Lesson 23: *List interface*

Classes that implement the *List interface*:

Java includes three classes (*LinkedList*, *ArrayList*, and *Vector*) that **implement** the *List* ***interface*** This *interface* and the three classes are made available by importing *java.util.**:

List interface methods:

***List* method signature**	**Action**
void add(int index, Object o)	Inserts the object o at the position specified by *index* after all existing objects at that index and greater are moved forward one position.
boolean add(Object o)	Appends o to the end of the list. Returns *true*.
boolean addAll(Collection c)	Appends *c* to the end of the list.
void clear( )	Removes all elements from the list.
boolean contains(Object o)	Returns *true* if this list contains the specified object.
boolean containsAll(Collection c)	Returns *true* if this list contains all of the elements in *c*.
boolean equals(Object o)	Returns *true* if *List* object *o* has the **same elements** in the **same order** as the present list. If *o* is any other collection such as a *Set*, a *false* is returned.
Object get(int index)	Returns the object at the position specified by *index*.
int indexOf(Object o)	Returns the index of the first occurrence of the specified object…searching from left to right….or –1 if not found.
boolean isEmpty( )	Returns *true* if this list contains no objects.
Iterator iterator( )	Returns an *Iterator* object for this list**…Important…to be discussed in the next chapter.**
int lastIndexOf(Object o)	Returns the index of the **first** occurrence of *o*… when searching from **right to left**….or –1 if the object is not found.
ListIterator listIterator( )	Returns a *ListIterator* object for this list**… Important…to be discussed in the next chapter.**
Object remove(int index)	Removes the object at the position specified by *index* and returns the object.
boolean remove(Object o)	Remove the first occurrence of o (searching from left to right).
boolean removeAll(Collection c)	Removes from this list the first occurrence of all elements in *c*.
boolean retainAll(Collection c))	Retains only the elements in *c*.
Object set(int index, Object o)	Replaces the object at the position specified by *index* with o…Returns old object.
int size( )	Returns the number of objects in the list
Object[] toArray( )	Returns an *Object* array in the proper sequence.

Printing a *List* object:

It is possible to print the contents of an entire list named *lst* with *System.out.println(lst)*. A typical printout would look like the following if characters 'a' – 'g' are stored as the individual elements of the list (notice the surrounding square brackets):

```
[a, b, c, d, e, f, g]
```

Creating a *List* object:

There are three ways to **create a *List* object** since there are three classes (mentioned above) that implement the *List* interface.

1. List lst = new LinkedList( );
2. List lst = new ArrayList( );
3. List lst = new Vector( );

This is basically an array with an initial *capacity* and having the ability to increase its size with a specified *increment* amount when a new storage attempt exceeds the present size.

The *Arrays.asList( )* method:

If *ary* is an ordinary, singly-subscripted array, then *Arrays.asList(ary)* will return a *List* object in which the elements of the list are the elements of *ary*. It is also possible to make very simple lists easily as shown by the following:
List lst = Arrays.asList("A", "B", "C", "D");

It should be noted that it is not possible to *add* or *remove* elements from the resulting *List* object; however, it is possible to use the *set* method to change values. Iterators can be produced from the list and used to step through the items in the list.

Important features:

Here are some salient facts about these three *List* types:

1. The lists consist of nothing but objects of **type *Object***. **Any** type object can be stored in a list; however, they are immediately and automatically converted into *Object* type objects for storage.

2. A list can have **different** types of objects initially stored in it; however, in actual practice most lists are restricted to just one type.

3. The objects retrieved from a list generally need to be **cast** to a **specific** object type before being used. For example, unless the object *lst* was created using type parameters, *Double d = lst.get(2);* won't work but *Double d = (Double)lst.get(2);* will.

4. Sort *List*, *ArrayList*, *LinkedList*, or *Vector* object *obj* with *Collections.sort(obj);*

On your own:

This lesson has been purposely left vague and sparse. The exercises that follow are all "doable" using the information in this lesson, especially the descriptions of the methods in the interface on the preceding page. The student is on his own to "ferret out" the information needed to answer the questions. As an example of doing this, consider problem 3 on the next page. Even though *Iterator* objects have not been presented yet, just look over the methods in the interface and see which one deals with an *Iterator*. No knowledge of what an *Iterator* is or how it works is actually needed.