Lesson 43….. *Iterator*/*ListIterator*

What is iteration?

If someone asks, "Have you studied iteration yet?", you should answer, "Yes." **Loops** are iteration structures. Each pass through a loop is an iteration. Granted, if you say "iteration," it sounds like you are trying to impress someone with your vocabulary; nevertheless, iteration is a continual repeating of something…so looping fits the definition.

Position of an iterator:

Here, we wish to loop through the various objects in a list... or in other words, simply to step through positions of these objects. The concept of **position** for an iterator is central to our understanding of *Iterator* and *ListIterator* objects. Consider the following objects in a list:

```
A B C D E F G H I J
```

These objects have positions as indicated below:

```
0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J
```

However, this familiar indexing scheme for the object in a list is **not** the scheme for indexing the **position of an iterator**.

For an iterator, think of the cursor in a word processor. The cursor is never **on** any particular character; rather, it is always **between** two characters. And so it is with the position of an iterator: it is always **between** two of its list objects.

```
A B C,D E F G H I J
```

Notice that we are symbolically letting a comma indicate the iterator position and that it is **between** the C and D objects above.

But what is the index of this position? In the above example, it is 3 since it comes **before** the object of index 3…D in this case.

Possible positions:

In general the possible positions of an iterator are:

1. Just before the first item (index would have a value of 0)

2. Between two adjacent list objects

3. Just after the last item (index would be the size of the list)

Two iterator interfaces:
We have two iterator interfaces available to us. The first is very simple and the other is considerably more robust.

1. *Iterator* …….. interface specified in *java.util.Iterator*
   Recall that in the *List* interface that we previously studied, one of the method's signature was *Iterator iterator( )*. We **create** an *Iterator* object by calling this method.
2. *ListIterator*….. interface specified in *java.util.ListIterator*
   Recall that in the *List* interface that we previously studied, one of the method's signature was *ListIterator listIterator( )*. We **create** a *ListIterator object by calling this method.*

Iterator

The *Iterator* interface has only three methods. It is commonly used with *List* objects but can be used with other classes as well.

Iterator method signature	Action
boolean hasNext( )	Returns *true* if there are any items following the current position.
Object next( )	Returns item following current position and then advances the position… providing there is at least one item after the current position.
void remove( )	Removes the item returned by last successful *next( )* …providing there were no other intervening *remove* operations.

Code example:

```
List lst = new ArrayList( ); //first, create a List object
… add items to the list…
Iterator itr = lst.iterator( ); //now we have the object we want, itr.
//The position of the Iterator is at the head of the list (preceding the first object)
```

ListIterator

The *ListIterator* **has the above three methods** (action for *remove* is slightly different) **plus** some additional ones. It is only used with *List* objects, hence the name ***List****Iterator.*

ListIterator method signature	Action
void remove( )	Removes the item returned by last successful *next( )* or *previous( )* …providing there were no intervening *add* or *remove* operations.
boolean hasPrevious( )	Returns *true* if there are any items preceding the current position.
Object previous( )	Returns the item preceding the current position and moves the position back.
int nextIndex( )	Returns index of next item (-1 if none). In effect this is the current position of the *ListIterator*.
int previousIndex( )	Returns index of previous item (-1 if none).
void add(o)	Insert object o just left of the current position.
void set(o)	Replaces the last item returned by last successful *next( )* or *previous( )* with object o …providing there were no intervening *add* or *remove* operations.

Creating an iterator:
An iterator object (either *Iterator* or *ListIterator*) is created in the following way:

Code example:

```
List lst = new ArrayList( ); //first, create a List object
ListIterator itr = lst.listIterator( ); //now we have the object we want, itr.
//The position of the ListIterator is at the head of the list (preceding first object)
```

Special usage of the *for*-loop:

It is possible to use either an *Iterator* or a *ListIterator* object directly with a for-loop.

Code example:

```
List lst = new ArrayList( );
…add some items to the list…
ListIterator itr = lst.listIterator( );

for( Integer iw = (Integer)itr.next( ); itr.hasNext( ); iw = (Integer)itr.next( ) )
{
        …do something using iw…
}
```

Using type parameters with an iterator:
Notice in the following code example a cast is required in the last line even when type parameters are used to instantiate the *ArrayList* object:

Code example:

```
ArrayList<String>aryLst = new ArrayList<String>( );
aryLst.add("hello");
aryLst.add("goodbye");
Iterator itr = aryLst.iterator( );
String s = (String)itr.next( );          //Won't compile without this cast
```

If the iterator is built using a type parameter, the cast is not required:

Code example:

```
ArrayList<String>aryLst = new ArrayList<String>( );
aryLst.add("hello");
aryLst.add("goodbye");
Iterator<String>itr = aryLst.iterator( ); //Type parameter used here
String s = itr.next( );                   //Cast not required
```

Although the above two examples used *Iterator*, the same is also true for *ListIterator*.

Enhanced for-loop example:

```
ArrayList<String>aryLst = new ArrayList<String>( );
aryLst.add("hello");
aryLst.add("goodbye");
for(Iterator iw : aryLst) //An iterator is not explicitly created here; however, the
{ … }                     // code behind the enhanced-for uses an iterator to step
                          // through the items in the list.
```

Warning:

- When running an *Iterator* or *ListIterator* on a list, don't modify the list with *List* methods. Errors may result.
- Also, it's not a good idea to simultaneously have two *Iterator*s and/or *ListIterator*s where both of them modify the list in some way. Errors may result.