Simple List Iterators

Simple iterators, iterator interfaces, implementing a simple iterator in a linked list class, using a simple iterator

Example of inefficient code

Why is this client code exceptionally inefficient?

```
List<Integer> slow = new LinkedList<Integer>();
// add some integers to the list

int numEvens = 0;
for (int i = 0; i < slow.size(); i++) {
   if (slow.get(i)% 2 == 0)
      numEvens++;
}
```

- For *each* iteration of the outer loop, there is sweep through the list starting from its beginning to compute slow.get(i).
- The computational complexity is $O(n^2)$.

Overall task: provide clients of custom linked lists O(n) traversals

- Simply traversing a linked list should be O(n).
- It is easy to traverse a linked list when inside the list class.
- But with our old SimpleLinkedList classes, there is no way that a client can traverse a list without a sequence of calls to get(int index).
- An **iterator** is an object that can traverse a custom list.
- java.util has an Iterator<T> interface that spells out iterator behaviors.

Iterators and "simple iterators"

- The Iterator<T> interface in java.util declares 4 methods:
  1. boolean hasNext()
  2. T next()
  3. void remove()
  4. void forEachRemaining()
- The remove() and forEachRemaining() methods are the most difficult to define.
- A *simple iterator* is an iterator without remove() or forEachRemaining().
- The purpose of simple iterators is to traverse custom lists.
- *Note:* You cannot use a for-each loop to do the traversal on custom lists. You can use for-each loops for traversals on the built in collections classes.

Simple iterator interfaces

You can define an iterator interface with generics.

```
public interface SimpleIterator<E> {
  /**
   * Does the iterator have more elements to traverse?
   * @return true if the list iterator has more elements.
   */
  boolean hasNext();
  /**
   * Gets the next element in the list and pushes the iterator
   *   down the list.
   * @return the next element in the list.
   * @throws NoSuchElementException if there's no next element
   */
   E next();
}
```

Implementing a simple iterator

Inside the outer class SimpleLinkedList<E>, you can define a private inner class. The name of the inner class in this example is Cursor. The inner class should implement SimpleIterator<E>.

```
private class Cursor implements SimpleIterator<E> {
   Node traveler = head;

   public boolean hasNext() {
      return traveler != null;
   }
   public E next() {
      if (traveler == null)
          throw new NoSuchElementException();
      E data = traveler.data;
      traveler = traveler.next;
      return data;
   }
}
```

Making the simple iterator available to clients

Add a new method to the list class that simply returns a new instance of the inner iterator class.

```
public SimpleIterator<E> iterator() {
    return new Cursor();
}
```

Using the simple iterator

This code can go in any client class:

```
public void example() {
    SimpleLinkedList<String> list =
                          new SimpleLinkedList<String>();
    list.addToFront("abc");
    list.addToFront(”de");
    list.addToFront(”fg");
    SimpleIterator<String> i = list.iterator();
    while(i.hasNext()) {
        System.out.println(i.next());
    }
}
```

Output:

```
fg
de
abc
```

Take care *when* you get the iterator

This code is similar to the last:

```
public void example() {
    SimpleLinkedList<String> list =
                          new SimpleLinkedList<String>();
    SimpleIterator<String> i = list.iterator();
    list.addToFront("abc");
    list.addToFront(”de");
    list.addToFront(”fg");
    while(i.hasNext()) {
        System.out.println(i.next());
    }
}
```

Output:

Simple iterator issues

- Client code: always get the iterator immediately before you need to use it.
- Client code: never get an iterator, change the list, then use the same iterator. Changing lists can destroy the integrity of the cursor. For example:
  - The “old” iterator can miss newly added list items.
  - The “old” iterator can visit items that are no longer in the list.
- Iterators are cheap. Get a new one whenever you need one.

Alternative implementations

Instead of defining your own SimpleIterator interface, your list class can implement Iterable<E> (from java.util).

```
public interface Iterable<E>() {
   Iterator<E> iterator();  // only 1 method declared
}
```

Then declare your list class to implement Iterable<E>.

```
public class MyList<E> implements Iterable<E> {
```

Now the iterator from your class will support all the Iterator methods (not just next() and hasNext()).