**Iterator Worksheet**

Preparation for IteratorLab

The Collection Framework specifies that some Collections, including Lists and Sets, contain an inner, private object, called an "**iterator**," that allows the data structure to traverse through its data items one-by-one. Programmers do not need to know how the data is actually structured. All the programmer needs to know is how to call the iterator’s methods, which is what the API is for. That’s the point. This lab introduces you to iterators and what they can do. (In fact, you have used iterators before when you used the Scanner class to read data from a file.) Here are relevant methods for Iterators:

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

* Iterator<E> iterator() *// every list can instantiate a private Iterator object*
* ListIterator<E> listIterator()

**interface java.util.Iterator<E>**

* **boolean** hasNext()
* E next()
* **void** remove() *// removes the last element that was returned* by next

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

Methods in addition to the Iterator methods

* **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

1. Instantiate a concrete subclass of List<E> named theList which stores Student objects. Then instantiate an iterator on theList:

List<Student> theList = new ArrayList<>();

Iterator<Student> it = theList.iterator();

1. Now use the iterator’s methods to println every item in theList:

**while(it.hasNext())**

**System.out.println(it.next());**

1. The best way to understand how iterators behave is to imagine a pointer that points \*between\* two items. When the iterator is instantiated, the imaginary pointer is \*before\* the first item in the list. Try it out:

Given: theList ["A", "B", "C", "D", "E"]  
 Given: ListIterator<String> it = theList.listIterator();

Imagine the iterator in the theList this way: ^**A B C D E**

Now show the list and the location of the pointer:

it.next(); //\_\_\_A^ B C D E\_\_\_\_\_\_\_\_\_\_\_\_

it.next(); //\_\_\_A B^ C D E\_\_\_\_\_\_\_\_\_\_\_\_

it.remove();//\_\_\_A C^ D E\_\_\_\_\_\_\_\_\_\_\_\_

it.next(); //\_\_\_A C D^ E\_\_\_\_\_\_\_\_\_\_\_\_

it.add("X"); //\_\_\_A C X D^ E\_\_\_\_\_\_\_\_\_\_\_\_

1. The it.remove() method *“removes the last element that was returned by next.”*
2. it.add(E) *"adds obj before the element that will be returned by next."* The add(E) method works with the imaginary pointer in all positions, in front, in the middle, and at the end, just as you would expect.
3. Let lsit be a ListIterator<E> on myList that contains two Strings ["A", "B"].

After each command, show the state of the iterator lsit:

ListIterator<String> lsit = list.listIterator();//\_[^”A”, “B”]\_

lsit.add("X"); //\_[“X”^, “A”, “B”]\_

lsit.next(); //\_[“X”, “A”^, “B”]\_\_

lsit.add("Y"); //\_[“X”, “Y”, “A”^, “B”]\_

lsit.next(); //\_[“X”, “Y”, “A”, “B”^]\_

lsit.add("Z"); //\_[“X”, “Y”, “A”, “Z”, “B”^]\_

lsit.next(); //NoSuchElementException

1. Assume al is an ArrayList<String> and you have to write code to delete every "Bob"   
   from al. Below are three solutions. Decide if each works or not. If it doesn’t work, name the Exception it throws.

Student A:

for(int i = 0; i<al.size(); i++)

if( al.get(i).equals("Bob") )

al.remove( i );

Student B:

ListIterator<String> li = al.listIterator();

while( li.hasNext() )

if( li.next().equals("Bob") )

li.remove();

Student C:

int position = 0;

for(String str: al)

{

if(str.equals("Bob"))

al.remove( position );

position++;

}

1. A *for-each* loop or an Iterator: how do you choose?
2. Using an Iterator or a ListIterator: how do you choose?
3. Code written before Java 1.5 does not change (wrap) int to Integer automatically. The programmer had to do that every time. Code written after Java 1.5 wraps int to Integer automatically. Since we are using a current version of Java, you don't have to worry about that.

**IteratorLab Assignment**

Complete the IteratorLab shell, using either for-each loops or iterators, not regular for-loops. Notice Lines 11-12, where they made the for-each loop backwards compatible with primitives. The replaceNeg method on Line 17 replaces all negative numbers with 0.