Project 3: Stacks and Queues
Due: Tues, Oct 28, at 11:50 am

Project partners. For this project you will work in pairs. The teams are:

Jessica & Ruichen, Katie & Lauren, Asaf & Albert, Noah & Yifan, Juan & Grace, Elissa & Moustafa, Pong & Will, Izamar & Steven, Greg & Roberto, Carson & Murat, Nicolas & Jamie

Working in pairs. Here are some guidelines:

1. You should always try to code together, i.e., both people at the same workstation at the same time.

2. Alternate who does the typing so that each person gets to code for half the time. Do not let the same person be the typist for all the code.

3. Set up a schedule of meeting times right away. Two or three meetings a week of 3 hours each is probably a good start. (I would schedule meetings with each other that total at least 6 hours a week, and keep another 3 on reserve.)

Comments and style. Javadoc-style comments are required throughout your code for this project and all future projects. As an example, see the commenting style in the code that has been provided to you in previous projects. For a quick javadoc style guide see:


10% of your grade will be allotted to coding style, documentation/comments, and correctly following submission instructions.

Submission. Your completed project should be submitted twice: First, via moodle, then also in hard copy at the start of class on the day it is due. All files that you submit via moodle should be collected into a folder called <username>_proj3. For example, if you are Joe Smith your folder should be called jsmith_proj3. You should include all .java files, as well as their corresponding compiled .class files. Be sure that the .class files you submit are exactly what I would get if I compiled the .java files you submit. You should also include any other supporting files (.txt files, etc, if you have any). The folder should then be compressed into a .zip file (or some other standard type of compression file). The compressed file should then be uploaded using the Project 3 link on the COM212 moodle page. See course website for late project submission policies.
Part A (20%). Matching Brackets. Complete a program that performs the bracket matching task we discussed in class. Your program will prompt the user for an input string, and then output whether the string has properly-matching brackets.

Roughly speaking, brackets are properly matched in a string of text if:

1. every opening bracket has its own corresponding closing bracket,
2. every closing bracket corresponds to an opening bracket, and
3. no opening bracket reaches its corresponding closing bracket until all the opening brackets coming after it have already been closed.

The formal definition of whether a string’s brackets are matched can be derived from the pseudocode for the algorithm to solve the problem, which we studied in class and can also be found on page 235-236 of the GTS text.

For examples on correctly matched and incorrectly matched strings of brackets, see page 235 of the GTS text. (These might be good test inputs for your program once you think it’s working.)

As we discussed in class, a simple way to solve this problem is to use a stack. Therefore, you will need to implement a Stack class, with its standard fields and methods as discussed in class. You may choose whether to implement an array-based stack or a linked-list-based stack. You can test that your stack operations are working properly from a main method (used just for testing) within your stack class.

To solve this Bracket Matching problem, you will also implement the following classes:

- The BracketMatchApp class. (Only contains a main method, no fields or other methods.)
  - the main method: /* drives the user input/output of the program */

    prompt user for input string
    output whether or not the string has its brackets matched properly

  - The simplest way to do user input from the console is probably to use java’s built-in Scanner class. Here is an example from your text of how to use it. Test it to see how it works. Here are some other helpful links on how to use scanner objects: http://www.cs.williams.edu/~jeannie/cs136/scanner.pdf (note on this document: apparently the nextString() method doesn’t work, so you have to use next()) and http://www.java-made-easy.com/java-scanner.html.

- The BracketMatcher class.
  - Fields:
    - hard-coded array of characters (of type char) holding the opening brackets (these should include: ‘(', '[', '{', '<')
    - hard-coded parallel array of characters (of type char) holding the closing brackets. (You will need to read about the Java type char, say, here.)
Methods:
- `boolean isOpeningBracket(char c)`
  - returns true iff the character `c` is an opening bracket
- `boolean isClosingBracket(char c)`
  - returns true iff the character `c` is a closing bracket
- `boolean corresponds(char open, char close)`
  - returns true iff the character `open` is a bracket that corresponds to the closing bracket character `close`
  - note that we set up parallel arrays of brackets to help us with this task
- `boolean checkBrackets(String s)`
  - returns true iff the string `s` has brackets that are all matched up properly
  - calls the other three methods to achieve this
  - You’ll probably also need to familiarize yourself a little more with Java’s String class so you know how to do a little string manipulation (like indexing into a String to access the characters that comprise it). [Here](#) is the official Javadoc documentation for the String class. Seems to me like the `charAt(i)` method should be helpful.
Part B (20%). Playing the Market. From P-6.36 from the GTS text: “When a share of common stock of some company is sold, the capital gain (or, sometimes, loss) is the difference between the share’s selling price and the price originally paid to buy it. This rule is easy to understand for a single share, but if we sell multiple shares of stock bought over a long period of time, then we must identify the shares actually being sold. A standard accounting principle for indentifying which shares of a stock were sold in such a case is to use a FIFO protocol—the shares sold are the ones that have been held the longest (indeed, this is the default method built into several personal finance software packages). For example, suppose we buy:

- 100 shares at $20 each on day 1
- 20 shares at $24 each on day 2
- 200 shares at $36 each on day 3

then sell:

- 150 shares at $30 each on day 4.

Applying the FIFO protocol means that of the 150 shares sold, 100 were bought on day 1, 20 were bought on day 2, and 30 were bought on day 3. The capital gain in this case would therefore be $100*$10 + 20*$6 + 30*(-$6) = $940.”

Write a program with the following four main menu options:

1. Buy
2. Sell
3. Total Capital Gain So Far
4. Quit

If option 1 is chosen, the user should be prompted to enter how many shares they wish to buy and at what price. These shares should be added to the queue of shares currently held, and the program should output a confirmation of the purchase. It should then again prompt the user with the main menu.

If option 2 is chosen, the user should be prompted to enter how many shares they wish to sell and at what price. These shares should be removed from the queue of shares currently held, the capital gain/loss should be updated, and the program should output a confirmation of the sale. It should then prompt the user with the main menu again.

If option 3 is chosen, the total capital gain (or loss) from all transactions so far should be displayed. Then the user should be prompted with the main menu again.

If option 4 is chosen, the program should end.

To achieve this, your code should be organized into three classes.

1. The first is a standard Queue class (as discussed in class and in readings). You may choose whether you wish to implement the queue as an array or a singly linked list. You can test that
your Queue operations are working properly from a main test method within the Queue class.

2. The second is a class that will handle and record all the transactions...

The CapGain class.

- **Fields:**
  - double totCapGain;
    - the current total capital gain or loss in dollars
  - Queue sharesHeld;
    - a queue of the purchase prices of all shares currently held

- **Methods:**
  - void mainMenuPrompt()
    - print out the main menu
  - void buy(int numShares, double price)
    - add numShares shares at price price to the queue
  - void sell(int numShares, double price)
    - remove numShares shares from the queue, calculating gain/loss for each share (based on the price it was purchased for and the price it is being sold for), and updating the totCapGain.
  - double getTotalCapGain()
    - return totCapGain

3. And finally, a class for the main method that will drive the whole program (input/output)...

The CapGainApp class.

- **a main method that will do something along the lines of:**
  - create new CapGain object called cg
  - create new Scanner object for I/O /*see Part A for more on Scanners*/
  - do {
      /*yes this “do-while” loop is an actual Java loop construct you can use*/
      cg.mainMenuPrompt();
      get answer from user
      if answer is 1 {
        get numShares from user
        get price from user
        call cg.buy(numShares, price)
      } else if answer is 2 {
        ...
      } else if answer is 3 {
        nicely output cg.getTotalCapGain()
      } else {
        output farewell message
      }
    } while (user choice is between 1 and 3)

- **note:** it is better to use iteration (as outlined above), not recursion, to keep bringing up the main menu until the user quits
**Part C (20%). No PEMDAS Please.** You have been using one of very classic calculator, the HP-35, for a long time. It was the first handheld calculator manufactured by Hewlett-Packard in 1972. However, after a disastrous accident (dropped it in a sink), it is no longer functional. You miss this calculator so much you finally decided to implement its special form of postfix calculation yourself.

From C-6.19 of the GTS text: “Postfix notation is an unambiguous way of writing an arithmetic expression without parentheses.” Sounds great, doesn’t it?

Here is an explanation of postfix notation. You should solve the practice problems on slides 7 and 8 to check that you understand what’s going on. More on postfix notation here: http://www.minich.com/education/wyo/acsl/prefix/index.htm

Your program will input an arithmetic expression in postfix notation and then evaluate it, outputting the answer. (Thankfully, this problem is easier to devise an algorithm for than, say, evaluating an expression in the regular old infix notation we always use.)

To simplify things, we will assume that the input expression only consists of single-digit values. We will also assume only the four basic arithmetic operators are used: +, -, *, and /. When the user inputs a forward slash (“/”) it should be interpreted as regular division (not integer division).

Before you begin, you and your partner must come up with an object-oriented design to solve this problem. You can model your design after the way the solutions to Parts A and B were laid out for you. As in those parts, your solution here should have at least 3 classes. Yes, you can re-use the Queue or Stack classes you created for Parts A and B. (You will probably not need both of them.) You will need to submit a document that describes/diagrams the classes in your design and their purposes, their fields and methods, and how they each depend on or use one another. This design document will count for 5% of your grade for Part C.
Part D (25%). Photoshop Edit History. Everybody loves digital photos nowadays. You often use photo-editing software such as Adobe Photoshop to create interesting images. Unfortunately your computer got a notorious computer virus, the *Undo History Killer*. It corrupted all the undo history tracking modules from all applications on your machine. However, you really want to use Photoshop again to touch up your birthday party pictures with the undo/redo feature, when you suddenly realize you have the ability to implement it yourself!

The Photoshop app provides history state of up to 20 entries because each editing operation may require large memory space to be stored. For this assignment you should maintain a maximum of 10 entries in the edit history manager.

We will simulate this editing operation with simple text strings such as “delete”, “blur”, “crop”, and so on. These text commands are user-entered inputs from the terminal (or command line tool). There are two special commands, “undo” and “quit”. When the edit history manager receives an “undo” command, it shows the last command in the edit history. See the below execution example.

```
Type edit command> delete
Type edit command> crop
Type edit command> blur
Type edit command> undo
result > undone blur command
Type edit command> delete
Type edit command> undo
result > undone delete command
Type edit command> undo
result > undone crop command
Type edit command> undo
result > undone delete command
Type edit command> undo
result > no more edit entry to undo
Type edit command> quit
Good-bye!
```
It is safe to assume that all input strings is one of editing commands except “undo” and “quit”. In other words, you only need to detect if the input command is either “undo” or “quit”. Otherwise, you just add it to the edit history.

Here are some implementation guidelines.

The EditManager class: this class maintains edit history entries and performs undo commands. You may re-use/revise any data structures you’ve created from other parts of this project or previous projects to implement underlying data structure for this edit history manager. Make sure that the manager can store an edit history of only up to 10 entries. ADTs you might consider for accomplishing this include the Deque ADT or a LeakyStack (see page 256 of GTS).

The EditTester class: Similar to the calculator project, you will need to implement a test driver class to verify your implementation. The main method will create EditManager object and other necessary local variables to take user input and process it via EditManager object.
Part E (5%): Choose between the following two options. If you complete both, make a note of it in the comments of your program for some bonus credit.

1. **Enhanced Bracket Matcher.** Have your program from Part A not only output whether or not the input string has matching brackets, but if the brackets don’t match, have it report the character number where it discovered the error (starting at 0 from the left) and the incorrect character it found there. For example, for the input string:

   a{b{c]d}e

   the output from the program should be:

   Error: ‘]’ at position 5

2. **Better Postfix Evaluator.** Fix your program from Part C so that it works on inputs with values that are more than one digit long (you may assume the user will enter spaces between values).

**BONUS 1 (+10%):** Write a program for parsing and evaluating arithmetic expressions. For an idea of the algorithm, see [http://www2.lawrence.edu/fast/GREGGJ/CMSC150/Infix/Expressions.html](http://www2.lawrence.edu/fast/GREGGJ/CMSC150/Infix/Expressions.html) or [http://www.smccd.net/accounts/hasson/C++2Notes/ArithmeticParsing.html](http://www.smccd.net/accounts/hasson/C++2Notes/ArithmeticParsing.html). Start with assuming the expression has no parentheses. After you get that working, for an extra 5% you can add code to handle parens.

**BONUS 2 (+10%):** How about “redo”? If you look at edit functions in most programs such as MS Word, Text Editor, etc, you see a “redo” command too. The behavior of redo is slightly different from undo. It will only keep entries that have been undone by the “undo” command, and it also clears the redo history when a new edit command comes in. (Note that this is analogous to the behavior of the “forward” and “back” buttons of a web browser.) Play with this “undo” and “redo” behavior with any program (or web browser) and implement the “redo” command too.