# Lab: 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 in different positions.

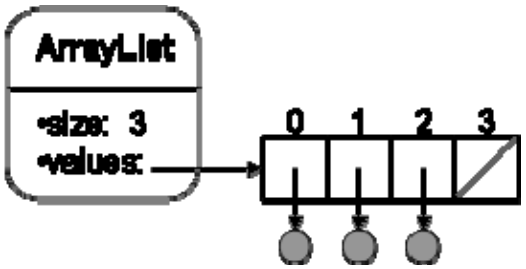
Because a list data structure can be implemented in a variety of ways, `List` has been defined as an interface.

**`interface java.util.List<E>`**

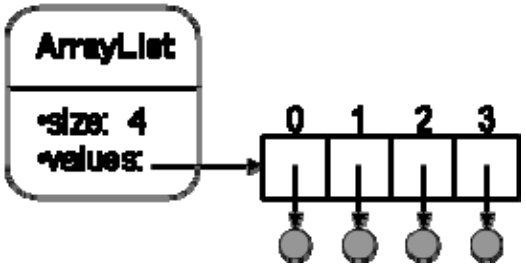
- `int size()`
- `boolean add(E obj)` `//` appends `obj` to end of list; returns `true`
- `void add(int index, E obj)` `//` inserts `obj` at position `index` (0 ≤ `index` ≤ size), `//` moving elements at position `index` and higher to the `//` right (adds 1 to their indices) and adjusts size
- `E get(int 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: `ArrayList`

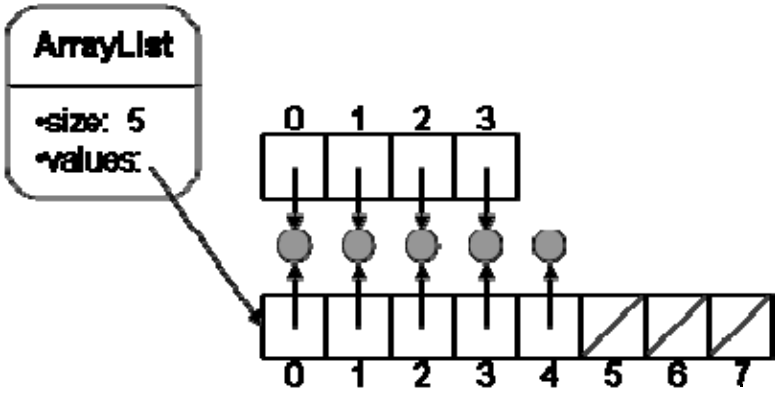
Does the `List` interface look familiar? That's because the `ArrayList` class implements the `List` interface. We know *what* we can do with an `ArrayList`, but we don't know *how* it works. How does `ArrayList` implement the `List` interface? It uses an array! The array will typically be longer than the number of elements in the list, so `ArrayList` also remembers the size of the list.




Usually, when an element is added to the `ArrayList`, a new element is added to the array, and the size is increased.



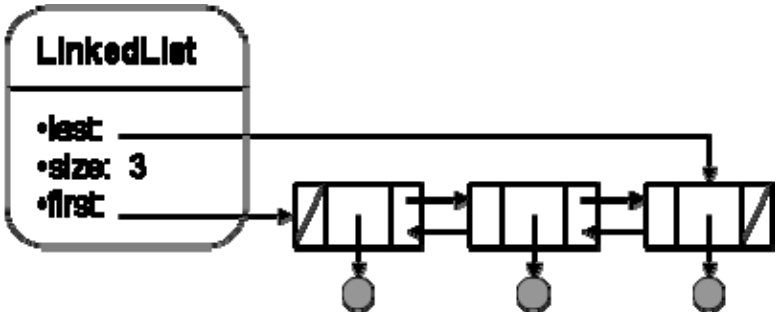

If, however, the array is full when an element is added, `ArrayList` will replace the array with one that is twice as long, and copy all of the old elements into it.




## *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.*** 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).

**`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!)

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

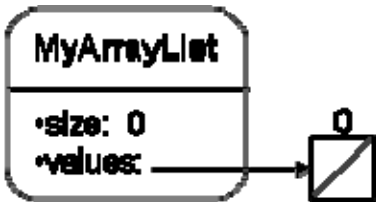
- `void add(E obj)` `//` adds `obj` before the element that will be returned by `next`
- `void set(E obj)` `//` replaces the last element returned by `next` with `obj`

## *Background: `MyArrayList<E>`'s Capacity*

Download MyArrayList.java. Unlike the real `ArrayList<E>` class, `MyArrayList<E>` will not implement the `List<E>` interface (because there are many more methods in Java's `List<E>` interface than the ones that appear in the AP Java subset). `MyArrayList<E>` has two instance variables: `size` and `values`. Do *not* add any others!

This is the first time we're implementing an `<E>` class (something we thankfully do *not* need to know how to do for the AP exam). `MyArrayList<E>` stores its values in an array of type `Object[]`(because Java stubbornly refuses to let us make a new array of type `E[]`.) So, whenever you need to return one of those values, you'll need to promise it's of type `E` (and ignore Java's warning message).

`MyArrayList<E>`'s constructor has already been written for you. A new `MyArrayList<E>` is initially empty, and `values` points to a 1-element array.




## *Exercise: `MyArrayList<E>`'s Capacity*

Complete the `doubleCapacity` helper method, which you will call whenever you need to add an element and `values` is full.

```
//postcondition:  replaces the array with one that is
//                twice as long, and copies all of the
//                old elements into it
private void doubleCapacity()
```

Complete the `getCapacity` method, which will be used to test that the capacity changes appropriately.

```
//postcondition:  returns the length of the array
public int getCapacity()
```

## *Exercise: `MyArrayList<E>`'s Capabilities*

Now go ahead and complete the following methods. Be sure to call `doubleCapacity` whenever you need to add an element and the array is full.

- `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: `MyArrayListTester`*

Download MyArrayListTester.java, which you may run to test and debug your `MyArrayList<E>` code. The tester performs random operations on both your `MyArrayList<E>` and on the real `ArrayList<E>` class. When the tester's `DEBUG` variable is set to `true`, you'll see it print out what operation is being performed and the contents of "your" list and the "real" list. The tester is great for debugging obscure errors and for showing your teacher that your code works. On the other hand, you may be better off writing your own simple test cases to do a first pass at debugging. In the end, though, your code must past the tester to get checked off.

## *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.

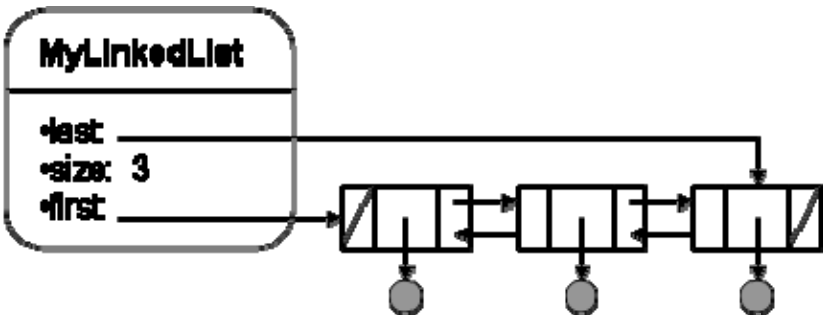
## *Background: `MyLinkedList<E>`*

Download MyLinkedList.java. (Like `MyArrayList<E>`, `MyLinkedList<E>` will not implement the `List<E>` interface.) `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`. (When you've finished the lab, you may decide to go back and complete the other methods you see listed in `MyLinkedList<E>` for additional credit.)

- `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.

## *If you finish early*

Note: It is not possible to earn an A+ grade on this lab without completing the following:

- Complete all of the methods in MyArrayList.java and MyLinkedList.java, including the `Iterator`s.

- Implement the `listIterator` method for `MyArrayList` and/or `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. (Modified means that elements have been added or deleted from the list.)