CS-260 Lab #2: Double Linked Lists

General

For this lab you will gain an introduction to Abstract Data Types and coding a pointer-based doubly-linked list. The basic assignment for this lab is to code a linked-list container class that implements the Java "List" interface (for part 2a) and the Java "Queue" interface (for part 2b).

As with all of my labs, you can enhance this in any way as long as your lab implements all of the requirements as specified below.

General requirements for all labs:
At the beginning of any java files that you create or modify, include a header comment including **your name**, **creation date**, **Lab #**, and **purpose of this java file**.

Each public class and method that **you** write should have a **JavaDoc** formatted comment specifying at least what the class/method's purpose is. These comments can be fairly short at (one sentence). Use @return and @param where appropriate. You should additionally have in-line comments at any point in the code where the complexity is not easily obvious from the code itself. Methods that are implementing standard interfaces may simply inherit or copy the JavaDoc from the interface.

Your code should be **properly formatted**; please make use of the Eclipse code formatting tool. You should use **EXCEPTION HANDLING** where ever it makes sense to do so. There are some exceptions that are required by the interfaces that you are implementing.

Concepts

This lab will introduce you to Java programming techniques for a doubly-linked list ADT and give you practice using the Java Iterator Interface along with experience implementing the **Java List and Iterable interfaces. Take a look at the these interfaces in the JavaDocs as you will be implementing a subset of these interfaces.**

Background

Review the material in the online text. Be sure to complete the sections on Nodes, Overview of List ADTs and LinkList sections prior to doing this lab. I also suggest that you practice building LinkedLists using the TRAKLA2 Java applets (see the links on the class weekly outlines).

Each node in a doubly linked list contains as least **three** fields: the data, and two pointers. One pointer points to the previous node in the list, and the other pointer points to the next node in the list. The previous pointer of the first node, and the next pointer of the last node are both null.

Note that this project MUST use Java Generics; in the code below, "T" is the generic parameterized type variable from your DLList declaration. Change it accordingly.

Here's the Java class definition for a doubly linked list node; my requirement is to make this an inner class of your DLList class:

```
class DLLNode {
  // fields
    DLLNode prev;
    T data;
    DLLNode next;

  // methods

    // 3 constructors
    DLLNode() {
          this(null, null, null);
    }

    DLLNode(T d) {
          this(null, d, null);
    }

    DLLNode(DLLNode p, T d, DLLNode n) {
          prev = p;
          data = d;
          next = n;
    }
}
```

The implementations for CS260 will always have the Node structures done as inner classes of the container class. Thus Nodes will NOT be known outside of the container.

Assignment

#2a. Write a new class called DLList that IMPLEMENTS the Java **List** interface. Your implementation must use a pointer-based, double linked list implementation. You do not have to implement all of the public methods, but you must implement at least the following:

add(T), add(T, pos), clear(), contains(T), get(pos), isEmpty(), iterator(), remove(T), remove(pos), and size()

Consider the other methods a "challenge" exercise; try to do as many as you have time for.

Get as much of this working as possible; you must at least show some work on each of the above methods.

Notes for 2a:

1. Remember that you must use .equals(..) when comparing data elements, but you must use "==" when comparing Node references (i.e. curr == null).
2. You must write your container to use Java Generics; your container must define and use parameterized type definitions.
3. You must also implement the interface "CompareCount" which has only one method. I have provided this interface in the project file. You only need to count comparisons in the "contains" method. See the TestList class as an example.
4. Eclipse will generate the shell for all of the interface and inherited methods (if you do things properly). Try to implement as many of the methods as possible, even though you are not required to do all of them.

#2b. Add to the container class that you created in 2a so that it **also** implements the Java Queue INTERFACE. For this interface you must implement all of the methods, they are:

add(T), offer(T), remove( ), poll( ), element( ), peek( ).

Note that **add(T)** is already written from your list implementation, so these are just the same method. The add(T) method of your DLList class already implements this required method behavior of the Queue interface.

Discussion/notes on the entire lab:
Most of the Queue methods are implemented simply by calling methods that you have already written in your double linked list container class. The code for all of the queue methods should be very short.

The purpose of the Queue interface is to have a restricted version of the container that behaves as a Queue instead of a general list. This is then achieved by using a polymorphic variable who's static type is the Queue interface, but who's dynamic type is your double linked list.

It would be declared as follows (once you have completed your DLList class)

```
Queue queue1 = new DLList( );
```

"queue1" has a declared type of Queue, so the only methods that can be used with it are the ones from the Queue interface, so it now behaves only as a Queue instead of a general list (unless of course if a programmer casts it back to a general list).

Do some quick tests of your implementation by doing a couple of things:

1) Write a "main" that builds a DLList, adds several items into the list using the various add methods, then try removing a few items, then get some items from

the list, and finally write a "for-each" loop that walks the list printing out each item.

2) Do the same sort of quick tests on your Queue implementation.

3) Your containers should now be able to replace an ArrayList in many programs. Try bringing your DLLIst definition into the project one or two labs from your CS162 class that used ArrayLists and replace the ArrayList definition with your DLList class. If you have done things correctly, you should be able to simply change the field definition from an ArrayList to a DLList.

4) I have provided a much more complete Junit testing class for the List container. Once you have the basics going, you should use this to start testing your container. If you get errors, you should look at the failing test method and it's code in the Junit test class. That will give you some direction for uncovering the error in your code.

5) Once you have the Junit tests passing, then you must also test your code by changing the SpellChecker to now use your data structure. Edit the "Dictionary" class and change the "TestList" field to be a field of your DLList.

In our next lab we will be using the Queue you created to implement some of the required methods for a Tree implementation of the Java "Set" interface.

Submission

Turn in your complete source, project files, directory structure, etc. of you ECLIPSE project directory. Please **export** the entire project using the Eclipse export utility). This zipped file should be turned in using the same method and done for previous labs.

Requirements for 2a): For full credit, you must have the DLList class header properly defined as described above (implementing both List and CompareCount interfaces, and correctly using Generics). You must show work on all of the required List interface methods. Your code does not need to compile at this time. IT MUST BE SUBMITTED PROPERLY (watch the project export video) or you will loose points on the submission.

Requirements for 2b): You must finish the remaining requirements, modify the Dictionary class to use your DLList class, pass the Junit testing, pass the SpellChecker testing, and properly count comparisions in your "contains" method.