ETEC2101	Lab2: **Doubly-linked list class**	Points: **100** (130 points possible)
Assigned: **9/7/2016**	Due: **9/16/2016 before 8:00am**	

Objectives: Explore Java Generics and Implement our own LinkedList class[1]

Tasks:

1. Read through the "LabNotes" section below.
2. Download the sample test program from the web page. Your job is to construct the LinkedList class so it will work as shown in the comments.
3. (**25 points**) Basic LinkedList class structure
   a. Make a generic class (of type E) **LinkedList** in a package called **etec2101**.
   b. Protected attributes of LinkedList:
      i. **mBegin** and **mEnd**: two "pointers" (references in Java) to the beginning (root) and end (tail) of the linked list. Both will be *null* if the list is empty.
      ii. **mSize**: the size of the list.
      iii. The (single) constructor of LinkedList should initialize the attributes to appropriate values.
   c. Make a *nested class* protected (or private) called **Node** within the LinkedList class[2].
      i. Attributes: **mNext**, **mPrev** (both of type Node), and **mValue** (of type E).
      ii. Constructor should take an initial value of mValue and set mNext and mPrev to null.
4. (**16 points**) add two methods to LinkedList: **addToEnd** and **addToBeg**. They'll have similar structure but one adds to the end of the list, the other adds to the beginning. Both will take a value of type E.
5. (**2 points**) add a **length** getter method.
6. (**10 points**) Override the **toString** method of LinkedList. The returned string should be of the form
   `<empty>` If the list is empty
   `<[value1][value2][value3]…>` If the list is non-empty (don't include the ...)
7. (**7 points**) include an **at** method in the LinkedList class. This should take two parameters: an integer position and a Boolean indicating if we're measuring from the beginning or end. The method should throw an IndexOutOfBoundsException if an invalid position is given. Make the true parameter optional (assume fwd).
8. (**7 points**) Include an **insert** method. This should take an integer position. A new node should be created at that position ("shoving" everything else forward). Raise an exception if the index is invalid.
9. (**10 points**) add a **removeAll** method to LinkedList. This method should take a value (of type E) and remove all occurrences of it in the linked list. The method should return the number of elements that were removed.
10. (**9 points**) add a **count** method to LinkedListthat returns the number of occurrences of a given value that are in the linked list.
11. The next several methods involve the use of iterators.
    a. (**14 points**) Create a public LinkedListIterator class (nested or outside the class) that implements the Iterator interface (especially the **hasNext** and **next** method) – see the JavaDocs for an example. Make the constructor private so we can only create these internally.
    b. (**7 points**) Override the **remove** method of the LinkedListIterator. This method should remove the node containing the last value returned by next. It can only be called once before calling next again. See the JavaDocs for more details.
    c. (**5 points**) Add an **addAfter** and **addBefore** method to the LinkedListIterator class. It should take a value (of type E) which indicates that we wish to add after the current node pointed to by the Iterator. Make sure you update the LinkedList's mBegin and / or mEnd if necessary.
    d. (**8 points**) Allow forward and backwards iterators:
       i. Create an **iterator** and **riterator** method <u>in the LinkedList class</u>.
       ii. Modify the LinkedListIterator to allow forward / backwards traversal.
12. (**10 points**) Make a main program that fully tests your LinkedList class.

[1] In the real world, you probably would never have a need to implement your own LinkedList class. There is already a very polished LinkedList class (java.util.LinkedList). But…I think implementing a linked-list (or any data structure for that matter) gives you a much better understanding of how that structure works, including its efficiency and appropriateness for solving a given problem.
[2] This might be a bit clearer with the class defined in another .java file (instead of nesting it), but I just want to expose you to it here. I'll try to give you the option of *where* to put classes like this in future labs.

LabNotes:

Java Generics[3]

Suppose we write a class similar to IntList below (on the left). If we wanted a similarly-structured class that would work with float's, and without using generics, we'd have no choice but to copy-paste the entire class to a new one (FloatList let's say) and modify those lines that directly dealt with integers (as in the example, below and to the right). Notice the only thing that's changed in the copy-pasted version is in bold.

```
class IntList
{
   int[] mList;

   private int get(int index)
   {
      return mList[index];
   }

   public IntList(int size)
   {
      mList = new int[size];
      for (int i=0; i<size; i++)
         mList[i] = r();
   }
```

```
class FloatList
{
   float[] mList;

   private float get(int index)
   {
      return mList[index];
   }

   public FloatList(int size)
   {
      mList = new float[size];
      for (int i=0; i<size; i++)
         mList[i] = r();
   }
```

Usually if you ever find yourself copy-pasting code, there is a better (more elegant) solution. For this problem, the solution is to use Java Generics. In essence, we'll create a pattern for a class (or function) that has a generic type (usually called E or T [just by convention]) that will be supplied when we instantiate (or call in the case of a function). Here's the syntax:

```
public class ValueList<E>
{
      E[] mList;

      private E get(int index)
      {
            return mList[index];
      }

      public ValueList(int size)
      {
            mList = new E[size];
            for (int i=0; i<size; i++)
                  mList[i] = r();          // This only works for numeric types…
      }
}
```

So…E is basically a generic type. But, how do we tell the compiler *what* we want to use for E? The answer is when we create instances of ValueList:

```
ValueList<Integer> V1 = new ValueList<Integer>(55);
V1.printAll();
ValueList<Float> V2 = new ValueList<Float>(43);
V2.printAll();
```

Java Iterators

Make *__sure__* Jason goes through this thoroughly in class!

[3] If you've used C++ before, Java Generics are very similar to C++ templates (although they are more limited (and error-proof)).

Java Exceptions

When you get a run-time error in Java, an **exception** was thrown by a function / method you were calling. Often (especially when writing code you or someone else will use), we need to throw our own exceptions. To do this in a method you must indicate there is the possibility of throwing an exception like this:

```
public void foo() throws ArrayIndexOutOfBoundsException
{
      if (…)
            throw new ArrayIndexOutOfBoundsException();
}
```

There are a *lot* of Exception types in Java. You can see them at http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html (look under the Direct Known Subclasses part). Each of these Exception types has additional sub-classes. For example, click on *RunTimeException* (a sub-class of *Exception*) and then *IndexOutOfBoundsException* (a sub-class of *RunTimeException*) and then on *ArrayIndexOutOfBoundsException* (a sub-class of *IndexOutOfBoundsException*). The interesting thing is: you must throw an Exception object in the throw statement, but you can also throw a sub-class of Exception (or in our case a sub-sub-sub-class of Exception).

There are times when you need to (gracefully) *handle* exceptions as well. The way to do this is:

```
try
{
   // Some code that could potentially throw an IOException error or an SocketException
}
catch (IOException e)
{
   // Code to handle the exception.  e might have attributes / methods
   // that help you diagnose the problem
}
catch (xxx e)
{

}
finally
{
   // This block of code executes no matter what.  Useful for clean-up code (if any)
}
```

You must have at least one catch block to match the try. The finally block is optional.