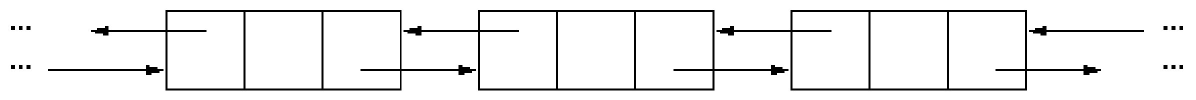
4.1.2.3 Implementing Doubly-Linked Lists

Although the Java programming language has not many composite data structures built-in, they can easily be implemented. In this section we shall discuss an implementation of *doubly-linked lists*. This example of a self-made container class also shows how to implement an enumeration interface for accessing the elements sequentially. If you wish, you may skip this subsection or come back to it later.

Overall Structure of Doubly-Linked Lists

If you wish to traverse a list both forwards and backwards efficiently, or if you wish, given a list element, to determine the preceding and following elements quickly, then the *doubly-linked list* comes in handy. A list element contains the data plus pointers to the next and previous list items as shown in the picture below.

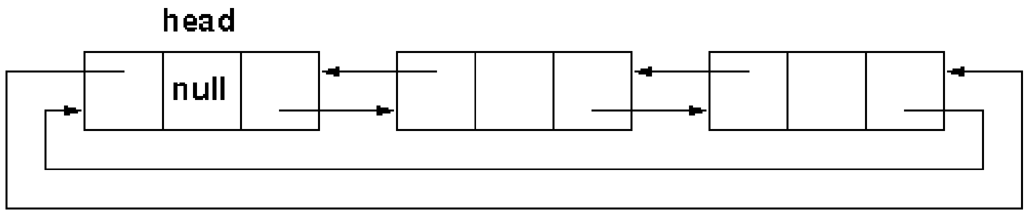
A Doubly-Linked List



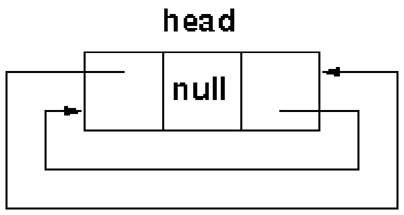
Of course we need a pointer to some *link* in the doubly-linked list to access list elements. It is convenient for doubly-linked lists to use a *list header*, or *head*, that has the same structure as any other list item and is actually part of the list data structure.

The picture below shows an empty and nonempty doubly–linked list. By following arrows it is easy to go from the list header to the first and last list element, respectively.

A doubly–linked list with 2 elements

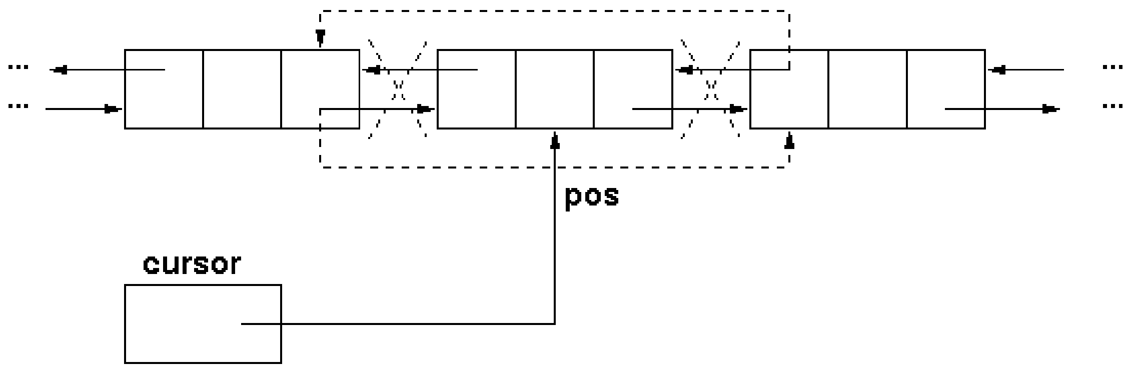


A doubly–linked list with no elements



Insertion and removal of an element in a doubly–linked list is in this implementation rather easy. In the picture below we illustrate the pointer changes for a removal of a list item (old pointers have been drawn solid and new pointers are dashed arrows). We first locate the previous list item using the *previous* field. We make the *next* field of this list item point to the item following the one in cursor position *pos*. Then we make the *previous* field of this following item point to the item preceding the one in the cursor position *pos*. The list item pointed to by the cursor becomes useless and should be automatically garbage collected.

Removal of an element of a doubly–linked list



In a pseudo-programming language we could write the following code:

```
    remove(cursor pos)
      begin
        if empty list  then
          ERROR("empty list.")
        else if cursor points at list header  then
          ERROR("cannot remove the list header")
        else

          pos ↑ previous ↑ next = pos ↑ next;


          pos ↑ next ↑ previous = pos ↑ previous;

        endif
      end
```

Implementing the List Items

The basic `ListItem` class is easily defined as follows

ListItem.java

```
  final class ListItem {
    Object obj;
    ListItem previous, next;

    public ListItem(Object obj) {
      this(null, obj, null);
    }

    public ListItem(ListItem previous, Object obj, ListItem next) {
      this.previous = previous;
      this.obj = obj;
      this.next = next;
    }
  }
```

A class definition with only two constructor methods. The keyword `final` ensures that this class has no subclasses nor that a user can derive a class from this one.

Implementing the Doubly-Linked List

The objects of type `ListItem` are placed in a doubly-linked list. For this we have to define a class `LinkedList`. It contains a special `ListItem`, called the `head`, to give the user a principal handle to the list items. The insertion methods (`insertBefore`, `insertAfter`) and the removal method (`remove`) in the `LinkedList` class need a pointer to the position in the list at which action should take place.

For this purpose we introduce a `ListIterator` class. A `ListIterator` contains the `LinkedList` to which it belongs and a pointer to the `ListItem` that is currently selected. The `ListIterator` class contains methods to move the cursor position. The `LinkedList` class contains the methods

- `Head`, which returns an iterator that points to the head of the linked list;
- `find`, which searches in the linked list for a requested object and, if found, returns an iterator pointing to the object

To step through all current list items, we added a procedure `elements` in the `LinkedList` class; it returns an `Enumeration` object.

Most of the Java code for the two classes below should speak for itself.

ListIterator.java

```
final class ListIterator {
  LinkedList owner;
  ListItem pos;

  ListIterator(LinkedList owner, ListItem pos){
    this.owner = owner;
    this.pos = pos;
  }

 /*
  * check whether object owns the iterator
  */
  public boolean belongsTo(Object owner) {
    return this.owner == owner;
  }

 /*
  * move to head position
  */
  public void head() {
    pos = owner.head;
  }

 /*
  * move to next position
  */
  public void next() {
    pos = pos.next;
  }

 /*
  * move to previous position
  */
  public void previous() {
    pos = pos.previous;
  }
}
```

LinkedList.java

```
import java.util.*;

public class LinkedList {
  ListItem head;
 /*
  * creates an empty list
  */
  public LinkedList() {
    head = new ListItem(null);
    head.next = head.previous = head;
  }

 /*
  * remove all elements in the list
  */
  public final synchronized void clear() {
    head.next = head.previous = head;
  }

 /*
  * returns true if this container is empty.
  */
  public final boolean isEmpty() {
    return head.next == head;
  }

 /*
  * insert element after current position
  */
  public final synchronized void insertAfter(Object obj, ListIterator cursor) {
    ListItem newItem = new ListItem(cursor.pos, obj, cursor.pos.next);
    newItem.next.previous = newItem;
    cursor.pos.next = newItem;
  }

 /*
  * insert element before current position
  */
  public final synchronized void insertBefore(Object obj, ListIterator cursor) {
    ListItem newItem = new ListItem(cursor.pos.previous, obj, cursor.pos);
    newItem.previous.next = newItem;
    cursor.pos.previous = newItem;
  }

 /*
  * remove the element at current position
  */
  public final synchronized void remove(ListIterator cursor) {
    if (isEmpty()) {
      throw new IndexOutOfBoundsException("empty list.");
    }
    if (cursor.pos == head) {
      throw new NoSuchElementException("cannot remove the head");
    }
    cursor.pos.previous.next = cursor.pos.next;
    cursor.pos.next.previous = cursor.pos.previous;
  }

 /*
  * Return an iterator positioned at the head.
  */
```

```
        public final ListIterator head() {
          return new ListIterator(this, head);
        }

       /*
        * find the first occurrence of the object in a list
        */
        public final synchronized ListIterator find(Object obj) {
          if (isEmpty()) {
            throw new IndexOutOfBoundsException("empty list.");
          }
          ListItem pos = head;
          while (pos.next != head) {  // There are still elements to be inspected
            pos = pos.next;
            if (pos.obj == obj) {
              return new ListIterator(this, pos);
            }
          }
          throw new NoSuchElementException("no such object found");
        }

       /*
        * Returns an enumeration of the elements. Use the Enumeration methods on
        * the returned object to fetch the elements sequentially.
        */
        public final synchronized Enumeration elements() {
          return new ListEnumerator(this);
        }
      }
```

The `synchronized` keyword in many of the above methods indicates that the methods that have this modifier change the internal state of a `LinkedList` object which is not "thread-safe": for example, insertions and removals of list elements should not be carried out in random order, but in the order in which they are requested. The `synchronized` keyword takes care of this: when you call a `synchronized` instance method, Java first locks the instance so that no other threads can modify the object concurrently. See the section on interaction between threads for more details on synchronization.

Providing the List Enumeration

Java provides the `Enumeration` interface to step through the elements in an instance of a built-in container class such as `Vector`. You can look up the source code of the `Enumeration` interface in the development kit; it looks as follows:

```
public interface Enumeration {

    boolean hasMoreElements();

    Object nextElement();
}
```

So, these are the two methods that you can use to iterate through the set of values. Two examples of their use.

- Printing the class names of all applets in a Web page:

```
Enumeration e = getAppletContext().getApplets();  // get all applets
while (e.hasMoreElements()) {            // step through all applets
  Object applet = e.nextElement();
  System.out.println(applet.getClass().getName());
}
```

- Working with vectors:

```
Vector vector = new Vector();       // declaration of vector
vector.addElement(new ...);         // with addElement you can add items
Enumeration e = vector.elements();  // get all vector elements
while (e.hasMoreElements()) {       // step through all vector elements
  Object obj = e.nextElement();
  // work with the object .....
}
```

Of course we want to offer this enumeration facility for doubly–linked lists as well. The `elements` method of the `LinkedList` class already delivers an `Enumeration`, actually an instance of the `ListEnumerator` class.

The skeleton of this class is as follows:

```
import java.util.*;

final class ListEnumerator implements Enumeration {
  LinkedList list;
  ListIterator cursor;

  ListEnumerator(LinkedList l) {
  }

  public boolean hasMoreElements() {
  }

  public Object nextElement() {
  }
}
```

The enumerator contains

- a `LinkedList` instance variable that points to the doubly–linked list we want to enumerate;
- a `ListItem` instance variable that serves as a cursor to the doubly–linked list under consideration.

You may wonder why another cursor to step through the data structure, but when you traverse a linked list, maybe a cursor in use is at an important position before traversal and should stay there after traversal.

The constructor `ListEnumerator(LinkedList l)` generates the enumeration from a given doubly–linked list: it positions the cursor at the list header and move it one position further. In Java code:

```
ListEnumerator(LinkedList l) {
  list = l;
  cursor = list.head();
  cursor.next();
}
```

What remains to be done is the implementation of the two promised methods:

- `hasMoreElements`: simply check whether the cursor presently points at the list header.

  ```
  public boolean hasMoreElements() {
    return cursor.pos != list.head;
  }
  ```

- `nextElement`: return the data of the list item presently pointed at and move the cursor one step further. All this should happen unless the cursor is already pointing at the list header, in which case an exception is thrown. (See the section on exceptions for details about this topic)

  ```
  public Object nextElement() {
    synchronized (list) {
      if (cursor.pos != list.head) {
        Object object = cursor.pos.obj;
        cursor.next();
        return object;
      }
    }
    throw new NoSuchElementException("ListEnumerator");
  }
  ```

  Here you see another use of the `synchronized` keyword:

  `synchronized (`*`expression`*`)` *`statement`*,

  where *`expression`* is an object or array which is locked during execution of the *`statement`*. This ensures that no other threads can be executing the program section at the same time.

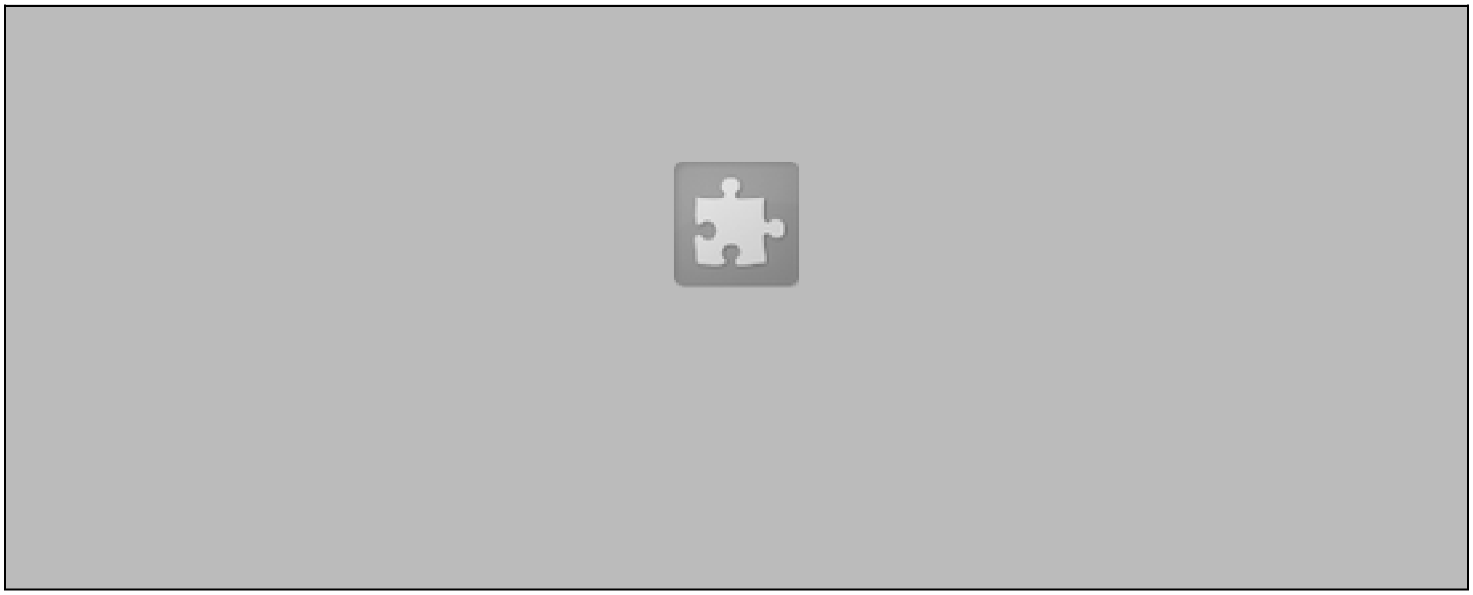
Collecting the Java code defining the `ListEnumerator` class we get:

ListEnumerator.java

```
import java.util.*;

final class ListEnumerator implements Enumeration {
  LinkedList list;
  ListIterator cursor;

  ListEnumerator(LinkedList l) {
    list = l;
    cursor = list.head();
    cursor.next();
  }

  public boolean hasMoreElements() {
    return cursor.pos != list.head;
  }

  public Object nextElement() {
    synchronized (list) {
      if (cursor.pos != list.head) {
        Object object = cursor.pos.obj;
        cursor.next();
        return object;
      }
    }
    throw new NoSuchElementException("ListEnumerator");
  }
}
```

Demonstrating the Doubly-Linked List Implementation

To show how the doubly-linked lists really work and to test the correctness of the implementation, it is advisable to write a graphical program that allows you to do this. Below, you see a Java applet that simulates a doubly-linked list of integers. You start with an empty list and the first four buttons allow you to insert or remove elements. There is also a cursor belonging to the list. It is always positioned at the list item that is drawn in red. The `previous` and `next` buttons allow you to move the cursor. At the right-hand side we have a button to step through all list elements and to find an object in the list.



The source code for this applet is available and can be used for testing other data structures such as queues, stacks, singly–linked (circular) lists, and so on.