Lab: Linked Lists

Background: `List`

A *list* (known in mathematics as a *sequence*) is an ordered, expandable collection of objects. Its elements may be accessed by index, and it may contain the same value at different positions.

Because a list data structure can be implemented in a variety of ways, `List` has been defined as an interface.

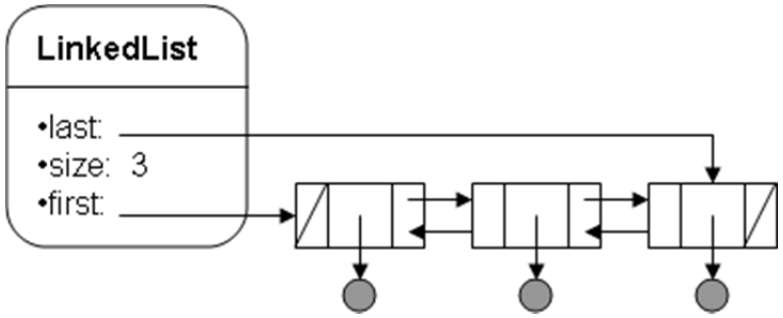
`interface java.util.List<E>`

- `int size()`
- `boolean add(E obj)` `//` appends `obj` to end of list; returns `true`
- `void add(int index, E obj)` `//` inserts `obj` at position `index` ($0 \le$ `index` $\le$ size), `//` moving elements at position `index` and higher to the `//` right (adds 1 to their indices) and adjusts size
- `E get(int index)`
- `E set(int index, E obj)` `//` replaces the element at position `index` with `obj` `//` returns the element formerly at the specified position
- `E remove(int index)` `//` removes element from position `index`, moving `//` elements at position `index + 1` and higher to the `//` left (subtracts 1 from their indices) and adjusts size `//` returns the element formerly at the specified position
- `Iterator<E> iterator()`
- `ListIterator<E> listIterator()`

Background: `LinkedList`

Java provides a second implementation of the `List` interface, and it goes by the unfortunate name of `LinkedList`. ***This is not the same as the linked list data structure consisting of a sequence of `ListNode` objects we have already investigated.*** Hence, for the rest of the course, we'll need to be careful to distinguish between "a linked list" and "a `LinkedList`".

Why did Java decide to call this class `LinkedList`? Because internally, just as an `ArrayList` is implemented in terms of an array, a `LinkedList` is implemented in terms of a linked list. Specifically, the `LinkedList` class keeps track of its size, and pointers to the first and last node in a doubly-linked list.



The `LinkedList` class also supports the following methods.

`class java.util.LinkedList<E> implements java.util.List<E>`

- `void addFirst(E obj)`
- `void addLast(E obj)`
- `E getFirst()`
- `E getLast()`
- `E removeFirst()`
- `E removeLast()`

Background: `Iterator`

Typically, we'll declare a list variable to be of the general type `List`, as follows. This gives us the flexibility to switch list implementations later, by changing only a single line of code.

```
List<Bacteria> favorites = new ArrayList<Bacteria>();
```

(Of course, if we're using methods specific to `LinkedList`, then we should declare our variable as a `LinkedList`.)

The time it takes to perform an operation on our list will depend on which implementation of `List` we choose. Consider the following code.

```
for (int i = 0; i < mooses.size(); i++)
     System.out.println(mooses.get(i));
```

This code will be extremely inefficient for one implementation of `List`. (Which one?) Therefore, we prefer to rewrite the code in terms of an `Iterator`. Java's `Iterator` interface lets you repeatedly ask for the next element of a collection. (`Iterator`s will be invaluable to us later in the course when we study `Set`s and `Map`s, whose elements cannot be accessed by index). Iterators are often useful for solving free response questions on the AP Exam; however they are not explicitly tested.

interface java.util.Iterator<E>

- `boolean hasNext()`
- `E next()`
- `void remove()` `//` removes the last element that was returned by `next`

Thus, we can rewrite our code as:

```
Iterator<Moose> it = mooses.iterator();
while (it.hasNext())
     System.out.println(it.next());
```

Calling the `List`'s `iterator` method will create a new instance of some class that implements the `Iterator` interface. The first call to the `next` method will return the first element in the list. The second call will return the one after that, and so on. You should never call `next` unless `hasNext` returns `true`.

Each of the following code segments exhibits a common error in using `Iterator`s. Can you identify and explain each of these mistakes?

```
while (mooses.iterator().hasNext())
     System.out.println(mooses.iterator().next());
```

```
Iterator<Moose> it = mooses.iterator();
Moose moose = it.next();
while (moose != null)
{
     System.out.println(moose);
     moose = it.next();
}
```

```
Iterator<Moose> it = mooses.iterator();
while (it.hasNext())
{
     if (it.next() != secretValue)
          System.out.println(it.next());
}
```

Although we'll usually just use an `Iterator` to traverse a collection, an `Iterator` can also be used to remove elements *from the original collection*.

```
Iterator<Mooose> it = mooses.iterator();
while (it.hasNext())
     if (it.next().hasBittenMe())
          it.remove();
```

Note that the `remove` method will remove the value that `next` has already returned.

Java's `ListIterator` interface extends the `Iterator` interface to include a couple extra methods. (Yes, you can extend an interface!) We might not use the `ListIterator` interface in this course and it will not appear on the AP test.

`interface java.util.ListIterator<E> extends java.util.Iterator<E>`

- `void add(E obj)` `//` adds `obj` before the element that will be returned by `next`
- `void set(E obj)` `//` replaces the last element returned by `next` with `obj`

Background: Double Jeopardy

The `ListNode` class can only be used to construct singly-linked lists. Since we'll need to make a doubly-linked list for the `MyLinkedList<E>` class, you'll first need to download DoubleNode.java. A `DoubleNode` is just like a `ListNode`, but it also stores a pointer to the previous node in the list. Take a look at the `DoubleNode` code and make sure it looks like you expect.

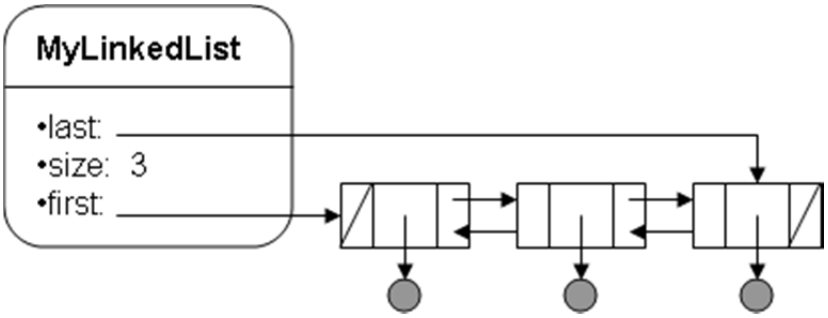
Background: `MyLinkedList<E>`

Download MyLinkedList.java. (Like `MyArrayList<E>`, `MyLinkedList<E>` will **not** implement the `List<E>`.) `MyLinkedList<E>` has the following instance variables. **Do *not* add any others**.

```
private DoubleNode first;
private DoubleNode last;
private int size;
```

`MyLinkedList<E>`'s constructors has already been written for you. A new `MyLinkedList<E>` will initially have `size` `0`, and `first` and `last` will be `null` (since there aren't any nodes yet).

When elements have been added to the list, the pointers `first` and `last` will point to the first and last nodes of the list.



Exercise: Burning the `MyLinkedList<E>` at Both Ends

To access the last element, it would be silly to start at `first` and walk down the linked list. Therefore, you should go ahead and complete two helper methods—`getNodeFromFirst`, which will find a node starting from `first`, and `getNodeFromLast`, which will find a node starting from `last`.

Next, complete the helper method `getNode`. `getNode` should call `getNodeFromFirst` if `index` is in the first half of the list, and `getNodeFromLast` otherwise.

Exercise: Hyperlinks

Now implement each of the following methods in `MyLinkedList<E>`, being sure to make appropriate use of `getNode`.

- `int size()`
- `Object get(int index)`
- `Object set(int index, Object obj)`
  `// replaces the element at position index with obj`
  `// returns the element formerly at the specified position`
- `boolean add(Object obj)` `// appends obj to end of list; returns true`
- `Object remove(int index)` `// removes element from position index, moving`
  `// elements at position index + 1 and higher to the`
  `// left (subtracts 1 from their indices) and adjusts size`
  `// returns the element formerly at the specified position`

Exercise: `MyLinkedListTester`

Download MyLinkedListTester.java and use it to test and debug your `MyLinkedList<E>` class.

Exercise: Complete the Remaining Methods in `MyLinkedList.java`

Complete the following classes within `MyLinkedList.java.` Note that these classes are not part of the `list` interface.

```
public void add(int index, E obj)
public void addFirst(E obj)
public void addLast(E obj)
public E getFirst()
public E getLast()
public E removeFirst()
public E removeLast()
```

You must write your own tester for these methods, and it must be your design – completely documented and not added to the provided tester. You must document completely why your tester actually tests the given methods and why your tests are sufficient.

If you finish early….

- Implement the `listIterator` method for `MyLinkedList`. An interface will come in handy here since you will be making a private inner class. Also, if you try to extend your iterator, you will have trouble with the state (which should be private).

- Modify your `Iterator`'s `next` method so that it throws an exception if the list has been modified since the `Iterator` was constructed.