Lesson 24: *ArrayList*

You will recall from Lesson 41 the *ArrayList* is one of several classes that implement the *List* interface. As its name suggests, *ArrayList* also involves arrays. Basically, everything we learned in Lessons 18 and 19 concerning arrays can also be applied to *ArrayList* objects, however, with slightly different methods.

Comparing *ArrayList* to ordinary arrays:

So, a legitimate question to ask at this point is, "Why clutter our brains with a new set of commands for the *ArrayList* if it serves the same purpose as do ordinary arrays?" We are going to discuss the advantages of the *ArrayList* class over ordinary arrays and, to be fair, its disadvantages.

Advantages:

Ordinary arrays are fixed in size. When we create an array, we anticipate the largest possible size it will ever need to be, and when instantiating the array, dimension it to that size. We call this the physical size of the array and it always remains that size even though at some point in your program you may wish to only use a portion of the array. The size of that portion is called the logical size. Your own code must keep up with that size. By contrast, the *ArrayList* expands and contracts to meet your needs. If you remove items from or add items to your *ArrayList*, the physical and logical sizes are always identical. This could be very important if you wish to be conservative of memory usage. With memory being so abundant and inexpensive today, this is no longer the advantage it once was.
One of the *add* methods allows very easy insertions of new items in the interior of the list without the nuisance of having to pre-move preexisting items.
A final advantage is that iterator objects are provided, whereby we can easily traverse the list.

Disadvantages:

ArrayList can only store objects. If we wish to store primitives such as integers, *double*s, or *boolean*s, they must be converted into their wrapper class counterparts (see Lesson 21). This was once a nuisance because they had to be converted manually, but is now circumvented with the advent of Java 5.0+ and its autoboxing feature. Similarly, when we retrieve things from an *ArrayList*, they come out as objects. "Big deal", you say….certainly, if we store objects in the list, then we expect to get objects back when we retrieve from the list. Yes, but it's worse than one might think. When retrieving an object from the list, it doesn't come back as the same type object that was originally stored. Rather, it comes back as an *Object* type object (recall the cosmic superclass from Lesson 35). It will be necessary to cast it down to its original object type…yet another nuisance (partially circumvented by Java 5.0+ if type parameters are used as discussed below).

So, what are the methods we use with *ArrayList*? Look back at Lesson 41 on the *List* interface. Since ***ArrayList* implements the *List* interface**, those are the methods. We will now offer sample usage and/or discussion of several of the more important methods.

In each of the following examples we are to assume an *ArrayList* object has been created via
ArrayList aryLst = new ArrayList( ); or List aryLst = new ArrayList( );

Type parameters:

With the addition of type parameters to Java 5.0+, it is also possible to create an *ArrayList* object as follows (See Appendix AF for the related topic of generics.):

```
ArrayList<String> aryLst = new ArrayList<String>( );
```

With the advent of Java 7.0 this line of code can be shortened to:

```
ArrayList<String> aryLst = new ArrayList<>( );
```

The *<String>* part indicates that objects we add to the list can only be *String* types. (Instead of *String* we could use any object type.) This insures "type safety" and would result in a compile time error if we ever tried to add some other type object to *aryLst*. Type parameters also remove the burden of casting *Object* type objects retrieved from a list back to their original type. Unfortunately objects retrieved from an *ArrayList* using an iterator must still be cast **unless** the iterator also uses generics.

In the following examples, assume that only *Integer* type objects have been stored in the list and that *aryLst* was created with *List<Integer>aryLst = new ArrayList<>( );*

void add(Object o) //**signature**

Example:

```
aryLst.add(13);
//pre Java 5.0
Integer jw = new Integer(j);
aryLst.add(jw); // add jw to the end of the list
```

void add(int index, Object o) //**signature**

Example:

```
aryLst.add(3, 13);
//pre Java 5.0
Integer jw = new Integer(j);
aryLst.add(3, jw); //inserts jw at index 3 after moving the existing object at index
//3 and greater, up one notch.
```

Object get(int index) //**signature**

Example:

```
int q = aryLst.get(3);
//pre Java 5.0
Object obj = aryLst.get(3); //retrieve object at position 3
Integer qw = (Integer)obj; //cast down from Object to Integer
int q = qw.intValue( ); //convert back to int type.
```

Object remove(int index) //**signature**

> **Example:**
> int q = aryLst.remove(3);
> //pre Java 5.0
> Object obj = aryLst.remove(3); // removes object at position 3 (then compacts the list)
> Integer qw = (Integer)obj; //cast down from Object to Integer
> int q = qw.intValue( ); //convert back to int type.

Object removeLast( ) //**signature**

> **Example:**
> int q = aryLst.removeLast( );
> // pre Java 5.0
> Object obj = aryLst.removeLast( ); // removes object at end of list and returns that object
> Integer qw = (Integer)obj; //cast down from Object to Integer
> int q = qw.intValue( ); //convert back to int type

.

Object set(int index, Object o) //**signature**

> **Example:**
> int q = aryLst.set(3, 13);
> //pre Java 5.0
> Integer jw = new Integer(13);
> Object obj = aryLst.set(3, jw); // replaces object at position 3 with jw and returns original object
> Integer qw = (Integer)obj; //cast down from Object to Integer
> int q = qw.intValue( ); //convert back to int type.

boolean isEmpty( ) //**signature**

> **Example:**
> aryLst.isEmpty( ); // returns true if there are no objects in the list

int size( ) //**signature**

> **Example:**
> aryLst.size( ); //returns the number of objects in the list

void clear( ) //**signature**

> **Example:**
> aryLst.clear( ); //removes all objects from the list

With Java 5.0+, autoboxing makes the following three methods (signatures are shown) easy to use. For example, if we are seeking the integer 13, the argument sent to the method would simply be 13.

> int indexOf(Object o)
> int lastIndexOf(Object o)
> boolean contains(Object o)

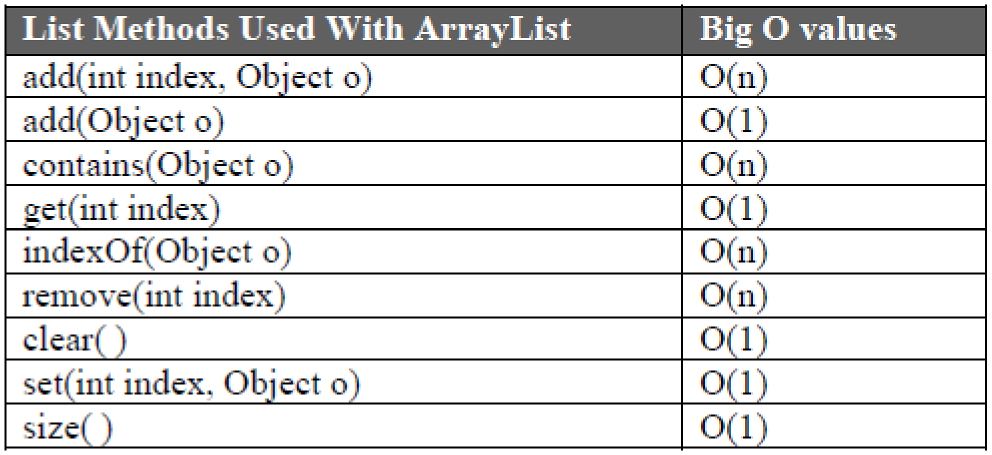
Constructors:

> ArrayList( ) //Default constructor
> ArrayList(Collection c) //Constructs a list with the elements of the specified collection.
> ArrayList(int j) //For fast storage, preallocates space for j elements; however, more

```
            //than j can be stored.
```

Big O values:

Determine the efficiency of algorithms using *ArrayList* methods with the following table:



List Methods Used With ArrayList	Big O values
add(int index, Object o)	O(n)
add(Object o)	O(1)
contains(Object o)	O(n)
get(int index)	O(1)
indexOf(Object o)	O(n)
remove(int index)	O(n)
clear( )	O(1)
set(int index, Object o)	O(1)
size( )	O(1)

Two very important methods, *iterator( )* and *listIterator( )* will be discussed in Lesson 23.