CIS*2430 (Fall 2021) Assignment One

Instructor: F. Song

Due Time: October 18, 2021 by 11:59 pm

In this course, you will be working on a project called "ePortfolio", which will be built incrementally over the semester through three individual assignments. In addition to the implementation, you are also required to follow proper coding styles and establish good habits for testing and documentation.

General Description

An investor often needs to maintain a portfolio of different investments so that the person can keep track of the actions for buying or selling investments, searching for relevant investments, updating prices, and calculating the total gain of the portfolio. An investment is better modeled by an object so that you can distinguish different attributes and apply suitable methods for accessing and changing these attributes. For this project, we limit ourselves to two kinds of investments: Stock and MutualFund. Here are two examples illustrating the related attributes for Stock and MutualFund objects, respectively:

Attributes	Stock Example	MutualFund Example
symbol	AAPL	SSETX
name	Apple Inc.	BNY Mellon Growth Fund Class I
quantity	500 shares	450 units
price	$142.23	$42.21
bookValue	$55,049.99	$23,967.00

Each investment should have a symbol, a name, a quantity, a price, and a bookValue. For each stock, we need to pay commissons when buying or selling shares of the stock, which will affect the values for "quantity" and "bookValue", and the calculation of the gain for the stock. We can assume that the commission is $9.99 each time we buy or sell certain shares of a stock. For example, if we initially buy 500 shares of AAPL stock at the price of $110.08 per share, the quantity will be 500, and the bookValue will be 500 × 110.08 + 9.99 = $55,049.99. Later on, if the price is changed to $142.23 per share, the gain will be (500 × 142.23 – 9.99) – 55,049.99 = 71,105.01 – 55,049.99 = $16,055.02. Alternatively, if we sell 200 shares of this stock at $142.23 per share, the payment received will be 200 × 142.23 – 9.99 = $28,436.01, the quantity will be reduced to 300, and bookValue will be adjusted to 55,049.99 × 300/500 = $33,029.99.

For each mutual fund, we do not pay any fee if we buy certain units of the fund, but if we sell certain units of the fund, we need to pay a redemption fee of $45. For instance, if we initially buy 450 units of SSETX at $53.26 per unit, the quantity will be 450 and the bookValue will be 450 × 53.26 = $23,967.00. Later on, if the price goes down to $42.21 per unit, the gain will be (450 × 42.21 – 45.00) – 23,967.00 = 18,949.50 – 23,967.00 = –$5,017.50. Alternatively, if we sell 150 units of this fund at $42.21 per unit, the payment received will be 150 × 42.21 – 45.00 = $6,286.50, the quantity will be reduced to 300, and the bookValue will be changed to 23,967.00 ×300/450 = $15,978.00.

In a portfolio, one may own multiple stocks and/or mutual funds, and as a result, we need to use two ArrayLists to store them: one for all stocks and the other for all mutual funds. One big advantage of using ArrayLists is that they are dynamic so that their sizes can grow or shrink as needed when one buy or sell investments. Given a portfolio, we can perform different operations and for this project, we will implement five of them:

(1) *buy*: own a new investment or add more quantity to an existing investment.

(2) *sell*: reduce some quantity of an existing investment.

(3) *update*: refresh the prices of all existing investments.

(4) *getGain*: compute the total gain of the portfolio by accumulating the gains of all individual investments.

(5) *search*: find all investments that match a search request and display all attributes of these investments. A search request may contain up to three fields: a symbol, a set of keywords that appear in the name of an investment, and a price range (with a pair of lower and upper values for the price). For example, a search request may just contain a symbol (e.g., AAPL) and in this case, only the investment with this symbol will be returned. If a search request contains the keywords "Growth Fund", then all investments whose names contain these keywords will be retured. If a search request contains more fields, e.g., "AAPL" as the symbol and "10.00-100.00" as the price range, then simply matching the symbol AAPL is not enough; the price of the investment should also fall within the given price range. There are several special cases for the price range: e.g., "15.00" means exactly $15.00; "10.00- " means $10.00 or higher; and "-100.00" means $100.00 or lower. For any of these fields, if the input is empty, it will match any value in the corresponding attribute of an investment. At the extreme, if the user leaves all these fields empty for a search request, then all the investments will be matched and returned in the search result.

Specific Requirements for Assignment One

Assignment One will build a basic system for "ePortfolio" and the remaining two assignments will add further enhancements. More specifically, you are asked to implement the following tasks for this assignment:

(1) A command loop that accepts any one of these commands: *buy*, *sell*, *update*, *getGain*, *search*, and *quit*. For the *buy* command, the user needs to enter the kind of investment (either stock or mutualfund), followed by the symbol for the investment. This symbol allows us to check if an investment with the same symbol of the given kind exists in the system. If so, we need to get input for new quanity and price; otherwise, we should also get the input for name. Note that bookValue is calculated based on the quantity and price of a new purchase along with the commission if relevant, and if the new purchase is adding to an existing investment, the new bookValue will be added to the existing bookValue for the investment along with the adjustments for quantity and price. For *sell*, the user needs to provide a symbol, an actual price,

and a quantity value. Note that selling is only possible if the investment exists and the available quantity is greater or equal to the requested quantity, and if the remaining quantity is greater than 0, we will also adjust the bookValue; otherwise, we will delete the investment from the portfolio. For *update*, we simply go through all existing investments and ask the user to enter the new prices. For *getGain*, we will calculate the total gain for all the individual investments based on their current prices. For *search*, the user needs to provide values for up to three fields: symbol, keywords for the name, and a price range. Note that the user can enter an empty string (i.e., no value) for any or all of these fields. The *quit* command simply exits the command loop and then terminates the program.

(2) As described earlier, you should use two ArrayLists to store the investments: one for stocks and the other for mutual funds. When adding an investment to the corresponding list, you need to check the list to see if an investment with the same symbol already exists. If so, just increase the quantity, update the price, and adjust the bookValue for the existing investment; otherwise, create a new investment with the given information. Since all investments should have unique symbols, we should also check the other list to make sure that no existing investments have the same symbol as the one for the new purchase.

(3) Based on the description so far, your implementation should include at least three classes: Stock, MutualFund, and Portfolio. Both Stock and MutualFund classes can have the same set of attributes, including symbol, name, quantity, price, and bookValue, but they can have different methods to compute the bookValue, payment received, and gain of an investment. The Portfolio class requires more work, such as maintaining two ArrayLists for stocks and mutualfunds for buying, selling, updating, and computing the total gain for the related investments, searching the lists sequentially for the matched investments, and displaying the result on the screen.

(4) For each search request, you typically need to scan through both lists for stocks and mutuals: for the price range, you need to make sure that it covers the price of each matched investment; and for the search keywords, you need to make sure that they all appear in the name attribute of an investment, although they can be in a different order. Note that when matching two keywords, they have to be equal at the word level but the cases can be ignored. For example, “Bank” and “bank” are a match, but “Bank” and “Banking” are not. Also, the keywords “Toronto Bank” is matched by the name “Bank in Toronto”, since the search keywords can be matched in any order as long as they are all matched within the name.

(5) For all the class definitions, you need to follow the conventions by making all the instance variables private and providing suitable accessor and mutator methods. Also, as much as possible, you should provide suitable constructors and those commonly expected methods for a class, including the “equals” and “toString” methods. Any methods that work on an individual stock or mutualfund should be defined in its corresponding class, but any methods that work on a list of investments should be defined in the Portfolio class.

(6) You should organize all of your classes in one package called ePortfolio and use Javadoc to create a set of external documents so that a TA can examine the API’s of your package directly during the marking process.

Deliverables:

All implementations should be done individually in Java, and your program will be marked for both correctness and style. By correctness, we mean that (1) your programs are free of syntactic and semantic errors and can be compiled successfully; (2) your programs are logically correct and without runtime errors; and (3) your programs should give appropriate runtime messages (also called prompts) and are reasonably roboust in handling user input. To make sure that your programs are logically correct and without runtime errors, you need to prepare a test plan along with a set of test cases. The test plan should describe all the major steps used in testing your program and cover all possible conditions for testing. For example, to search an element on a list, we need to consider a situation where the element is not on the list, and if the element is on the list, whether it is located at the beginning or at the end or in the middle of the list. All test cases follow trivially from the test plan and provide concrete examples that can be used for testing with the corresponding input and output values. To ensure that your programs are reasonably roboust in handling user input, you need to exercise defensive programming. Although you can clearly show a prompt asking for a certain kind of input (e.g., “quit”), the user may still enter a shorter input (such as “q” or “Q”) or something quite different (e.g, “bye” or “exit”). Your program should accept most of the reasonable values (such as "quit", "Quit", "QUIT", “q” or “Q”), but reject all the irrelevant values (such as “bye” or “exit”). Furthermore, for the irrelevant values, you should give the user a feedback message and ask the user to re-enter the required values.

A good style in programming allows people to understand and maintain (modify or extend) your programs easily. This often includes (1) meaningful names and well-indented layout for control structures (branches, loops, and blocks); (2) internal documentation (the various comments within your programs); and (3) external documentation (additional description outside your programs, usually the README file). In Java, different naming conventions are used for representing classes, variables and methods, and symbolic constants. There is no limit about the length of an identifier; so try to use meaningful names for the benefit of readability. For internal documentation, Java introduces “Javadoc”, which will automatically turn comments in /** some comments here */ or /* some comments here */ before public classes, public instance/class variables, and public instance/class methods into a set of html documents, like what is shown in a Java API on a web browser. In addition to Javadoc comments, the traditional comments can still be used to explain the meanings of local variables and highlight the major steps within a particular method. For external documentation in the README file, you can describe (1) the general problem you are trying to solve; (2) what are the assumptions and limitations of your solution; (3) how can a user build and test your program (also called the user guide); (4) how is the program tested for correctness (i.e., the test plan should be part of the README file); and (5) what possible improvements could be done if you were to do it again or have extra time available.

In summary, a complete submission should include: (i) a README file; (ii) all the Java source files; (iii) JavaDoc files; and (iv) all the data files if relevant. All the files should be organizaed as follows and any violations to these requirements are subject to some mark reductions:

(a) Create a new folder named <userid>_a<#> and include all the related files into this folder, which typically contains a README file, a javadoc folder, a package folder with all *.java files, and possibly some data files. For example, since my central email address at the

University of Guelph is “fsong@uoguelph.ca”, my userid is “fsong” and my folder name for the submission of Assignment One should be “fsong_a1”. Please do not include any unrelated files in your submission, and in particular, do not include any compressed files (e.g., those ended with the suffixes: .tar, .tar.gz, .tgz, .zip, .rar) in this folder since they will complicate the process for testing your submissions on the Sneakoscope system.

(b) On Mac or Linux machines, the recommended utilities for combining and compressing your submissions are “tar” and “gzip”. First, do “tar –cvf fsong_a1.tar fsong_a1” to create the file “fsong_a1.tar”, and then do “gzip fsong_a1.tar” to create the file “fsong_a1.tar.gz”. You can also combine the two steps together by doing “tar -czvf fsong_a1.tgz fsong_a1”. On Windows, you can use either zip or rar utility to create the corresponding “fsong_a1.zip” or “fsong_a1.rar”. Please do not rename your compressed file afterwards, for example, changing “fsong_a1.rar” to “fsong_a1.tar.gz”. The use of a suffix helps other people to uncompress your files properly, and renaming them defeats the purpose. As a result, if your files are created with “tar” and “gzip”, use the suffix “.tar.gz” for your compressed file or the suffix “.tgz” if you combine the two steps together; and if with “zip”, use “.zip”; and if with “rar”, use “.rar”. Any other combinations will be violations to the rule.

(c) After uploading your compressed file to the relevant dropbox on our CourseLink account, please also verify if your submission is indeed uploaded successfully by downloading and decompressing it on your local machine so that you can examine the contents of your submission for correctness.