ETEC2101.01	**Lab01: Digging Into Java**	Points: **100**
Assigned: **8/22/2016**	Due: **9/7/2016 at 11:59pm SHARP (no more extensions☺)**	

Objectives:

- Get you used to the your chosen IDE and the Java language (esp. the OOP aspect)
- Get used to my lab style. Note: early labs will be very detailed / rigid; as we progress, descriptions will be a little more vague / open-ended.

Tasks:

1. Read through the entire lab
    a. Make sure you understand what you're being asked to do before coding starts.
    b. Ask lots of questions!
    c. Start now – if you want until Thursday++ you'll be super-stressed.
    d. Don't forget the "Additional Notes" section – this will often include material not discussed in lecture (but it's still fair game for the quiz).

2. Create a new Java application project (name it whatever you like)

3. Download the starting database file from the web page (**dbase.txt**). Put it in your top-level project folder.
    a. Note: If you open it in Windows notepad, it'll look like one long line. Try using Notepad++ or your IDE.

4. (**5 points**) All of your Java classes should be in the lab01_xx (where xx is your initials) package except the main program which should be in the global namespace.

5. (**15 points**) Transaction class (in a file called **Transaction.java**)
    a. Instance variables (all of these should be protected)
        i. **mID** (int): the ID# of this transaction
        ii. **mAccount** (int): The account the transaction is applied towards
        iii. **mAmount** (float): The amount of the transaction (positive = deposit, negative = withdrawal)
        iv. **mNotes** (String): Any notes associated with this transaction
    b. Instance methods
        i. There should be two constructors[1]. Both are passed the account, amount, and notes. The first constructor should take an ID#. The second should auto-create the ID# (to be one higher than any other transaction created to date)[2].
        ii. compare
            1. Should take another Transaction reference and an integer indicating which field to compare upon (1 = ID, 2 = Account#, 3 = Amount, 4 = Note-length).
            2. Should return a boolean indicating if this element is less than the one passed (using the given field) [you can return false if the int doesn't have a valid value[3]]
    c. If a Transaction is converted to a string (e.g. if in a System.out.println call), it should be of this format (see the example program for details):
        `[xxxx:yyyy:$zzz:www]`
        (where xxx is the ID#, yyyy is the account#, zzz is the amount, and www is the notes)
    d. [This would be a great place to test your code…]

[1] This is called method **overloading**. Java will pick the appropriate constructor based on the passed arguments.
[2] Hint: You might need a class variable here…
[3] Or better yet, **throw** a new Exception…(+5 points)

6. (**65 points**) Database class (in a file called **Database.java**)
    a. Instance variables (all of these should be protected)
        i. **mTransactions** (Transaction[]): The array of transactions. I want this to be a "normal" array (not ArrayList, LinkedList, etc.)
        ii. **mSize** (int): The number of elements of mTransactions we're using (this will be less than or equal to mTransactions.length).
        iii. **mFileName**: The file-name we load the database from initially (if available) and save to when done with the program. This value should be passed to the constructor.
    b. Class variables
        i. **sSizeIncrement** (int): If we attempt to add a new Transaction and mTransactions is full (mSize is equal to mTransactions.length), then mTransactions should be re-allocated to be this much larger. You can initialize this to a reasonable value (e.g. 20).
    c. Instance methods:
        i. Constructor: Should take the file name as an argument. If this file exists (i.e. no Exceptions are thrown), read all data from the file, populating the mTransactions array. You'll find it handy to call the addTransaction method (used in other places).
        ii. **addTransaction**: Should take a Transaction and add it to mTransactions. The mTransactions array should be "expanded" (i.e. create a new larger array, copy everything from the old, and replace the old) if necessary.
        iii. **sortTransaction**: Should take an integer as a parameter indicating which field to sort on. The method should apply the bubble-sort algorithm to mTransactions based on that field. Hint: use the compare method we wrote in the Transaction class.
        iv. **removeTransaction**: Should take an id# as a parameter and remove that element. Note: In an array, you must find the given id first (if it doesn't exist, do nothing, except perhaps display a message to the screen). Then you must "collapse" the array (copy everything from position i + 1 to mSize-1 to positions i to mSize – 2) and then decrement mSize.
        v. **saveDBase**: Should write all data to a text file (identical in format to the one you downloaded). Note: any new records created should appear in this new file.
        vi. *You'll probably also need some "getters" to return (in a controlled fashion) data from your class. Ask about these if you need a hint.*

7. (**5 points**) Main program (in a file called **MainClass.java**)
    a. At a minimum, create a Database instance and test all methods.
    b. If attempting the bonus, just make sure the menu code runs automatically.

8. (**10 points**) Style
    a. Use good internal variable names (e.g. "menu_choice" is better than "x").
    b. Add comments. This should include a sentence or two above / beside each class, method, and attribute and internal comments on any long / tricky sections.

9. (**Up to 20 bonus points**) Menu
    a. Create an interactive menu as seen in the example program.
    b. When appropriate "commands" are entered, methods in the database instance should be called.
10. Submit your work to our blackboard page as a **zip** file with source included. If you don't properly submit your file (**-10 points**).

Sample Run

This is one sample run of my solution. Text that is highlighted indicates the user entered that value.

```
***************
* Main Menu    *
***************
1. Display Records
2. Sort Records
3. Add Record
4. Remove Record
5. Show account balance
9. Quit
>>> 1
How many records to display (Enter for all):
0 [0:1452:$13.21:initial deposit]
1 [3:7963:$287.19:Mr Jones's birthday money]
2 [7:1452:$-4.17:]
3 [4:2296:$99.99:Almost a $100]
4 [8:7963:$-114.33:]
5 [2:5564:$1.0E8:Is this money counterfeit?]
6 [1:1452:$-5.15:]
7 [5:2296:$0.01:Now it's $100]
8 [6:7963:$-180.0:Uh-oh...]

***************
* Main Menu    *
***************
1. Display Records
2. Sort Records
3. Add Record
4. Remove Record
5. Show account balance
9. Quit
>>> 2
Sort by:
        1: ID
        2: Account#
        3: Amount
        4: Note-length
>>> 1
***************
* Main Menu    *
***************
1. Display Records
2. Sort Records
3. Add Record
4. Remove Record
5. Show account balance
9. Quit
>>> 1
How many records to display (Enter for all):
0 [0:1452:$13.21:initial deposit]
1 [1:1452:$-5.15:]
2 [2:5564:$1.0E8:Is this money counterfeit?]
3 [3:7963:$287.19:Mr Jones's birthday money]
4 [4:2296:$99.99:Almost a $100]
5 [5:2296:$0.01:Now it's $100]
6 [6:7963:$-180.0:Uh-oh...]
7 [7:1452:$-4.17:]
8 [8:7963:$-114.33:]

***************
* Main Menu    *
***************
1. Display Records
2. Sort Records
3. Add Record
4. Remove Record
5. Show account balance
9. Quit
>>> 1
How many records to display (Enter for all): 3
0 [0:1452:$13.21:initial deposit]
1 [1:1452:$-5.15:]
2 [2:5564:$1.0E8:Is this money counterfeit?]

***************
* Main Menu    *
```

```
***************
1. Display Records
2. Sort Records
3. Add Record
4. Remove Record
5. Show account balance
9. Quit
>>> 3
Account#: 1452
Amount: $15.39
Enter Notes (Hit Enter for no notes): blah
***************
* Main Menu    *
***************
1. Display Records
2. Sort Records
3. Add Record
4. Remove Record
5. Show account balance
9. Quit
>>> 1
How many records to display (Enter for all):
0 [0:1452:$13.21:initial deposit]
1 [1:1452:$-5.15:]
2 [2:5564:$1.0E8:Is this money counterfeit?]
3 [3:7963:$287.19:Mr Jones's birthday money]
4 [4:2296:$99.99:Almost a $100]
5 [5:2296:$0.01:Now it's $100]
6 [6:7963:$-180.0:Uh-oh...]
7 [7:1452:$-4.17:]
8 [8:7963:$-114.33:]
9 [9:1452:$15.39:blah]

***************
* Main Menu    *
***************
1. Display Records
2. Sort Records
3. Add Record
4. Remove Record
5. Show account balance
9. Quit
>>> 5
Enter Account#: 1452
Account 1452 has a balance of $19.279999
***************
* Main Menu    *
***************
1. Display Records
2. Sort Records
3. Add Record
4. Remove Record
5. Show account balance
9. Quit
>>> 9
GOODBYE!!!
```

Additional Notes: Bubble-Sort

There are *many* sorting algorithms out there[4]. Each has strengths and weaknesses. Bubble-Sort is generally considered the least efficient, but (one of the) easiest to implement sorting algorithms. It's often used as a baseline for comparison with other algorithms. My variant of it looks like this (in pseudo-code)

Input: **L** (a list of **n** values, generally, but not necessarily, unsorted)
Output: **L** (sorted in ascending order "in-place")
Algorithm in *pseudo-code*[5]:

```
sorted = False
while not sorted:
      sorted = True
      for i in range(0, n-1):   # i is 0, 1, 2,…, n-2
            if L[i] > L[i + 1]:
                  # It's out-of-order.
                  swap elements L[i] and L[i + 1]
                  sorted = false
      end for
end while
return L
```

Example:

Suppose L is [5, 1, 3, 0, 2] (n is 5)
After the first iteration of the while loop, L will look like [1, 3, 0, 2, 5] (we did 4 swaps)
After the second iteration of the while loop, L will look like [1, 0, 2, 3, 5] (we did 2 swaps)
After the third iteration of the while loop, L will look like [0, 1, 2, 3, 5] (we did 1 swap)
After the fourth iteration of the while loop, L will (still) look like [0, 1, 2, 3, 5]. sorted remains true, so we exit the while loop.

[4] You'll see several in ETEC3401 (Algorithms) next Fall.
[5] Pseudo-code is simplified code (without many of the details necessary to implement it in Java).