would be difficult and time consuming because the date is a String. Therefore, a date like "17-Sep-04" would end-up sorted* before *"5-Apr-99", which is incorrect!*

  A faster way to proceed is to create a simple number from the date. (For example, Microsoft uses a floating point number to store dates and times, but that's complicated to set-up.) Since we are only interested in days and not the time of day, a simple integer can be easily created for this purpose.

  One way to do this is to multiplex (combine thru multiplication) the year, month, and day into a single number. This number can then be easily used for searching and sorting. For example, they can be multiplexed using the following formula:

  *hashDate = 10,000*year + 100*month + day*

  where year, month, and day are integers derived from the String date. (For year, note that "99" will become 1999, whereas "00" thru "18" will become 2000 thru 2018.)

  For example, using this formula, the date "17-Sep-04" would become: 20040917 –and– "5-Apr-99" would become: 19990405. You will write the createHashDate method to do this conversion.

Main Class

The Main method will read an input file containing 19 years of daily prices for QQQ, create a *Stock* object, and then use that object to load the daily prices into *DailyPrice* objects.

- Write code to input a file called "QQQ-Prices.txt" which contains the 19 years of daily QQQ prices. Create a *Stock* object based on the 1st row of the file. As you read subsequent rows, call the *addDailyPrice* method you wrote to store its values. Print the total number of dates stored, and the sum of the first 2,000 rows.

  One simple method of reading a text file is given on page 404 of the "Building Java Programs" text. There are lots of other examples on the internet.

  *FOLLOWING IS THE FORMAT OF "*QQQ-Prices.txt"…

  The first row of the file contains the symbol for the stock (in this case, "QQQ"). Call the Stock *constructor with that value.*

  Each subsequent row of the file contains 2 values: a date and a corresponding closing stock price, separated by a single tab character. The format for the date is day, month, year; each value separated by a hyphen. For example: "23-Dec-14" or "3-Jan-02". The format for the price value is double. For example: "28.13". Therefore, a typical row might be "17-Mar-08<tab>98.47". There will be over 4,000 rows.

QUESTIONS:

How many dates of data did you read and store? What was the sum of the first 2,000 prices?

Lab 2: Sorting

DISCUSSION

To be useful, the stock prices must be sorted into ascending-date order, because subsequent operations depend on calculating the change in price between one day and the next.

The input file is sorted in descending-date order, just as it was retrieved from the internet. That order won't work for day-over-day calculations. You must sort the data into ascending-date order.

PROBLEM:

Reverse the order of the data that was stored.

As a data check, print how many date values are now in the array, and print the sum of the first 2,000 price values.

PROCEDURE:

This can be done by sorting the data in order of ascending date. Note that *hashDate* provides a convenient int value for sorting.

PROGRAMMING STEPS:

Let's look at each Class in turn…

DailyPrice Class

- Write appropriate accessor methods to retrieve the *hashDate* and the *price*.

Stock Class

- Write a sorting method to sort the *DailyPrice* objects into increasing order by date. Use the *hashDate* field as retrieved from each DailyPrice object as the substitute for the date.

Main Class

- Sort the prices by calling the sort method you just wrote.

QUESTIONS:

How many dates of data did you read and store?

What was the sum of the first 2,000 prices?

How will you determine if the sort worked properly?

Lab 3: Review the Data

DISCUSSION:
During the 19 years of this data set, the stock market has suffered 2 catastrophic crashes; one in 2001 (often called the "dot.com bubble") and a second in 2008 (often called the "credit crisis"). (You may wish to ask your parents how these two crashes affected them!) This lab will investigate how the crashes affected the value of QQQ.

PROBLEM:
Find the price of QQQ at these significant dates:

1. March 10, 1999 - The oldest date in the file
2. March 24, 2000 – the highest share price reached before the 2000 crash.
3. October 7, 2002 – the lowest price reached after the crash.
4. December 26, 2007 – the highest price reached before the 2008 crash.
5. March 9, 2009 – the lowest price reached after the crash.
6. April 25, 2017 – My birthday last year.
7. March 29, 2018 - The most recent date in the file

PROCEDURE:
Add a method that can search and retrieve a price for a given date.

PROGRAMMING STEPS:
Let's look at each Class in turn…

Stock Class

- Write a search method that for a given date (String = "dd-mmm-yy") will return the price of QQQ. For that given date, you will need to call the *hashDate* conversion method you wrote in Lab 1 to find the proper *hashDate* to search for.

Main Class

- Call the Stock Class search method you just wrote to find the 7 QQQ prices. Print the dates and prices.

QUESTIONS:
How significant were the 2 crashes (the crash between Mar 24, 2000 and Oct 7, 2002 –and– the crash between Dec 26, 2007 and Mar 9, 2009). How would you have reacted if you had had your savings for college invested?

Lab 4: A Buy-and-Hold Strategy

DISCUSSION:
One popular investment strategy is to buy a stock and simply hold it "forever". See what return you would have gotten by simply buying and holding one share of QQQ for the entire 19 years. This is called a "buy and hold" strategy. It assumes you would have valiantly held on to your shares and not sold them in desperation even through the 2 horrible crashes.

PROBLEM:
Compute the return on investment (ROI) for a buy-and-hold strategy on QQQ over the entire 19 years.

PROCEDURE:
Add code to compute the buy-and-hold ROI.

PROGRAMMING STEPS:
Let's look at each Class in turn…

Stock Class

- Write the method, *buyAndHoldROI*, that returns the total ROI (double) for having bought one share of QQQ on the first day and kept it until the last day.

  Traverse the array of prices <u>in ascending date order</u>, adding each day's return to the previous day's return, until you get to the last day. That last day is the ROI.

  Since you are dealing with percents*, however, the formula is not a simple addition. It is a running accumulation. The formula for the total return on any given day is:*

  runningTotalReturn[for today] =

  ((todaysPrice/yesterdaysPrice) * (runningTotalReturn[from yesterday]+1)) – 1

  where yesterday *and* today *traverse day-by-day through the entire data set in ascending order of date.*

  Note: for the 1st day in the array there is no "yesterday"; so set:

  runningTotalReturn[1st day] = 0.

Main Class

- Write code to call the *buyAndHoldROI* method and print the ROI.

QUESTIONS:

- What ROI did you get with the "buy and hold" strategy?

  Note - This is the return (increase) you would have received by buying one share of QQQ and holding it over the entire 19 year period. For example, if you bought an initial share for $100 and at the end of 19 years it was worth $200, you would have doubled your money. The ROI value would = 1. That is, the return (increase) was 1 times what you started with. If it was worth $250, the ROI would = 1.5, and so forth.

- Did your return approximately match the percent difference in prices between the first and last prices found in Lab 3?
- Would it have been worth holding the stock through the 2 crashes for that amount of return?

Lab 5: A Crash-Avoidance Strategy

DISCUSSION:
Stock market crashes (there are 2 in this data series) are very costly. It would be great if you could just avoid them entirely! This would require selling your shares somewhere around each market top, and buying back-in somewhere around the bottoms. You can only make this happen if you have an algorithm that tells you where the top is and where the bottom is. Since the market prices bounce around a lot even in normal times, it's really difficult (many say impossible) to determine tops and bottoms.

This section presents a simple algorithm that does not determine tops and bottoms, but rather determines when a crash is already happening! Thus it tells you to get out sometime *after* the top. It also determines when the crash is over, and thus when to buy back in. So this algorithm isn't perfect, but it can avoid a lot of damage.

PROBLEM:
Program this algorithm to see what return you would have gotten by simply **not** holding the stock during each entire crash. The algorithm determines if the market is crashing, and sells the stock when it is. When the crash is over, it buys the stock back again with the sale proceeds.

PROCEDURE:
A first traverse is made through the data to determine the extent of any declines in price. This is done by comparing each day's price to the maximum price found over the prior 20 days. Each decline is stored within a new field of *DailyPrice*.

Then, a second traverse is made to flag those declines which are extreme and thus represent a market crash in progress. This is done for each day by seeing if the average of the prior 20 days' declines exceeds (is worse than) -8%. If it does, a crash is in progress. A Boolean flag representing this crash status is stored within a 2nd new field of each DailyPrice.

A new ROI is computed where the stock is not held during these "crash" periods delineated by the Boolean flag.

PROGRAMMING STEPS:
Let's look at each Class in turn…

DailyPrice Class

- Add 2 additional private fields: "*pctPriceDrop*" (double) and "*doingCrash*" (boolean). Add appropriate mutator and accessor methods to put and get these values.

Stock Class

- Write the private method, *computePriceDrops,* that traverses the array of prices *in ascending date order*, and computes how much QQQ's price has fallen from the highest price found within the last 20 days.

  Therefore, for each date in the array…

1. Find the maximum price over the preceding 20 days. Call this *maxPrice*.

   (Note, for the 1st 20 days in the array, less than 20 days are available to be used.)

2. Find the % difference between this maximum price and the current price. Use the formula:

   pctPriceDrop = 100 * (todaysPrice/maxPrice -1)

   (Note that *pctPriceDrop* will always be zero or a negative number.)

   (Also note that *pctPriceDrop* = -10.0 would mean a decline of 10%.)

3. Save this number in today's *DailyPrice* object as *pctPriceDrop*.

- Write the private method, *findCrashes,* that traverses the array of prices *in ascending date order,* and determines whether QQQ is currently crashing. This method assumes that the *computePriceDrops* method has already run.

  Therefore, for each date in the array…

  1. Find the average of the previous 20 *pctPriceDrop* values.

     (Note, for the 1st 20 days in the array, less than 20 days are available to be used.)

  2. Determine if a crash is happening. Use the Boolean formula:

     doingCrash = (the average *pctPriceDrop* value) < -8.0

     (A crash is happening every day that *doingCrash* is True.)

  3. Save this number in today's *DailyPrice* object as *doingCrash*.

- Write the method, *skipCrashROI*, that returns the total ROI (double) for having bought QQQ on the first day, sold it during any crashes, and bought it back again after each crash. The method will do the following:
  1. Call the computePriceDrops method.
  2. Call the findCrashes method.
  3. Traverse the array of prices in ascending date order, accumulating each day's return until it gets to the last day.

     Use the same formula for the total return developed in Lab 4, **EXCEPT** if doingCrash then QQQ is not being held, and (todaysPrice/yesterdaysPrice) would be 1. Thus an if statement is required in the computation…

```
if (doingCrash)
{
   //this is a crash.  QQQ is not being held.
   //The runningTotalReturn doesn't change.
   runningTotalReturn[for today] = runningTotalReturn[from yesterday]

} else {

   //this is NOT a crash.  QQQ is being held.
   //Compute the new runningTotalReturn
   runningTotalReturn = ( (todaysPrice/yesterdaysPrice)
      * (runningTotalReturn[from yesterday]+1) ) - 1
}
```

(Note for the 1st day in the array, there is no "yesterday"; so set:

runningTotalReturn[1st day] = 0.)

Main Class

- Write code to call the *skipCrashROI* method, and print the final return.

QUESTIONS:

What ROI did you get with this strategy?

> *Note: This is the return (increase) you would have received by buying 1 share of QQQ and holding it only when the* findCrashes *method determined that QQQ's price was not in a crash. As in Lab 4, the return here represents the increase from what you started with. For example, a value of 1 indicates you have doubled your money.*

Compare it to the 'buy and hold' return of Lab 4. Which is better? Would you feel better about not holding QQQ during a crash?

Are we in a crash right now?

If you wanted to investigate this strategy further, you might want to see when and how often it picked crashes. Perhaps it is "trigger happy" and is flagging "crashes" that don't turn out to be real crashes. How could you determine when *findCrashes* calculated there was a crash? How could you determine whether each crash found by *findCrashes* was actually a period during which QQQ lost value?