CSC 171 LAB 08

Lab Instructions

The goal of this lab is to give you an opportunity for more practice with abstract classes and interfaces, and a first chance to experiment with polymorphism.

For this lab, you will need to reuse and modify the code that you wrote in Lab 07. For Eclipse users, start by creating a new project for Lab 08, and copying the code from the src folder in Lab 07 to the src folder in Lab 08. You can then right-click and choose "Refactor -> Rename" from the pop-up menu to rename the package to "**`csc171.lab08`**". You do not need to copy any tests that you wrote.

The code for "`ListOfItems`" is included at the end of this document for those of you that may not have completed it.

You will be testing much of your code in sections 9 and 10, but you are still required to write smaller tests where appropriate for each section. Tests should be implemented in the package "**`csc171.lab08.tests`**".

1. An interface can inherit the methods defined in another interface using the **`extends`** keyword similar to the way that classes do. Create a new interface named **`EnhancedItems`** that **`extends Items`** and adds the following methods:
    a. **`public Object remove( int index )`** - removes the item at the specified index from the collection and returns the item.
    b. **`public void insert( Object item, int index )`** - inserts the specified item into the collection at the specified index.

2. Modify the **`AbstractItems`** class so that it implements **`EnhancedItems`** rather than **`Items`**. This will cause compilation errors in your **`ArrayOfItems`** and **`ListOfItems`** classes because these classes do not implement the new methods in the **`EnhancedItems`** interface. Repair these compilation errors by adding stubbed implementations of the missing methods. Recall that a stubbed method is a method that does nothing more than returning a default value like **`0`** or **`null`**.

3. Implement the **`remove`** method in your **`ArrayOfItems`** class. A removed item should no longer be displayed in the output of the **`toString`** method, nor should it be returned as a result of calling the **`get`** method. Your **`size`** method should also return an appropriate value (e.g. 3 before the remove, and 2 after). A proper implementation will require you to *shift* the items in your array to fill in the gap created by the removed item. For example, assume that your array had the following contents before remove is called (null indicates positions in the array that haven't been filled yet):

["first"]["second"]["third"]["fourth"][null]

If **remove(1)** is called, the value **"second"** should be removed from the array and the items **"third"** and **"fourth"** *shifted* one position to the left..

["first"]["third"]["fourth"][null][null]

4. Implement the insert method on your **ArrayOfItems** class. An inserted item should be placed into the correct index in the array, even if it is between items already in the array. Your **toString** and **get** methods should reflect the new sequence of items. A proper implementation will require you to shift the items in your array to make room for the new item. For example, assume that your array had the following contents before insert is called:

   ["a"]["b"]["d"][null][null]

   After **insert("c",2)** is called to insert the value **"c"** at index **2**, **"d"** is shifted one position to the right to make room**:**

   ["a"]["b"]["c"]["d"][null]

5. Modify your **ArrayOfItems** class so that it can hold more items than its initial capacity would allow. For example, if you create an **ArrayOfItems** large enough to hold 5 elements, you should be able to add 6 or more elements to the collection. This will require you to modify your add method to determine when the array is full. If the array is full, a larger array will need to be created (e.g. 2X larger than the current array) and the items from the old array copied to the new array. This should allow you to add as many items as you want to your collection without causing errors.

6. Implement the **remove** method in your **ListOfItems** class. As with **ArrayOfItems**, a removed item should no longer appear when the **toString** or **get** methods are called. A proper implementation should remove the Node containing the item by changing the "next" reference in the Node *before* the remove index so that it references the Node *after* the remove index. Your method need not handle an index of 0.

7. Implement the **insert** method in your **ListOfItems** class. As with **ArrayOfItems**, the new item should be inserted into the proper position in the list, even if it is between two other items already in the list. A proper implementation will update the "next" reference in the Node *before* the insertion index to refer to the newly inserted Node, and the newly inserted Node will will refer to the Node *after* the insertion index. Your method need not handle an index of 0.

8. At this point you should have two fully functional, though relatively simple, implementations of the List data structure. In your README, discuss briefly why it was important to define the **add**, **get**, **insert**, and **delete** methods in the **Items** and **EnhancedItems** interfaces, but not to try to provide implementations in the

`AbstractItems` class. Also discuss why it *did* make sense to implement the **`addAll`** and **`toString`** methods in the **`AbstractItems`** class.

9. In Labs 07 and 08 you have created two interfaces (**`Items`** and **`EnhancedItems`**), an abstract class (**`AbstractItems`**), and two concrete implementations (**`ArrayOfItems`** and **`ListOfItems`**). Create a class called **`Lab08Test`** that has a *single* (i.e. not overloaded) method called "**`testItems.`**" This method should take a single parameter, The type of the parameter should be one the interfaces or classes that you created. The method should do the following:
    a. Add at least 5 unique items to the collection one at a time.
    b. Add an array of items to the collection.
    c. Insert a unique item somewhere in the middle of the collection.
    d. Insert a unique item after the last index (size).
    e. Removes an item from somewhere in the middle of the collection.
    f. Removes the item at the last index (size - 1).

    Make sure to print the collection using the **`toString`** method after each step above.

    Note: If you had trouble implementing any of the methods required for one of the steps above on either (or both) of your collections, please write a comment above the specific step, and mention it in your README as well. The TAs should not dock points here for problems with a previous section. For example:

```
// the following does not work with my ListOfItems
items.insert( "B", 4 );
```

10. Write a **`main`** method in your **`Lab08Test`** class that creates an instance of the class and calls the **`testItems`** method twice: once with an **`ArrayOfItems`** with an initial capacity of 5, and once with a **`ListOfItems`**. In your README, briefly describe how you chose which class or interface to use for the parameter type in the **`testItems`** method, and how polymorphism enabled you to use the same method to test both of your collections. Your output should look something like this:

```
Calling testItems with ArrayOfItems
a b c d e
a b c d e f g h
a b c d A e f g h
a b c d A e f g h C
a b c d A e f h C
a b c d A e f h
Calling testItems with ListOfItems
a b c d e
a b c d e f g h
a b c d A e f g h
a b c d A e f g h C
```

```
a b c d A e f h C
a b c d A e f h
```

HAND IN

Before handing in, create two additional files in your lab directory:

1. Create a README that contains the following:
    a. Your contact information (name, class, lab session), TA name, and assignment number.
    b. A brief (one paragraph at most) description of the assignment.
    c. Instructions explaining how to run your code. While it is possible that some TAs will compile, run, and test your code in Eclipse, it is also possible that they will want to compile and run it from the command line. You should include instructions on how to compile and run each of the executable Java classes.
2. Create a file titled "SampleOutput" showing the results of running your code. (You may copy and paste from the Eclipse console). If appropriate, please include a comment above the output for each section (e.g. "#OUTPUT FOR SECTION 1") and put a few blank lines between each section.

Hand in by uploading the compressed (i.e. "zipped") folder containing your assignment files to Blackboard.

```
package csc171.lab08;
public class Node {
	private Node node;
	private Object item;

	public Node() {}

	public void setNext( Node node ) {
		this.node = node;
	}
	public Node getNext() {
		return node;
	}
	public void setItem( Object item ) {
		this.item = item;
	}
	public Object getItem() {
		return item;
	}
}
```

```
package csc171.lab07;

public interface Items {
	public void add( Object item );
	public void addAll( Object[] items );
	public Object get( int index );
	public int size();
	public String toString();
}

package csc171.lab07;

public abstract class AbstractItems implements Items {
	@Override
	public void addAll( Object[] items ) {
		for( Object item : items ) {
			add( item );
		}
	}

	@Override
	public String toString() {
		String output = "";

		int size = size();
		for( int i=0; i<size; i++ ) {
			output = output + get( i ) + " ";
		}

		return output;
	}
}
```

```
package csc171.lab07;

public class ArrayOfItems extends AbstractItems {
	private Object[] items;
	private int size;

	public ArrayOfItems( int maximumSize ) {
		items = new Object[maximumSize];
		size = 0;
	}

	@Override
	public void add(Object item) {
		items[size] = item;
		size = size + 1;
	}

	@Override
	public Object get(int index) {
		return items[index];
	}

	@Override
	public int size() {
		return size;
	}
}
```

```
package csc171.lab08;

public class ListOfItems extends AbstractItems {
	private Node head;
	private Node tail;
	private int size;

	public ListOfItems() {
		head = new Node();
		tail = head;
		size = 0;
	}

	@Override
	public void add(Object item) {
		if( size() == 0 ) {
			head.setItem( item );
		}
		else {
			Node node = new Node();
			node.setItem( item );
			tail.setNext( node );
			tail = node;
		}
		size = size + 1;
	}

	@Override
	public Object get(int index) {
		Node node = head;
		for( int i=0; i<index; i++ ) {
			node = node.getNext();
		}

		return node.getItem();
	}

	@Override
	public int size() {
		return size;
	}
```

```
}
```