DEPARTMENT OF COMPUTER SCIENCE
Graham Roberts

An Example Linked List Class

This example shows how a linked linked class can be implemented using a chain of list element objects to represent a list. A basic set of list operations is provided as well as two example iterators. This code is primarily for showing implementation techniques and should not be taken as a definitive list class implementation.

The list supports 'Lisp Style' head and tail methods (car and cdr in Lisp), allowing the value at the head of a list to be returned or a copy of the tail of a list (all values expect the one at the head of the list). It is also possible to insert a value at the head of a list (cons).

The classes and interfaces are:

- interface InsertIterator<E>
- interface InsertIterable<E>
- interface SimpleList<E>
- class LinkedList<E>
- class LinkedListTest (JUnit Test class)

This UML class diagram shows the relationship of the classes and interfaces to each other.

Linked List UML Class Diagram

Note that ListElement, LinkedListIterator and LinkedListInsertIterator are all declared as nested classes inside LinkedList. The cross in a circle notation on the associations represents the nesting relationship. Not shown are inheritance relationships with the Iterator and Iterable interfaces declared within the standard Java class libraries.

The InsertIterator interface extends the standard Iterator interface from the Java Class Libraries.

```
/**
 *  Example interface for iterator that allows insertion.
 *  Copyright (c) 2006
 *  Dept. of Computer Science, University College London
 *  @author Graham Roberts
 *  @version 1.0 01-Mar-06
 */

import java.util.*;

public interface InsertIterator<E> extends Iterator<E>
{
  /**
   * Insert a new value following the current value.
   * @param value  value to insert in list.
   */
  void insert(E value);
}
```

The InsertIterable interface extends the standard Iterable interface from the Java Class Libraries.

```
/**
```

```
 * An extension of the Iterable interface that adds a method
 * to return an iterator that allows values to be inserted
 * to the underlying container during iteration.
 * Copyright (c) 2006
 * Dept. of Computer Science, University College London
 * @author Graham Roberts
 * @version 1.0 01-Mar-06
 */

public interface InsertIterable<E> extends Iterable<E>
{
  /**
   * Calling this method asks for an iterator that can
   * be used to insert a value at the current position
   * during an iteration.
   * @return an insert iterator object.
   */
  InsertIterator<E> insertIterator();
}
```

The SimpleList interface implemented by the LinkedList class. This extends InsertIterable, which declares a method to return an iterator that allows values to be inserted during an iteration.

```
/**
 * Example list data structure interface.
 * Copyright (c) 2006 Dept. of Computer Science, University College London
 *
 * @author Graham Roberts
 * @version 1.0 01-Mar-06
 */

public interface SimpleList<E>
  extends InsertIterable<E>
{
  /**
   * Insert a new value at the head of the list.
   *
   * @param val data object reference to insert.
   */
  void insertHead(E val);

  /**
   * Return value at head of list. The list is unchanged.
   *
   * @return reference to object at head of list or null if the
   * list is empty.
   */
  E getHead();

  /**
   * Return tail of list - the list that is the shallow copy of the current list
   * minus the head element.
   *
   * @return a new list that is a shallow copy of the current list minus
   * the head element, or an empty list if there is no tail.
   */
  SimpleList<E> getTail();

  /**
   * Test to see if list is empty.
   *
   * @return true if list is empty, false otherwise.
   */
  boolean isEmpty();
}
```

Class LinkedList implementing the SimpleList interface.

```
/**
 *  Simple generic Linked List class to demonstrate the basic
 *  principles of implementing a linked list. This should not be taken as a
```

```
 *  production quality class.
 *  Copyright (c) 2006
 *  Dept. of Computer Science, University College London
 *  @author Graham Roberts
 *  @version 2.0 01-Mar-06
 */

import java.util.*;

public class LinkedList<E> implements SimpleList<E>
{
  /**
   * A list is a chain of ListElement objects. head references
   * the first object in the chain or is null if the list is
   * empty.
   */
  private ListElement<E> head;

  /**
   *  Helper class used to implement the list chain. As this is a private helper
   *  class it is acceptable to have public instance variables. Instances of
   *  this class are never made available to client code of the list.
   *  Note that this a generic class that declares a type variable T rather
   *  than E, to avoid confusion with the E declared by LinkedList<E>.
   *  Also, this is a static class, so cannot access E directly and must be
   *  a generic class to enable the strong type checking.
   */
  private static class ListElement<T>
  {
    /**
     *  Next node in chain reference.
     */

    public ListElement<T> next;
    /**
     *  Data object reference.
     */
    public T val;

    /**
     *  Constructor for ListElement.
     *
     *  @param  next  next node in chain reference or null
     *  @param  val   data object reference
     */
    public ListElement(ListElement<T> next, T val)
    {
      this.next = next;
      this.val = val;
    }

    /**
     *  Recursive helper method to copy the chain of ListElements, including
     *  this element.
     *
     *  @return    reference to copied chain of elements.
     */
    public ListElement<T> copy()
    {
      return new ListElement<T>(next == null ? null : next.copy(), val);
    }
  }

  /**
   *  Constructor for LinkedList. By default a list is empty, marked by head
   *  being null.
   */
  public LinkedList()
  {
    head = null;
  }

  /**
   *  Private helper constructor for LinkedList to quickly construct
   *  a new list given a chain of elements.
   *
```

```
  *  @param  e  reference to the element chain forming the list to be held by
  *  the new list object. The chain is not copied.
  */
 private LinkedList(ListElement<E> e)
 {
   head = e;
 }

 /**
  *  Insert a new value at the head of the list.
  *
  *  @param  val  data object reference to insert.
  */
 public void insertHead(E val)
 {
   head = new ListElement<E>(head, val);
 }

 /**
  *  Return value at head of list. The list is unchanged.
  *
  *  @return    reference to object at head of list or null if the
  *  list is empty.
  */
 public E getHead()
 {
   if (head == null)
   {
     return null;
   }
   else
   {
     return head.val;
   }
 }

 /**
  *  Return tail of list - the list that is the copy of the current
  *  list minus the head element. The list elements are copied but not the
  *  objects held in the list (shallow copy).
  *
  *  @return    a new list that is a copy of the current list minus the head
  *  element, or an empty list if there is no tail.
  */
 public SimpleList<E> getTail()
 {
   if ((head == null) || (head.next == null))
   {
     return new LinkedList<E>();
   }
   return new LinkedList<E>(head.next.copy());
 }

 /**
  *  Test to see if list is empty.
  *
  *  @return    true if list is empty, false otherwise.
  */
 public boolean isEmpty()
 {
   return head == null;
 }

 /**
  *  Iterator class to allow each list element to be
  *  visited in sequence. The iterator class is nested in the list class
  *  and is non-static meaning it has access to the state of the
  *  list object being iterated.
  *  This class implements the standard generic Iterator interface but is
  *  not a generic class itself. The type variable E declared by the enclosing
  *  list class can be used in this class as it is a member class not a static
  *  class.
  */
 private class LinkedListIterator implements Iterator<E>
 {
   /**
```

```
    * Instance variable used to store the current position of the iteration.
    * This class uses the technique of creating a dummy list element added to
    * the head of the list chain. This makes handling an empty list easier and
    * also allows the condition current == null to mean that the end of the
    * iteration has been reached.
    */
   protected ListElement<E> current = new ListElement<E>(head,null);

   /**
    *  Determine if there is another element in the sequence, i.e.,
    *  another value in the list.
    *
    *  @return    true if another element in the sequence is available, false
    * otherwise.
    */
   public boolean hasNext()
   {
     return (current != null) && (current.next) != null;
   }

   /**
    *  Return the object reference of the next value in the list. The position
    *  is moved forward before the value is returned.
    *
    *  @return  the next object reference in the sequence or null if at the end
    *  of the sequence.
    */
   public E next()
   {
     if (hasNext())
     {
       current = current.next;
       return current.val;
     }
     return null;
   }

   /**
    * The remove operation is not supported by this iterator. This illustrates
    * that a method required by an implemented interface can be written to not
    * support the operation but should throw an exception if called.
    * UnsupportedOperationException is a subclass of RuntimeException and is
    * not required to be caught at runtime, so the remove method does not
    * have a throws declaration. Calling methods do not have to use a try/catch
    * block pair.
    *
    * @throws UnsupportedOperationException if method is called.
    */
   public void remove()
   {
     throw new UnsupportedOperationException();
   }
}

/**
 *  Return a new Iterator object for the current list.
 *
 *  @return new iterator reference.
 */
public Iterator<E> iterator()
{
  return new LinkedListIterator();
}

/**
 * This iterator class provides an iterator that can insert a value
 * in a list following the current item. This class also illustrates several
 * Java features:
 * - it is both a subclass and implements an interface.
 * - it is a subclass of another member class in the enclosing list class.
 * - InsertIterator extends the Iterator interface to declare the additional
 * insert method.
 */
private class LinkedListInsertIterator
  extends LinkedListIterator
  implements InsertIterator<E>
```

```
  {
    /**
     * Insert a new value following the current value.
     * @param value  value to insert in list.
     */
    public void insert(E value)
    {
      if (head == null)
      {
        insertHead(value);
        current = new ListElement<E>(head,null);
        return;
      }
      if (current != null)
      {
        current.next = new ListElement<E>(current.next,value);
      }
    }
  }

  /**
   *  Return a new InsertIterator object for the current list, allowing
   *  items to be inserted into the list.
   *
   *  @return new insert iterator reference.
   */
  public InsertIterator<E> insertIterator()
  {
    return new LinkedListInsertIterator();
  }
}
```

The JUnit test class for the LinkedList class.

```
/*
 * JUnit test class for LinkedList class, including its iterators.
 * Copyright (c) 2006
 * Dept. of Computer Science, University College London
 * @author Graham Roberts
 * @version 2.0 01-Mar-06
 */

import junit.framework.* ;

import java.util.*;

public class LinkedListTest extends TestCase
{
  private SimpleList<Integer> empty ;
  private SimpleList<Integer> one ;
  private SimpleList<Integer> several ;

  public void setUp()
  {
    empty = new LinkedList<Integer>() ;
    one = new LinkedList<Integer>() ;
    one.insertHead(0) ;
    several = new LinkedList<Integer>() ;
    several.insertHead(2) ;
    several.insertHead(1) ;
    several.insertHead(0) ;
  }

  public void testGetHead()
  {
    assertNull(empty.getHead()) ;
    assertEquals(new Integer(0),one.getHead()) ;
    assertEquals(new Integer(0),several.getHead()) ;
    assertEquals(new Integer(1),several.getTail().getHead()) ;
    assertEquals(new Integer(2),several.getTail().getTail().getHead()) ;
  }

  public void testEmpty()
  {
```

```
    assertTrue(empty.isEmpty());
    assertFalse(one.isEmpty());
    assertTrue(one.getTail().isEmpty()) ;
    assertFalse(several.isEmpty());
    assertFalse(several.getTail().isEmpty()) ;
    assertFalse(several.getTail().getTail().isEmpty()) ;
    assertTrue(several.getTail().getTail().getTail().isEmpty()) ;
  }

  public void testIterator()
  {
    for (Iterator<Integer> iterator = empty.iterator() ; iterator.hasNext(); )
    {
      fail("Iterating empty list and found element") ;
    }
    int counter = 0 ;
    for (Iterator<Integer> iterator = one.iterator() ; iterator.hasNext(); )
    {
      assertEquals(new Integer(counter++),iterator.next()) ;
    }
    assertEquals(1,counter);
    counter = 0 ;
    for (Iterator<Integer> iterator = several.iterator() ; iterator.hasNext(); )
    {
      assertEquals(new Integer(counter++),iterator.next()) ;
    }
    assertEquals(3,counter);
  }

  public void testIteratorViaEnhancedForLoop()
  {
    int counter = 0 ;
    for (int i : empty)
    {
      fail("Iterating empty list and found element") ;
    }
    counter = 0 ;
    for (int i : one)
    {
      assertEquals(counter++,i) ;
    }
    assertEquals(1,counter);
    counter = 0 ;
    for (int i : several)
    {
      assertEquals(counter++,i) ;
    }
    assertEquals(3,counter);
  }

  public void testInsertEmpty()
  {
    InsertIterator<Integer> iterator = empty.insertIterator();
    assertFalse(iterator.hasNext());
    iterator.insert(2);
    assertTrue(iterator.hasNext());
    assertEquals(new Integer(2),iterator.next());
  }

  public void testInsert()
  {
    InsertIterator<Integer> iterator = one.insertIterator();
    assertTrue(iterator.hasNext());
    iterator.next();
    assertFalse(iterator.hasNext());
    iterator.insert(2);
    assertTrue(iterator.hasNext());
    assertEquals(new Integer(2),iterator.next());
    assertFalse(iterator.hasNext());
  }

  public void testInsertAfterEnd()
  {
    InsertIterator<Integer> iterator = one.insertIterator();
    iterator.next();
    iterator.next();
```

```
    assertFalse(iterator.hasNext());
  }
}
```

Last updated: September 2, 2006