CIS*2430 (Fall 2021) Assignment Two

Instructor: F. Song

Due Time: November 8, 2021 by 11:59 pm

In Assignment One, you should have created at least three different classes (Stock, MutualFund, and Portfolio) and are able to buy/sell investments, update prices, get the total gain, and search for the relevant investments. In Assignment Two, you are asked to simplify your code and add some improvements. In particular, you will create a super class for all stocks and mutual funds, and reduce the two ArrayLists into one in order to minimize the code redundancy. Additionally, you will load the existing investments from a file and save all investments (existing plus new in the ArrayList) back to the file every time you run your program. Finally, you will create a HashMap for the name keywords so that the search performance can be greatly improved.

Specific Requirements for Assignment Two

(1) Create a super class *Investment* that contains *Stock* and *MutualFund* as its subclasses. Note that all the common members of the subclasses should be pushed into the super class so that they can be inherited by the subclasses. Any other members such as the overriding methods and constructors should be defined in the corresponding subclasses. All of these changes should help simplify the two subclasses of *Stock* and *MutualFund* considerably due to the code reuse of inheritance.

(2) Reduce the two ArrayLists for stocks and mutualfunds to one ArrayList so that you can keep all the invements in one place. The new list should have the type "ArrayList<Investment>" and will help simplify your code for accessing the relevant investments considerably in your Portfolio class. In particular, checking if an investment with the given symbol exists or not can now be done in the one ArrayList for both stocks and mutualfunds so that no two different investments can have the same symbol.

(3) Expand your code so that you can load the existing investments from a file at the start and save all the investments (including the existing and any newly added investments) back to the given file at the end of your program. This implies that you need to specify a filename at the command line such as "java Portfolio <filename>" in order to run your program. For the purpose of input, we recommend the following format for the file:

```
type = "stock"
symbol = "AAPL"
name = "Apple Inc."
quantity = "500"
price = "142.23"
bookValue = "55049.99"

type = "mutualfund"
symbol = "SSETX"
name = "BNY Mellon Growth Fund Class I"
quantity = "450"
```

```
price = "42.21"
bookValue = "23967.00"
```

For the purpose of output, you will write the investments in the same format so that it can be used for input in the next run of your program. Note that the first time you run your program, the input file may not exist, and in such a case, you can skip input but create the file for output at the end of your program.

(4) In addition to the sequential search in Assignment One, you can use a HashMap as an index for the name keywords. To create such an index, you will tokenize the names of all investments and create a mapping entry for each keyword. For example, if the word "Bank" appears in the names of investments at the positions 0, 3, 7, and 10 of the ArrayList, the mapping entry will map "bank" to a list of [0, 3, 7, 10]. Similarly, the mapping entry for "toronto" may map to a list of [0, 5, 7, 12, 15]. To maximize the possible matches, all the keywords in the index and from the search request need to be normalized to lower cases so that "bank", "Bank", and "BANK" can all be matched.

To search for name keywords "Toronto Bank", you will first find "toronto" in the index to get a list of [0, 5, 7, 12, 15] and then find "bank" to get a list of [0, 3, 7, 10]. After that, you can reduce the search to the intersection of [0, 7], since only the investments at these positions in the ArrayList will contain both keywords "toronto" and "bank". If there are also requirements for a symbol and a price range, you will then search the reduced list of [0, 7] sequentially for all possible matched investments. Obviously, this can greatly speed up the search performance for a request that contains name keywords. If no name keywords are specified in a search request, you will still need to search the entire ArrayList sequentially to find all matched investments.

Note that this implementation assumes that all investments are stored in a single ArrayList so that their positions can be used to identify them. The name index needs to be created as soon as you load the existing investments from the input file and then updated each time you add a new investment to the list or delete an existing investment from it. When adding a new investment, all the keywords in its name should be checked: if one is already on the HashMap, just add the new position at the end of its position list; otherwise, create a new entry on the HashMap so that the new keyword is assocated with a list that contains the new position. For example, assuming that a new investment with the keywords "Toronto Star" is stored at the position 20 of the ArrayList, we will update the above position list for "toronto" to [0, 5, 7, 12, 15, 20], and if "star" is not on the HashMap, we will create a new entry for "star" and initialize its position list to [20]. When deleting an investment from the ArrayList, all the position lists on the HashMap need to be examined so that the positions with the values equal to the position of the deleted investment will be removed and those with the values greather will be decremented by one. If a position list becomes empty during this process, the corresponding entry will be removed from the HashMap as well. For example, if we delete the investment at position 12, the above lists for "toronto", "bank", and "star" will be changed to [0, 5, 7, 14, 19], [0, 3, 7, 10], and [19], respectively. Note that you do not need to save the index before terminating the program since it can be easily computed in the next run of your program.

(5) You should keep all of your classes in one package and use Javadoc to create a set of external documents so that the TA can easily examine the contents of your package for marking.

Deliverables: Similar to what is required for Assignment One, but for submission, you need to create a new folder named <userid>_a2 and include all the related files into this folder. For example, I would name the folder "fsong_a2" if I were to make a submission for Assignment Two.

Clarification for a Test Plan: For the documentation of an assignment, you are asked to provide a test plan, which consists of all major conditions you considered for testing, not how many test cases you used during testing.

For all the commands, you need to accept all reasonable variations for input but reject anything that is not reasonable. Take "quit" as an example, all of "q", "Q", "Quit", "quit", and "QUIT" are considered as reasonable, but things like "bye" and "exit" are not reasonable. For unreasonable input, you should provide a warning message and let the user try again.

For search, you need to identify all major conditions for the function. For example, to search an element on a list, we need to consider these four conditions:

1. the given element is not on the list;
2. the given element is at the start of the list;
3. the given element is at the end of the list;
4. the given element is somewhere between the two ends in the list.

For Assignment One, our search is a bit more complex in that there are different components in a search request, and the name keywords can contain more keywords which are case-insensitive for matching and can be in different order. In addition, words are matched as a whole, not partially (e.g., bank and banking are not matched). Nevertheless, you can still identify all the major conditions for a relatively complete testing on your code. Similarly, you should identify all major conditions for buying an investment to your program and other operations. In addition, when inputting a value, make sure that it's valid: e.g., a price should be positive; a quantity can't be zero, and so on.

Of course, for Assignment Two, you may need to adjust your test plan so that any new features can also be tested, but the above examples and explanations should help you understand what is expected for a test plan. Based on a test plan, we can easily come up with some test cases (i.e., real input/output pairs). If you like, you can add your test cases to the README file, but they are optional, since the TAs are likely to use their own test cases for marking.