Iterators

Iterator:

An ArrayList (as well as Vector) allows you to select an object from a list of objects (a container) using natural numbers. But what if you’d like to select from an object using some other criterion? An iterator is such an alternative. An iterator moves, one at a time (usually only in one direction) through the container and selects objects. The Java Iterator interface provides the following methods:

- **Object next()** - returns the next element in the container
- **boolean hasNext()** - checks if there is a next element
- **void remove()** - (optional operation). removes the element returned by next()

Here is a code fragment demonstrating an iterator in Java. We fill ArrayList with integers, and sum them up using an iterator instead of a for-loop:

```java
ArrayList num = new ArrayList();
for(int i = 0; i < 5; i++)
    num.add(new Integer(i));

Iterator it = num.iterator();
// num.add(new Integer(5)); ConcurrentModificationException
int sum = 0;

while (it.hasNext())
    sum += ((Integer) it.next()).intValue();

System.out.println(sum);
```

The iterator is indifferent to the underlying data structure. In the next fragment we fill ArrayList with various objects and then remove a few of them:

```java
ArrayList num = new ArrayList();
for(int i = 0; i < 5; i++)
    { 
        num.add(new Integer(i));
        num.add("str" + i);
    }

Iterator it = num.iterator();

while (it.hasNext())
{
    Object tmp = it.next();
    if (tmp instanceof Integer)
        if (((Integer) tmp).intValue() == 2)
            it.remove();
}

for(int i = 0; i < num.size(); i++)
    System.out.println(num.get(i));
```
ListIterator:

Note, the Iterator given the above is a restrictive type of iterator that allows only to mover forward. The ListIterator is the more powerful iterator that allows traversing a container in any direction. The Java ListIterator interface provides the following methods:

- Object next() - returns the next element in the container;
- boolean hasNext() - checks if there is a next element;
- int nextIndex() – returns the index of the next element;
- Object previous() - returns the previous element in the container;
- boolean hasPrevious() - checks if there is a previous element;
- int previousIndex() – returns the index of the previous element;

and a few optional operations like add(), remove(), set(). It is generally considered a dangerous to change the contents while using an iterator.

Implementation:

The whole idea of the iterator is to provide an aggregated data and at the same moment hiding the underlying representation.

The most important to understand is that the Iterator (or ListIterator) is an object. It is considered to be a copy of the object that implements this interface. In addition, the iterator is responsible for keeping track of the current element.

Let us implement the ListInterator in the LinkedList class.

First of all we add a new method to the LinkedList class:

```java
public ListIterator listIterator(int index) {
    return new ListItr(index);
}
```

that returns a list-iterator of the elements in this list, starting at the specified position. If index = 0, we start with the first node, if index == size we start with the last node. The ListItr class will be implemented as a private inner class:

```java
private class ListItr implements ListIterator {
    private Stack stk;
    private Node next;
    private Node lastReturned;
    private int nextIndex;
    ...
}
```
where

- `lastReturned` is the reference to the node recently returned;
- `next` is the reference to the next node in the list;
- `nextIndex` is the index of the next node that changes from 0 to size;
- `stk` is the stack for storing references to all nodes preceding `next`.

Here is the implementation of the `next()` method:

```java
public Object next()
{
    if (!hasNext()) throw new NoSuchElementException();

    lastReturned = next;
    stk.push(next);
    next = next.getNext();
    nextIndex++;
    return lastReturned.getData();
}
```

Note, that before we advance the `next` reference we store it on the stack. See `LinkedListI1.java` for details.

The `LinkedListI1.java` class contains all methods, but `add()`, `remove()`, `set()`, implementation of which requires further structural changes. The problem with these methods is that user has to different ways to change the content of the linked list: either using the `LinkedList` or `ListItr` class functionalities. To avoid this duality, we would allow a user to make changes to the linked list only using the iterator. In order to support this on the programming level, we add two new fields, `count` to the `LinkedList` class and `expectedCount` to the `ListItr` class. Once a user adds (or removes) a new node to the `LinkedList` class we will increase `count` without changing `expectedCount`. In the `ListItr` we will always check whether or not `expectedCount=count`. If they not equal we will throw `ConcurrentModificationException`. If a user adds (or removes) a node using the iterator, we will increase both `expectedCount` and `count`. Here is the implementation of the `remove()` method, that removes a node returned either by `next()` or `previous()`

```java
public void remove()
{
    if (expectedCount != count)
        throw new ConcurrentModificationException();
    LinkedListI2.this.remove(lastReturned.getData());
    if (next == lastReturned)
        next = next.getNext();
    else
        nextIndex--;
    expectedCount++;
    lastReturned = head;
}
```

See `LinkedListI2.java` for a complete implementation.