CS/COE 0445 Spring 2015 Assignment 2

Online: Thursday, January 29, 2015
Due: All source (.java) and data files, plus a completed Assignment Information Sheet zipped into a single .zip file and submitted to the proper directory in the submission site by **11:59PM on Wednesday, February 11, 2015.**
(Note: see the submission information page for submission details)
Late Due Date: 11:59PM on Friday, February 13, 2015

Purpose: We have discussed Stacks both as a client (how to use them) and as an implementer. In this assignment you will get some practice using a Stack as a client in order to implement a simple simulation.

Idea:
You are the proprietor of a small ice-cream stand called "Lickety Splits". One of your specialties at the stand (as hinted at by the title) is your *famous-in-five-continents* banana split. People come from near and far just to taste those bananas, sauces, ice cream (3 kinds), whipped cream (fresh), nuts, and of course cherries (on top). Because of all the business you do, you must keep a lot of bananas on hand (crates full of them). However, since your store is so small, you must keep the bananas in a stack so you still have room to move around. In order to ensure a fresh product for your loyal customers, you have expiration dates on all your crates of bananas. The dates are in order from oldest (on top of the stack) to newest (on the bottom).

You have many suppliers of bananas, and each delivery made by a supplier consists of crates of bananas with a variety of expiration dates, depending on when they were packed. Unfortunately, the shipping companies are not very organized, and so the boxes in the trucks are in no particular order. Each time you receive a delivery, you must merge each new crate of bananas with those already in your inventory. This is necessary because dates on newly received crates of bananas may be anywhere in the range of dates of the bananas already on the stack. Naturally, you may have to remove some crates from your in-store stack – these will be stacked temporarily until you can put them back on the main stack. But remember, the crates are heavy, so you don't want to have to move them around too much. Furthermore, you pay a part-time employee by the crate to handle banana deliveries, so the more crates that have to be moved the more money you will pay.

When you whip up your world-famous banana splits, you always take bananas from the crate on your counter (not on the stack). Once the counter crate is empty you dispose of it and take the next crate from the top of your stack to use (unless you encounter the horrific special case of having no remaining crates of bananas!).

Remember, though, that the bananas do have expiration dates on them. Each day first thing you check your bananas versus the current date. Any crate (including the crate on the counter) with an expiration date less than the current date must be thrown away.

Periodically you would like to see how much you are spending on your bananas (and their movement), so you keep track of the costs of the crates and of the pay for your mover, and display this information when required.

Details:

Your assignment is to simulate this banana bonanza in the following way:
1) Maintain a global clock (using a counter), the value of which will be the current date. When your program begins, this clock should be set on 0 (this will be the beginning of the "Banana Epoch")
2) Represent a single crate of bananas as a Crate object (new class), which minimally must contain the following information: expiration date, initial banana count, current banana count and cost (for the entire crate). Your Crate class must also implement the Comparable interface, with expiration date as the comparison value, so that Crates can easily be compared to one another. Finally, since you are a good object-oriented programmer, you will keep your data private and provide necessary accessors and mutators for access to a Crate.
3) Maintain the stack of banana crates using a Stack data structure as discussed in class. Note that you will also

need a temporary Stack to use during some of your operations. **You must use the StackInterface from the text but may use either the ArrayStack or the LinkedStack for your implementation.**

4) Read in and process a file whose name is entered on the command line, which will contain a number of the following commands:

   a. **`receive`** – receive a new shipment of n crates of bananas. These will have to be merged into your Stack. After this command, the next line will contain an integer, **`n`**, indicating how many crates are to be received. The **`n`** lines after that will contain information about the individual crates of bananas. In particular, each of the **`n`** lines will be formatted as follows:

      `<date> <count> <cost>`

      where **`<date>`** is an integer representing the expiration date for the crate, **`<count>`** is an integer indicating the number of bananas in the crate, and **`<cost>`** is a double representing the cost of the crate. See the test files for some examples. Remember that these crates will not be ordered in the file, but must be kept ordered on your Stack. The smaller the date, the "older" the bananas are. The oldest bananas must be kept on the top of the Stack. **Note that you must simulate this merge as accurately as possible.** Crates must be removed from the store Stack in order to locate the proper position for any new crates, as dictated by the expiration dates. Crates that were removed must be stacked somewhere temporarily and then placed back onto the store stack. At the end of the entire process no crates should remain on the temporary stack (since it blocks the door to the restroom), but it does not have to be cleared between each individual crate merge. To save money you want to process the received crates in as few moves as possible. In particular, you have a part-time employee whose only job is to unload the crates during receipt of a shipment. She gets paid $1 per crate, so the more moving she has to do, the more money it will cost you. Thus, you want to minimize the total number of crates moved during this process. Each crate move, whether to move a crate from the truck to the store stack, to move a crate from the store stack to a temporary stack, or to move a crate back from the temporary stack to the store stack, counts as a move. However, be careful not to over count moves. For example, if a crate is taken from the top of the store stack (pop) and put onto a temporary stack (push) this should count as only a single move even though both a pop and a push are being done. During each **`receive`** you must also keep track of how many crates were received and how much they cost. **If you require too many moves during your receive command, you will lose grade points on the assignment.**

   b. **`use`** – use bananas for current orders. After this command, the next line will contain an integer, **`k`**, indicating how many bananas are needed. These are to be taken from the crate on the counter, unless that crate has fewer than **`k`** bananas. If the counter crate has only **`m (< k)`** bananas, then take the remaining **`m`** bananas from the counter crate, get a new crate from the Stack to put onto the counter, and then take the additional **`(k – m)`** bananas from that crate. Be careful to handle special cases (ex: need several crates to satisfy order; not enough bananas in the store to satisfy order). **Note:** Once a crate is being used (i.e. it is on the counter) it can never be put back onto the stack. Thus it is possible (due to new shipments) for a crate on the top of the stack to actually expire before the counter crate.

   c. **`display`** – display your counter crate and your stack of bananas. This should indicate clearly the order of the crates and all of the information for each crate (expiration, current count, initial count, cost). **Interesting / Important note:** The Stack ADT does not provide for a display operation, so to display the crates on your stack, you must pop all of them to a temporary stack and then push them back onto your store stack. Unlike receiving the crates, however, in real life we would assume that the crate information could be read on the sides of the crates without having to actually move them. Thus, the pops and pushes are simply an implementation issue and would not actually be done in the store. Alternatively, you may extend either ArrayStack or LinkedStack to a new class called DisplayableStack which adds one operation to the Stack operations: toString().

   d. **`skip`** – skip to the next day. The global clock must be incremented here. Any bananas whose expiration dates are now less than the clock value must be thrown out. A message should be displayed for each crate that must be thrown away.

   e. **`report`** – report your banana expenses in a nicely formatted way. Your report must have two sections:

      i. The expenses from your most recent shipment. This should include the number of crates received, their cost, and the cost of moving them around (including how many actual moves were required when the crates were put away).

ii. The overall expenses from the beginning of the simulation. The same data as above should be shown.

Note: It is possible that more than one shipment is received between calls of the `report` operation. Item i. above should only take into account the most recent shipment, but item ii. above should take into account all shipments.

5) You must implement your program in a good object-oriented way. Specifically, you must create a class Crate (in file `Crate.java`) to represent your banana crates and your Stack should be a Stack of Crate objects. Also, your Crate class should have the accessors and mutators necessary to implement the functionality required without giving public access to variables**. It is probably a good idea to encapsulate the functionality of your overall store into a class as well, so as to keep the complexity of your main program lower.** Your main program should be called Assig2 and should be in file `Assig2.java`.
6) Print all of your output to standard output. Note that there is no interactive input to this program. All input is read in from the file, whose name is entered at the command prompt. For an idea of how your output might look, and to see some reasonable values for the moves, see some sample input and output files:

`test2-1.txt`	`test2-1-out.txt`
`test2-2.txt`	`test2-2-out.txt`
`test2-3.txt`	`test2-3-out.txt`

Make sure you **submit ALL of the following in a single .zip file** to get full credit:

1) All of the files that you wrote, well-formatted and reasonably commented:
   a) Crate.java
   b) Assig2.java
   c) Any other files that you wrote (ex: DisplayableStack.java)
   d) All data files
2) All files necessary for your Stack:
   a) StackInterface.java
   b) Either ArrayStack.java or LinkedStack.java (unless you wrote DisplayableStack.java)
3) Your completed Assignment Information Sheet. Be sure to help the TA with grading by clearly indicating here what you did and did not get working for the project.

As with Assignment 1, the idea from your submission is that your TA can compile and run your programs WITHOUT ANY additional files, so be sure to test them thoroughly before submitting them. **If you use an IDE for development, make sure your program runs from the command line without the IDE before submitting it.** If you cannot get the program working as specified, clearly indicate any changes you made and clearly indicate why, so that the TA can best give you partial credit.

Extra Credit Ideas: If you want to get some extra credit, here is an idea:

– Allow one or more additional non-trivial commands. For example, you can make your simulation more useful by also including purchases and the money earned from them. Thus, when showing your report it will include both expenses and revenues and you can determine if a profit has been made. If you do this option, you should write a second program and your own input files so that the TA can test both your original required submission and your extra credit.