Today I will discuss a few Java more interfaces and classes. These are closely related to the array and linked list data structures that you are familiar with.

`List` interface

In everyday life, a list is just an ordered set of items. In Java, there is a generic interface `List<E>`

`http://java.sun.com/j2se/1.5.0/docs/api/java/util/List.html`

which contains several methods that you are familiar with in working with lists, such as:

- `void add(E o)`, which appends the specified element to the end of this list,
- `void add(int index, E element)` which inserts the specified element at the specified position in this list (optional operation).
- `boolean isEmpty()` which returns true if this list contains no elements, and false otherwise
- `E get(int index)` which returns the element at the specified position in this list.
- `E remove(int index)` which removes the element at the specified position in this list
- `int size()` which returns the number of elements in this list.

Have a look through the tutorial, for example:

`http://java.sun.com/docs/books/tutorial/collections/interfaces/list.html`

You can define many classes that implement the interface `List`. For example, you could define a class `AList` which would implement the methods of `List` using an array. Or you could define a doubly linked list `DLinkedList` (such as we saw in earlier lectures) which would implement these methods using a linked list. You could define these classes to be generic, so that you have some flexibility in what type of object is stored at each list position.

You (personally) do not *need* to define such classes, however. Java has generic classes `ArrayList` and `LinkedList` that implement the interface `List`. Let's have a brief look at these.

`ArrayList` class

If you want to use an array to represent a list, then why do you need a separate class? When you declare an array such as

```
 MyClass[]  myArray = new MyClass[desiredSize];
```

there are Java methods can you use with this object. There are Java classes `Arrays` and `Array` that provide methods you manipulate array objects.
`http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Array.html`
`http://java.sun.com/j2se/1.5.0/docs/api/java/util/Arrays.html`
However, these methods do not include removal and insertions, which are operations that we need for lists.

The `ArrayList` class provides such methods, in particular, implementing the methods from the `List` interface (and more!). This class implements a list using an array, but it does not use the usual `[ ]` array syntax. Instead you access an array element using a method.

```
ArrayList  a = new ArrayList(initialsize);
   :
a.add(e);     //  It will also add e to the end of the list.
              //  This will expand the array if it is full (see below).
a.add(e,i);
a.remove(j);
a.get(k);
a.clear();
a.isEmpty();
a.size();     //  returns number of elements in the list (NOT the
              //  size of the underlying array
a.ensureCapacity(minCapacity);    //  int minCapacity
```

The `ArrayList` class addresses a fundamental problem with arrays, namely that they have a fixed size which is determined when the array is instantiated. (Once you fill an array with elements, you cannot add any more.) `ArrayList` gets around this limitation by expanding the array when necessary. There are two ways this can occur. The first is if you call method `add(e)` when the array is full. The method `add(e)` will automatically expand the array by a factor of about 1.5 (50 percent increase), and adds the element the end of the current list. The second way to expand the array is to call the method `ensureCapacity()`, which expands the array (if necessary) so that the size of the array is at least as large as the integer argument to this method.[1]

In either of the above, *array expansion* works as follows: (1) a new and larger array is created, (2) the array elements – which are reference variables – are copied from the old array to the new array, (3) the array reference points to the new array, instead of to the old (full) array.

The textbook (GT p. 228) discusses one possible implementation of `ArrayList` that is based on an extendable array. The idea is that when the array is full and of size $N \geq 1$, the method `add()` calls a private method that doubles the size of the array, making its size $2N$. With this scheme the array size would always be a power of 2. [ASIDE: Indeed there is a Java class, called `Vector`, which is similar to `ArrayList` and which expands by a factor of 2 when it is full, similar to what the GT textbook proposes. Check it out - many Java programmers find it useful.]

One final point to keep in mind: using an array to implement a list is not always sensible. If you want to add an element to an arbitrary position in an array, then you need to make room for this element. This requires shifting the pointers[2] which takes time $O(N)$ in the worst case that we are adding to the front, where N is the number of elements in the list. Similarly, if we remove an element, we need to shift all the pointers to fill in the hole in the array that is left by the removed pointer. If we will be doing alot of adding and removing of objects in the list, then we might consider using an an alternative data structure, such as the following.

[1]`ensureCapacity()` *repeatedly* expands the array by 50 percent, until it is at least as large as the specified capaciy.

[2]Recall that if you have an array of objects, then the objects are not stored in the array. Rather the array stores the addresses of the objects. So you don't need to move the objects themselves, you only need to move pointers to them. (Review lecture 7, and p. 1 of lecture 8.)

`LinkedList` class

Java has a generic `LinkedList` class which also implements the interface `List`. In addition to the methods from `List` it also has methods `addFirst()`, `addLast()`, `remove()`, `clear()`, `contains()`, and others. Because it is generic, you can have a list of objects whatever type you want: `Dog`, `Animal`, `Beagle`, `Rectangle`, `Shapes`, etc.

What's nice about the `LinkedList` class is that, as a user, you don't have to manipulate reference variables. However, you should be careful and keep in mind how linked lists work. For example, suppose you want to `get` the i-th element from a linked list. The only way to do it is to start at the beginning of the list and walk down the list, following the next references. In the worst case that the item you are getting happens to be at the end of the list, this `get` method runs in $O(N)$ time. So, if you are doing alot of `get(i)` calls, then you probably want to use an `ArrayList` rather than a `LinkedList`.

`Iterator` interface

Many algorithms that are based on collections of items – including linked lists as well as arrays, use a pointer or index variable to step through the collection. Because stepping through a collection is so common, Java defines a generic interface `Iterator<E>`:

```
 interface Iterator<E>{
   boolean hasNext( E )
   E       next();    //  returns the next element and advances
   void    remove();  //  removes the next element
 }
```

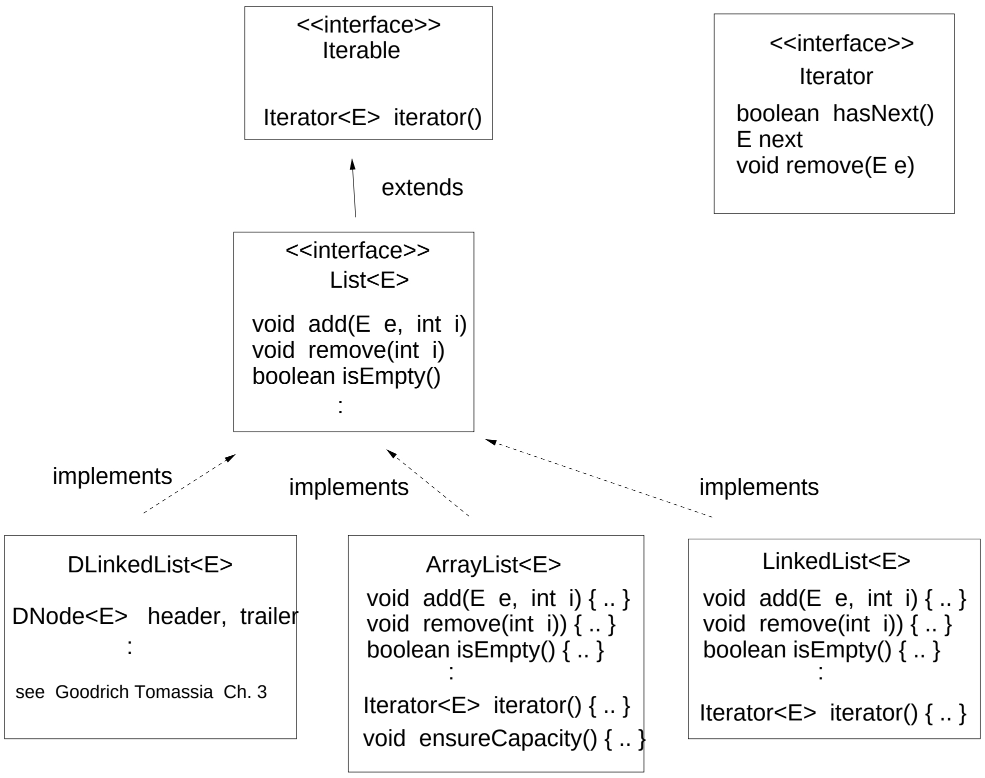
To understand why this is an interface and not a class, note that you may wish to define these methods in a way that depends on the collection you are iterating through. It may be an `ArrayList`, `LinkedList`, or some other class.

`Iterable` interface

Interestingly, `ArrayList` and `LinkedList` do not implement `Iterator`. Instead, `Iterator` makes it appearance as a *return type* in the method of another commonly used generic Java interface called `Iterable`:

```
interface Iterable<T>{
  Iterator<T>  iterator();
}
```

The interface `Iterable` is extended by the interface `List`, which we saw earlier is implemented by many classes, including `LinkedList` and `ArrayList`. This means that `LinkedList` and `ArrayList` each implement the method `iterator()` which (by the above definition of the `Iterable` interface) returns a generic object of type `Iterator`.



Example: list of rectangles

Let's look at an example of how iterator works. We define a class `RectangleList` which has two fields: a linked list of `Rectangle` objects, and an iterator over this list. The class also has four methods: a constructor, a method for adding a rectangle to the list, a method for displaying the list (which displays the height and width of each rectangle object), and a main method.

```
import java.util.Iterator;
import java.util.LinkedList;

public class RectangleList{

   private LinkedList<Rectangle>  myRectangleList;
   private Iterator<Rectangle>  iter;

   public RectangleList(){
      myRectangleList = new LinkedList<Rectangle>();
   }

   public void addRectangle(Rectangle r){
      myRectangleList.add(r);
   }
   public void display(){
      Rectangle r;
      iter = myRectangleList.iterator();

      while (iter.hasNext() ){
         r = iter.next();
         System.out.println( r.getWidth() + "  " + r.getHeight());
      }
   }

   public static void main(String[] args){
      RectangleList myList = new RectangleList();
      myList.addRectangle(new Rectangle(1.0, 4.0));
      myList.addRectangle(new Rectangle(2.0, 3.0));
      myList.addRectangle(new Rectangle(3.0, 2.5));
      myList.display();
   }
}
```

Enhanced for loop

In the `display()` method above, we used an iterator to explicitly walk through the elements in the list by following `next` references. This is not the only way go through a list, however. We could replace the `while` loop above by an *enhanced for loop*, which essentially does the same thing.

```
   for (Rectangle r1: myRectangleList){
      System.out.println( r1.getWidth() + "  " + r1.getHeight());
   }
```

You may now be asking why should you bother using the iterator? Why not just use the enhanced for loop all the time. The answer is that you sometimes want to iterate through the list, then pause, do something else, continue, pause, do something else, etc. The enhanced for loop doesn't allow this, but explicitly using the iterator does!

Final note: you can define several iterators if you wish. These could (at any given time) be at different points in the list.