Announcements

- Midterm 1 and Program 3 have been graded
  - Class average was 77%
  - There will be no curve to the midterm.
  - Don’t freak out, I will likely incorporate extra credit as part of future assignments
- Still working on Program 4
- Midterm 2 is November 6th

Midterm 1

Questions 41 - 45

- Line 34: foo
  - x = 5
  - y = 6
- Line 35 bar
  - x = 5
  - y = 6
- 41) Line 36 print bar.getX()
  - Prints 5
- 42) Line 37 print foo.getX()
  - Error - this is because foo's getX() method attempts to access a private variable that it inherited from Bar
- 43) Line 38 print Foo.subFive(foo)
  - Prints 1
  - The input parameter is foo, whom's "y" value is 6, and 6 minus 5 is 1
  - Since subFive is a static method this type of method invocation is okay

Midterm 1 (cont.)

Questions 41- 45

- 44) Line 39 print foo.y
  - Prints 1
  - This is because the foo object that was an input to the “subFive” method call on the preceding line was altered, and that change inside the “subFive” method affected the foo object outside of the method , due to foo being a reference type
- 45) Line 40 print bar.getX()
  - Prints 5
  - This is because the bar object has remained unchanged since the last “getX” method call

Midterm 1 (cont.)

Question 46

Let’s work through this one on the board...

Data Structures

- A **data structure** is a way of organizing, and performing operations on data.
- Operations performed on a data structure include accessing or updating stored data, searching for specific data, inserting new data, and removing data.
- A data structure is implemented using classes, methods, and interfaces
- A commonly used data structure is: an **ArrayList**
- We will now look at other common data structures, as well as the role that interfaces and abstract classes play in conjunction with data structures

Reference Type

- A **reference** is a variable type that refers to an object, it can be thought of as storing the memory address of an object

Linked Lists

There are two types of linked lists: **singly** and **doubly**, we will begin by looking at the singly-linked list.

- A **singly-linked list** a data structure made up of a series (or list) of consecutive nodes.
- Each **node** typically has data and a pointer (a reference) to the next node.
- The first node in a singly-linked list is called the **head**
- The last node in a singly-linked list is called the **tail**

Let’s look at an example...

Linked Lists (cont.)

- The **append** operation for a singly-linked list, inserts a new node after the list's tail node (java tends to call this operation: **addLast**)
- Appending a node addresses a common algorithmic question: What is the behavior when appending to a list that is empty, versus when the list is not empty?

Linked List’s

- Append to an empty list: If the list’s head pointer is null (i.e. it is empty), the list’s head and tail now point to (i.e. reference) the new node.
- Append to a non-empty list: If the list’s head pointer is not null (i.e. there is at least one element in the list), the tail node’s “next” pointer must reference the new node, THEN the reference to the tail node must change to be the newNode

Let’s look at an example...

Linked Lists (cont.)

- The **prepend** operation for a singly-linked list inserts the new node before the list's head node (java tends to call this operation: addFirst)
- Similar to the append operation, there are two cases to be concerned with: behavior when the list is empty versus behavior when the list is non-empty

Linked Lists (cont.)

- Prepend to an empty list: if the list’s head pointer is null (empty), the list’s head and tail pointers reference the new node.
- Prepend to a non-empty list: if the list’s head pointer is not null (i.e there is at least one element in the list), then the new nodes “next” pointer must reference the head node, and THEN the head node must reference the new node.

Let’s look at an example...

Linked Lists (cont.)

- The **insertAt** operation for a singly-linked list inserts the new node at a provided index. (java tends to call this operation add(int index, E element))
- Note: this is different from zybooks which talks about "InsertAfter" which inserts after a specified node, the two operation are very similar.
- There are now three scenarios we have to worry about when doing this operation: empty list, insert at lists tail node, insert in the middle of the list

Linked Lists (cont.)

- Insert as the list’s first node: if the list’s head pointer is null, the list’s head and tail should then point to (i.e reference) the new node
- Insert at the tail node: If the list’s head pointer is not null (i.e the list is not empty) AND the index is the same as the Tail pointers current index, then we must set the Tail pointers previous nodes next to reference the new node, and the new node’s next pointer to reference the tail
- Insert in the middle of the list: if the list’s head pointer is not null, set the new node’s next pointer to reference the previous indexes nodes next reference, THEN set the next pointer at the previous indexes node to reference the new node

Note: These operations assume we have access to the previous indexes node