```
 1 package DataStructures;
 2
 3 import Exceptions.*;
 4
 5 // ListItr interface; maintains "current position"
 6 //
 7 // ******************PUBLIC OPERATIONS*********************
 8 // void insert( x )     --> Insert x after current position
 9 // void remove( x )     --> Remove x
10 // boolean find( x )    --> Set current position to view x
11 // void zeroth(         --> Set position prior to first
12 // void first( )        --> Set current position to first
13 // void advance( )      --> Advance
14 // boolean isInList( )  --> True if at valid position in list
15 // Object retrieve( )   --> Return item in current position
16 // ******************ERRORS********************************
17 // Exceptions thrown for illegal access, insert, or remove.
18
19 public interface ListItr
20 {
21     void    insert( Object x ) throws ItemNotFound;
22     boolean find( Object x );
23     void    remove( Object x ) throws ItemNotFound;
24     boolean isInList( );
25     Object  retrieve( );
26     void    zeroth( );
27     void    first( );
28     void    advance( );
29 }
```

Figure 6.12 Interface for the abstract list iterator

The iteration mechanism that Java would use is similar to the following (since `ListItr` is an interface, the `ListItr` that follows `new` would be replaced by some class that implements the `ListItr` interface)

```
// Step through List, using abstraction and an iterator
ListItr itr = new ListItr( theList );
for( itr.first( ); itr.isInList( ); itr.advance( ) )
    System.out.println( itr.retrieve( ) );
```

The initialization prior to the `for` loop is obtained by calling the constructor for `ListItr`. The test uses the `isInList` method defined for the `ListItr` class. The `advance` method advances to the next node in the linked list. We can access the item that is "current" by making a call to the `retrieve` method defined for `ListItr`. The general principle is that since all access to the list is through the `ListItr` class, we have guarantees of safety. Also, we can have multiple iterators traversing a single list.