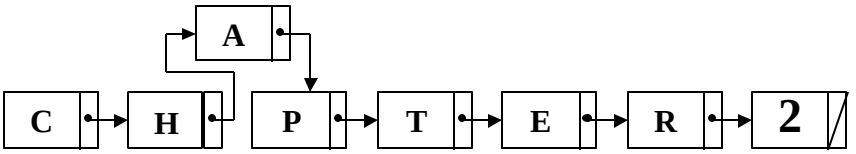

Lists and Iterators

2.1 Prologue

This chapter opens our discussion of a series of data structures that constitute a standard set of tools in software design and development. These include lists, stacks, queues, trees, priority queues, hash tables, and other structures.

A data structure combines data organization with methods of accessing and manipulating the data.

For example, an array becomes a data structure for storing a list of values when we provide methods to find, insert, and remove a value. Similar functionality can be achieved with a ***linked list*** structure, which we will explain shortly. At a very abstract level, we can think of a general "list" object: a list contains a number of values arranged in sequence; we can find a target value in a list, add values to the list, and remove values from the list.

The "List" can be further specialized. We can require, for example, that the values in the list be arranged in ascending (e.g., alphabetic) order and stipulate that the insert function put the new value into the appropriate place in the order. This can be called the "Ordered List."

The data structures that we are going to study are not specific to Java — they can be implemented in any programming language. In Java, a data structure can be described as an interface and implemented as a class or several classes. For example, the Java library interface `java.util.List` describes the "List" data structure. Two Java library classes, `ArrayList` and `LinkedList`, implement this interface.

In this chapter we will discuss the following topics:

- Some of the methods of the `List` interface
- The `ArrayList` and `LinkedList` library implementations of the `List` interface; their advantages and disadvantages in different situations
- Traversing a list using `Iterator` or `ListIterator` objects

We will also learn the basics of implementing our own linked list. Finally we will use the Java library class `LinkedList` in a case study that creates an actor index for a database of movies.

2.2 The `List` Interface

The `List` interface from the `java.util` package gives a formal description of a list. The Java library also provides two classes, `ArrayList` and `LinkedList` that implement the `List` interface. When you use the `List` interface or `ArrayList` or `LinkedList` classes, add

```
import java.util.*;
```

(or

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;
```

as needed) to your code.

Note: the `java.util.List` interface is not to be confused with the `java.awt.List` class, which represents a GUI component (a list of items displayed on the screen).

The `List` interface stipulates that the elements of a list are objects (i.e., have the `Object` data type). For example:

```
void add(Object obj);  // append obj to the list
Object get(int i);     // return the value of the i-th element
```

Since any class extends `Object`, you can put any kind of object into a list, including arrays and `String`s. The methods that retrieve values from the list, such as `get`, return an `Object`; you have to cast the object back into its type when you retrieve it from the list. For example:

```
List bandNames = new ArrayList();
bandNames.add("ABBA");
...
int i = ...;
String name = (String)bandNames.get(i);
```

With primitive data types (`int`, `double`, `boolean`, etc.), the situation is slightly more complicated because in Java they are not objects. If you want to have a list of `int`s or `double`s, you have to convert them into objects using the `Integer` or `Double` ***wrapper*** class (*Java Methods*, p. 268). For example:

```
List list = new ArrayList();
double x = ...;
int m = ...;

list.add(new Double(x));
list.add(new Integer(m));
...

int i = ...;

x = ((Double)list.get(i)).doubleValue();
m = ((Integer)list.get(i+1)).intValue();
...
```

The `List` interface has methods for accessing elements through their integer index (position in the list). The elements are numbered starting from 0, as in Java strings and arrays. There are also methods for searching for a given value in the list. The Java API for the `List` interface specifies over two dozen methods; however, some of them are marked as optional and don't have to be present in every implementation.

```
int size();                     // Returns the number of values
                                //   currently stored in the list
boolean isEmpty();              // Returns true if the list is empty,
                                //   otherwise returns false
boolean add(Object obj);        // Appends obj at the end of the list;
                                //   returns true
void add(int i, Object obj);    // Inserts obj before the i-th element;
                                //   increments the indices of the
                                //   subsequent elements by 1
Object set(int i, Object obj); // Replaces the i-th element
                                //   with obj; returns the old value
Object get(int i);              // Returns the value of the i-th
                                //   element
Object remove(int i);           // Removes the i-th element from the
                                //   list and returns its old value;
                                //   decrements the indices of the
                                //   subsequent elements by 1
```

Figure 2-1. Commonly used `List` methods

Like other interfaces, `List` does not tell us anything about constructors. It is up to a class that implements `List` to supply suitable constructors that initialize an empty list, build a list from given items, read a list from a file, and so on.

A few commonly used `List` methods are summarized in Figure 2-1. The `size()` method returns the total number of values currently stored in the list. The `boolean` method `isEmpty()` returns `true` if the list is empty, `false` otherwise. The `add(Object obj)` method appends a given value at the end of the list; the overloaded version with two arguments, `add(int i, Object obj)`, inserts the new value at the given position `i` and increments the indices of the subsequent elements by one. The `get(int i)` method returns the value of the *i*-th element in the list. The `set(int i, Object obj)` replaces the value of the *i*-th element with a new value and returns the old value. The `remove(int i)` method removes the *i*-th element and decrements the indices of all the subsequent elements by one. The `add`, `get`, `set`, and `remove` methods check that the given index `i` is within the legal range, $0 \le$ `i` $<$ `size()`. If the index is out of range, these methods cause a run-time error, `IndexOutOfBoundsException`.

2.3 Array Implementation of a List

A one-dimensional Java array already provides most of the functionality of a list. When we want to use an array as a list, we create an array that can hold a certain maximum number of values; we then keep track of the actual number of values stored in the array. The array's length becomes its maximum capacity and the number of values currently stored in the array is the size of the list.

However, Java arrays are not resizable. If we want to be able to add values to the list and not worry about exceeding its maximum capacity, we have to use a class with an `add` method, which allocates a bigger array and copies the list values into the new array when the list runs out of space. That's what the `ArrayList` library class does.

Before looking at the library class, let's make our own sketch for an encapsulated class that implements a list as an array (Figure 2-2). Our class defines a field of the `Object[]` type (array of objects) that holds the list's values. Another field keeps track of the current list size (the number of values stored in the array). The no-args constructor creates an array of some default capacity and makes the list empty (sets the count of stored values to 0). Another constructor creates an array of a given capacity, passed to it as a parameter.

Some methods of the Java `List` interface, such as `get` and `set`, translate directly into array operations. Other methods, such as `add` and `remove`, require extra work.

```
public class MyArrayList
{
  private static final int DEFAULT_CAPACITY = 16;
  private Object myValues[];
  private int myNumValues;

  public MyArrayList()
  {
    myValues = new Object[DEFAULT_CAPACITY];
    myNumValues = 0;    // optional -- default
  }

  public MyArrayList(int capacity)
  {
    myValues = new Object[capacity];
    myNumValues = 0;    // optional -- default
  }

  public int size()
  {
    return myNumValues;
  }

  public boolean isEmpty()
  {
    return myNumValues == 0;
  }

  public Object get(int i)
  {
    if (i < 0 || i >= myNumValues)
      throw new IndexOutOfBoundsException();
                // report an error by creating and "throwing"
                //   an IndexOutOfBoundsException object
    return myValues[i];
  }

  public void add(Object obj)
  {
    if (myNumValues == myValues.length)
    {
      Object temp[] = new Object[2 * myValues.length];
      for (int i = 0; i < myNumValues; i++)
        temp[i] = myValues[i];
      myValues = temp;
    }
    myValues[myNumValues] = obj;
    myNumValues++;
  }
}
```

Figure 2-2. A sketch for implementing a list class using an array

In `add` you need to expand the array's capacity when the list runs out of space. We have chosen in that case to double the current capacity: we allocate a new array twice the length of the old one and then copy all the values into the new array. You have to use such an `add` method with caution because copying large arrays may slow down your program. The `remove` method (not shown) may require shifting the remaining elements towards the beginning of the array.

The Java library class `java.util.ArrayList`, which implements the `List` interface, is much more involved, but the basic approach is similar to the sketch above.

Note that in our `MyArrayList` class, as well as in the `ArrayList` library class, the `add` method does not create a copy of the object when it adds it to the list; `add` adds to the list a reference to the original object.

Therefore, you can modify an object after it has been added to a list; when you extract that object from the list, you will get the modified value.

2.4 Linked Lists

The elements of an array are stored in consecutive locations in computer memory. We can calculate the address of each element from its index in the array. By contrast, the elements of a linked list may be scattered in various locations in memory, but each element contains the memory address of the next element in the list. In Java, referring to addresses of objects is easy because all objects are represented as references, which are essentially their memory addresses. The last element in the list points to nothing, so its reference to the next element is set to `null`.

Metaphorically, we can compare an array to a book: we can read its pages sequentially or we can open it to any page. A linked list is like a magazine article: at the end of the first installment it says, "continued on page 27." We read the second installment on page 27, and at the end it says, "continued on page 36," and so on, until we finally reach the ♦ symbol that marks the end of the article.

We will refer to the elements of a linked list as "nodes." A node contains some information useful for a specific application and a reference to the next node.

Let us say that the information stored in a node is represented by an object called `value` and that the reference to the next node is called `next`. We can encapsulate these fields in a class `ListNode` with one constructor, two accessors, and two modifiers (Figure 2-3).*

```
public class ListNode
{
  private Object value;
  private ListNode next;

  // Constructor:
  public ListNode(Object initValue, ListNode initNext)
  {
    value = initValue;
    next = initNext;
  }

  public Object getValue()
  {
    return value;
  }

  public ListNode getNext()
  {
    return next;
  }

  public void setValue(Object theNewValue)
  {
    value = theNewValue;
  }

  public void setNext(ListNode theNewNext)
  {
    next = theNewNext;
  }
}
```

Figure 2-3. A class that represents a node in a linked list (`Ch02\ListNode.java`)

* Adapted from The College Board's *AP Computer Science AB: Implementation Classes and Interfaces*.

Note two things about `ListNode`'s definition. First, "next" is a name chosen by the programmer: it is not required by Java syntax. We could have called it "link" or "nextnode" or whatever name we wanted. The name of the class, "ListNode," is also chosen by the programmer.

Second, the definition is self-referential: it refers to the `ListNode` data type inside the `ListNode` data type definition! The compiler is able to untangle this because `next` is a reference to an object and represents an address. An address takes a fixed number of bytes regardless of the data type, so the compiler can calculate the total size of a `ListNode` without paying much attention to what type of reference `next` is.

As an example, let us consider a list of departing flights on an airport display. The flight information may be represented by an object of the type `Flight`:

```
public class Flight
{
  private int number;           // Flight number
  private String destination;   // Destination city
  ...                           // Other fields and methods
}
```

Suppose a program has to maintain a list of flights departing in the next few hours, and we have decided to implement it as a linked list. We can use the following statements to create a new node that holds information about a given flight:

```
Flight flt = new Flight(...);
...
ListNode node = new ListNode(flt, null);
```

To extract the flight info we need to cast the object returned by `getValue` back into the `Flight` type:

```
flt = (Flight)node.getValue();
```

❖ ❖ ❖

A linked list is accessed through a reference to its first node. When a program creates a linked list, it usually starts with an empty list — the `head` reference is `null`:

```
ListNode head;        // head is set to null by default
```

The first node can be created using the `ListNode` constructor and appended to the list:

```
head = new ListNode(value0, null);
```

This results in a list with one node (Figure 2-4).

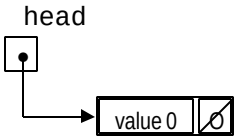

Figure 2-4. A linked list with only one node

A second node may be appended to the first:

```
ListNode node1 = new ListNode(value1, null);
head.setNext(node1);
```

Or, combining the above two statements into one:

```
head.setNext(new ListNode(value1, null));
```

This statement changes the `next` field in the `head` node from `null` to a reference to the second node. This is illustrated in Figure 2-5. Diagrams like this one help us understand how links in a linked list are reassigned in various operations.

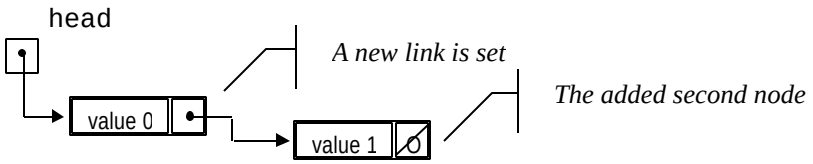

Figure 2-5. The second node is appended to the list

When we build a list, we can keep track of the last node and append a new node to it:

```
ListNode head = null, tail = null, node;
while (...) // more values to insert
{
  Object value = ... ; // get the next value
  node = new ListNode(value, null);
  if (head == null)
    head = node;
  else
    tail.next = node;
  tail = node;    // update tail
}
```

Figure 2-6 shows the resulting linked list.

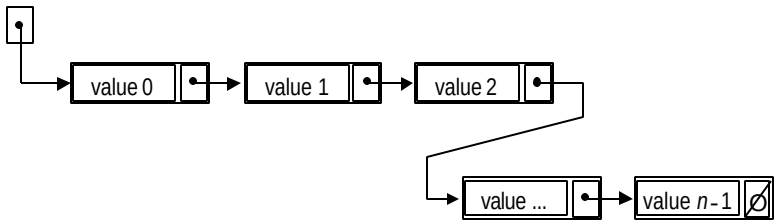

Figure 2-6. A linked list

Alternatively, we can easily attach a new node at the head of the list. For example:

```
ListNode node = new ListNode(value, null);
node.setNext(head);
head = node;
```

Or simply:

```
head = new ListNode(value, head);
```

The above statement creates a new node and sets its `next` field equal to the current head of the list. It then sets `head` to refer to the newly created node (Figure 2-7).

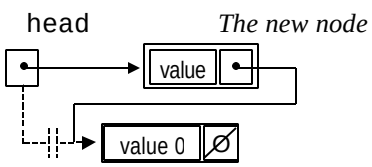

Figure 2-7. A new node inserted at the head of a linked list

2.5 Linked Lists vs. Arrays

As we have seen, a list can be implemented as either an array or a linked list — this is what the Java library classes `ArrayList` and `LinkedList` do. Since these two classes implement the same interface and provide the same methods, a novice might think that they are completely interchangeable. Indeed, either class is likely to work equally well on small lists. However, each of these implementations has its strengths and limitations, and choosing the wrong one may result in dramatic deterioration of performance on large lists.

The array implementation provides direct access to the *i*-th element of the array. This property, called ***random access***, is important in many algorithms. For example, the Binary Search algorithm requires access to the element directly in the middle between two elements. This is fast with arrays but slow with linked lists.

Arrays have two drawbacks, however. First, it is not easy to insert or remove a value at the beginning or in the middle of an array — a lot of bytes may need to be moved if the array is large. Second, we have to declare an array of a particular size. If the list outgrows the allocated space, we need to allocate a bigger array and copy all the values into it.

Linked lists get around both of these problems. First, a node can be easily inserted or removed from a linked list simply by rearranging the links. This is a crucial property if we have a frequently updated list. Second, the nodes of a linked list are allocated only when new values are added, so no memory is wasted for vacant nodes and a list can grow as large as the total memory permits.

The above considerations directly affect the performance of different methods in `ArrayList` and `LinkedList`. In `ArrayList`, the `get(i)` and `set(i, obj)` method calls are quick and become the primary tools for handling a list. But if we want to get to the *i*-th element in a linked list, we have to start from the head node and proceed down the list, counting the nodes, until we get to the *i*-th node. Thus the `get` and `set` operations may become costly. On the other hand, the `add(0,value)` method call may be quite slow for `ArrayList` because all the values must shift to make room for the added first value. In `LinkedList`, this method call just rearranges the links, so it is fast. If a class that implements linked lists keeps track of the list's last node, then appending new values at the end of the list is fast, too. And we can insert a whole linked list in the middle of another linked list or concatenate two linked lists together simply by rearranging a few links, without moving any data.

To make it simple to use linked lists appropriately, the `LinkedList` class provides a few specialized methods that are particularly efficient for linked lists: `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst`, and `removeLast` (Figure 2-8).

```
void addFirst(Object obj); // Appends obj at the beginning of the list
void addLast(Object obj);  // Appends obj at the end of the list
Object getFirst();         // Returns the first element
Object getLast();          // Returns the last element
Object removeFirst();      // Removes the first element from the
                           //   list and returns its old value
Object removeLast();       // Removes the last element from the
                           //   list and returns its old value
```

Figure 2-8. Efficient methods in `java.util.LinkedList`

On the surface,

```
Object obj = list.getLast();
```

and

```
Object obj = list.get(list.size() - 1);
```

may appear interchangeable. But, depending on the `LinkedList` implementation, the former statement may work faster.

2.6 Traversals and Iterators

A procedure that processes in sequence all the values stored in a data structure is called a ***traversal***. For an array, traversal is accomplished simply by accessing consecutive elements using a position index and incrementing the index on each iteration. For example, suppose an `ArrayList` contains `String`s. We can print them all out within a simple loop:

```
List movies = new ArrayList();
// Add values to the list:
...

// Print all the values in the list:
for (int i = 0; i < movies.size(); i++)
  System.out.println((String)movies.get(i));
```

In the last statement, `println(movies.get(i))`, without the cast to `String`, would also work: `println(Object obj)` uses `println(obj.toString())` and polymorphism takes care of calling the correct `toString` method for different types of objects.

❖ ❖ ❖

For linked lists, traversals that access elements through their indices are inefficient.

A loop

```
for (int i = 0; i < list.size(); i++)
{
  value = list.get(i);
  ...
}
```

does work for a linked list, but, as we discussed in the previous section, it is quite inefficient. If we have access to the head of the list, we can write a more efficient traversal loop for a linked list, which follows the links starting from the head of the list:

```
ListNode head;
// Add values to the list:
...

// Print all the values in the list:

ListNode node;
for (node = head; node != null; node = node.getNext())
  System.out.println((String)node.getValue());
```

This works fine as long as `head` is accessible to us. The problems begin when we implement a linked list as an encapsulated class. How can we deal with traversals? If we make the `head` field public in our class or if we provide a method that returns `head`, the whole list becomes exposed and our attempt at encapsulation fails. If we hide `head` inside our class and try to make traversal a class method, how will this method know what we want to do with each node? Traversals will become too specialized.

This dilemma is resolved with the help of ***iterators***. As the term suggests,

the purpose of an iterator is to iterate through the list and provide access to the consecutive elements.

Iterators work both for array lists and linked lists, but they are much more important for linked lists. If your method's argument is a generic `List` object and the method needs to traverse the list, use an iterator just in case you get a linked list.

An iterator is an object associated with the list. When an iterator is created, it points to a specified element in the list, usually the first element. We call the iterator's methods to check whether there are more elements to be visited left in the list and to obtain the next element. In Java, this concept is expressed in the library interface `java.util.Iterator`. Classes that implement the `Iterator` interface are implemented as private ***inner*** classes in `ArrayList` and `LinkedList` — a topic that is outside the scope of this book. What is important for us is that the list itself provides an iterator when we call its `iterator()` method. For example:

```
LinkedList list = new LinkedList();
// Add values to the list:
...

Iterator iter = list.iterator();
```

An iterator is an object with three methods:

```
Object next();       // Returns the next element in the
                     //   list and advances the iterator's
                     //   "cursor" to the following element
boolean hasNext();   // Returns true if there are more elements
                     //   to visit, false otherwise
void remove();       // Removes the last element returned by next
                     //   from the list
```

Now our list traversal may look as follows:

```
LinkedList list = new LinkedList();
// Add values to the list
...

Iterator iter = list.iterator();
while (iter.hasNext())
  System.out.println(iter.next());
```

The increment for the `while` loop is hidden inside the `next` method. For linked lists, this implementation is much more efficient than the one with `get(i)`.

You may be wondering at all this complexity. Why, for example, couldn't the `LinkedList` class itself have provided something like `startIterations`, `next`, and `hasNext` methods? The answer is, it could, as long as you used simple iterations. But when you tried nested iterations, this approach would fall apart.

Suppose, for example, you want to find duplicate titles in a linked list of movies. For that you have to run nested iterations on the same list, comparing every pair. The idea of a list serving as its own iterator would lead to an error, something like this:

```
String movie1, movie2;

movies.startIterations();
while (movies.hasNext())
{
  movie1 = (String)movies.next();
  movies.startIterations();  // Oops, a bug -- the outer loop's
                             //   position is lost
  while (movies.hasNext())
  {
    movie2 = (String)movies.next();
    if (movie1 != movie2 && movie1.equals(movie2))
      System.out.println("Duplicate name: " + movie1);
  }
}
```

What you need is two separate iterators, one for the outer loop and one for the inner loop:

```
String movie1, movie2;
Iterator iter1, iter2;

iter1 = movies.iterator();
while (iter1.hasNext())
{
  movie1 = (String)iter1.next();

  iter2 = movies.iterator();
  while (iter2.hasNext())
  {
    movie2 = (String)iter2.next();
    if (movie1 != movie2 && movie1.equals(movie2))
      System.out.println("Duplicate name: " + movie1);
  }
}
```

You may still notice a problem with the above code: it is inefficient because we always start iterations from the beginning of the list and test the same pair of movies twice. In addition, the same "duplicate name" message will be displayed twice. The `Iterator` interface turns out to be very limited: an iterator always iterates from the beginning of the list and only in one direction.

Luckily, Java offers a fancier iterator, the `java.util.ListIterator` interface, which overcomes these limitations. `ListIterator` extends `Iterator` (recall that in Java, a "subinterface" can extend a "superinterface").

A `ListIterator` object is returned by `List`'s `listIterator()` method. If you want to start iterations from the *i*-th element, use the overloaded version, `listIterator(i)`, where `i` is the initial position. In addition to the `next`, `hasNext`, and `remove` methods, `ListIterator` supplies the `previous` and `hasPrevious` methods.

If you obtain a `ListIterator` with the `listIterator(i)` call, then the first value returned by `next` will be the value of the element with the index `i`, and the first value returned by `previous` will be the value of the element with the index `i-1`.

It may be useful to envision a list iterator as a logical "cursor" positioned before the list, after the list, or between two consecutive elements of the list (Figure 2-9). `list.listIterator(0)` positions the cursor before the first element of the list (the element with the index 0). `list.listIterator(list.size())` positions the cursor after the end of the list. `list.listIterator(i)` positions the cursor between the elements with indices `i-1` and `i`. The `next` method returns the element immediately after the cursor position, and the `previous` method returns the element immediately before the cursor position.

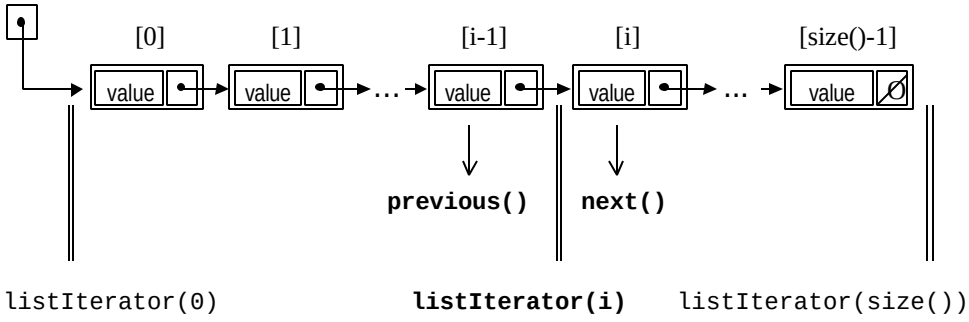

Figure 2-9. Logical "cursor" positioning for a list iterator

To traverse the list from the end backwards, all you have to do is obtain a list iterator that starts past the end of the list, then call `previous`. For example:

```
ListIterator iter = list.listIterator(list.size());
while (iter.hasPrevious())
{
  value = iter.previous();
  ...
}
```

The `ListIterator` interface also provides an `add` method that inserts an element into the list (at the current "cursor" position) and a `set` method that changes the value of an element. The commonly used `ListIterator` methods are summarized in Figure 2-10.

```
Object next();          // Returns the next element in the list
boolean hasNext();      // Returns true if the next element in the
                        //   list is available and false otherwise
Object previous();      // Returns the previous element in the list
boolean hasPrevious(); // Returns true if the previous element in the
                        //   list is available and false otherwise
void add(Object obj);  // Inserts a new element obj at the current
                        //   iterator position
void set(Object obj);  // Sets the value of the last element that was
                       //   returned by next or previous to obj
void remove();          // Removes from the list the last element that
                        //   was returned by next or previous
int nextIndex();        // Returns the index of the element that
                        //   will be retrieved by next
int previousIndex();    // Returns the index of the element that
                        //   will be retrieved by previous
```

Figure 2-10. Commonly used `java.util.ListIterator` methods

Now, armed with `ListIterator`s, we can make our nested loop for finding duplicates in a list more efficient, comparing each value only with the subsequent values in the list:

```
String movie1, movie2;
ListIterator iter1, iter2;

iter1 = movies.listIterator();

while (iter1.hasNext())
{
  movie1 = (String)iter1.next();
  if (!iter1.hasNext())  // if this is the last element
    break;

  iter2 = movies.listIterator(iter1.nextIndex());
     // Start the inner loop at the current position
     //   of iter1

  while (iter2.hasNext())
  {
    movie2 = (String)iter2.next();
    if (movie1.equals(movie2))
      System.out.println("Duplicate name: " + movie1);
  }
}
```

2.7 *Lab:* Movie Actors Index

A movie database consists of a movies file and an actors file. The task is to produce a cross-index, listing for each actor or actress all the movies that he or she starred in. Use linked lists in this project.

The movies file lists movies in chronological order by date of release. It is a text file. Each line has the year of release, the movie title, and the list of actors and actresses starring in the movie. Use `movies.txt` in the `Ch02\movies` folder on your student disk as an example or create your own. The actors file contains names of actors and actresses, one name per line. Use `actors.txt` in the `Ch02\movies` folder or create your own. The actor cross-index should display the actor's name and the list of movies he or she starred in, starting from the most recent ones.

Follow these steps:

1. Write a class `LinkedListFromFile` derived from `LinkedList`. In addition to the default (no-args) constructor, supply a constructor that takes the file name as an argument, reads lines of text from the file, and adds them to the list (in the same order). Recall that the `EasyReader` class (available at this book's companion web site) provides an easy way to open a text file and read data from it. For example:

```
EasyReader file = new EasyReader(fileName);
if (file.bad())
{
  ... // display an error message and quit
}

String str;
while ((str = file.readLine()) != null)
{
  ...    // Add str to the list
}
```

2. Write a main class that creates two linked lists, one from the `actors.txt` file and the other from the `movies.txt` file. Use your `LinkedListFromFile` class to construct the lists. For each actor or actress in the actors list, scan the list of movies and print out all those in which that person stars. Use a simple iterator to obtain the names of actors from the list. Use a list iterator (a `ListIterator` object) to obtain records from the list of movies. Iterate backwards in the list of movies, starting at the end, to display the more recent movies first.

   Assume that the names of actors start in a fixed position in the movie record. If the record contains additional information (e.g., the movie director's name), assume that it starts at a fixed position after the names of the actors. Search for a matching actor name only in a substring that cuts to the relevant portion of the record.

3. Test your program thoroughly. In particular, make sure that the first and the last elements in each list are handled properly. Redirect the console output to a file or use the `EasyWriter` class to write the output to a file instead of the console window. Then you can examine the output carefully and make sure that it matches the data.

2.8 Summary

A data structure combines data organization with methods for accessing and manipulating the data. The "List" is a data structure that contains a number of values arranged in a linear sequence. It provides methods for inserting a value into the list, removing a value from the list, and traversing the list (i.e., processing all values in sequence, visiting each value once).

The Java library `List` interface provides the `size`, `isEmpty`, `get`, `add`, `set`, and `remove` methods, summarized in Figure 2-1. The `ArrayList` and `LinkedList` library classes implement the `List` interface. These classes assume that the values in the list have the type `Object`. If you need to put values of a primitive data type such as `int`s or `double`s into the list, you need to first convert them into objects using the appropriate wrapper class, `Integer` or `Double`.

It is assumed that the elements in the list are indexed starting from 0. "Logical" indices are used even if a list is not implemented as an array. The `size()` method returns the number of values in the list. The `get(i)` call returns the element indexed by `i`. This method and other methods that use indices generate a run-time error if an index is out of bounds.

In an `ArrayList`, the elements occupy consecutive memory locations. In a linked list, the information is stored in nodes that may be scattered in memory, but each node, in addition to the value of the element, holds a reference to (the address of) the next node. A node of a linked list can be represented by a simple class `ListNode` (Figure 2-3). If `head` is a reference to the first node in the list, the statement

```
head = new ListNode(newValue, head);
```

appends a node holding `newValue` at the beginning of the list.

A traversal of a list is an operation that visits all the elements of the list in sequence and performs some operation or displays the value of each element. The following `for` loop can be used to conveniently traverse a linked list:

```
ListNode node;
for (node = head; node != null; node = node.getNext())
{
  SomeClass value  = (SomeClass)node.getValue();
  ... // process value
}
```

The `get(i)` and `set(i)` methods work for both `ArrayList` and `LinkedList` classes. Due to the direct-access property of arrays, these methods are fast and

efficient for `ArrayList`. However, they may be quite inefficient for `LinkedList` because they need to scan through the list to find the *i*-th element.

On the other hand, the specialized methods `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst`, and `removeLast` work fast for a `LinkedList`.

Both implementations of a list, as an array or as a linked list, have their advantages and disadvantages. A programmer has to choose carefully between the two, depending on the situation.

For lists implemented as `LinkedList`, it is better to use iterators for traversing the list. `Iterator` is a library interface; a class that implements `Iterator` supplies the methods `hasNext`, `next`, and `remove`. An iterator is obtained from the list itself, by calling its `iterator` method. For example:

```
Iterator iter = list.iterator();
while (iter.hasNext())
{
  SomeClass value = (SomeClass)iter.next();
  ... // process value
}
```

A more comprehensive `ListIterator` object is returned by `List`'s `listIterator` method. A `ListIterator` can start iterations at any specified position in the list and can proceed forward or backward. For example:

```
ListIterator iter = list.listIterator(list.size());
while (iter.hasPrevious())
{
  SomeClass value = (SomeClass)iter.previous();
  ... // process value
}
```

The `ListIterator` methods are summarized in Figure 2-10.

Exercises

1. Which of the following statements compile with no errors (assuming that the necessary library names are imported)?

(a) `List list = new List();`
(b) `List list = new ArrayList();`
(c) `List list = new LinkedList();`
(d) `ArrayList list = new List();`
(e) `ArrayList list = new ArrayList();`
(f) `LinkedList list = new List();`
(g) `LinkedList list = new ArrayList();`

2. Mark true or false and explain:

(a) A list can contain multiple references to the same object. _____ ✓
(b) The same object may belong to two different lists. _____
(c) `java.util.List`'s `remove` method destroys the object after it has been removed from the list. _____ ✓
(d) `java.util.List`'s `add` method makes a copy of the object and adds it to the list. _____
(e) Two variables can refer to the same list. _____ ✓

3. Write a method

```
public void removeFirstLast(List list)
```

that removes the first and the last elements from the list. ✓

4. Finish the following method using indices:

```
/** list contains Integer objects.  Removes the
 *  largest value from list.  (If several
 *  elements have the largest value, removes the
 *  first one of them.)
 */
public void removeMax(List list)
```

5. Write a method ✓

```
public void append(List list1, List list2)
```

that appends objects from `list2` to `list1` using indices.

6. What is the primary reason for using iterators rather than indices with Java library classes that implement `java.util.List`?

7. (a) Rewrite the method in Question 4 using an iterator.
(b) Rewrite the method in Question 5 using an iterator.

8. Fill in the blanks in the initialization of `node3`, `node2`, `node1` and `head`, so that `node1`, `node2`, and `node3` form a linked list (in this order) referred to by `head`. ✓

```
ListNode node3 = new ListNode("Node 3",________ );
ListNode node2 = new ListNode("Node 2",________ );
ListNode node1 = new ListNode("Node 1",________ );
ListNode head = ________ ;
```

9. Fill in the blanks in the following method:

```
/** Returns true if the list referred to by head
 *   has at least two nodes, false otherwise.
 */
public boolean hasTwo(ListNode head)
{
  return ____________________________________ ;
}
```

10. Write a method

```
public ListNode removeFirst(ListNode head)
```

that unlinks the first node from the list and returns the head of the new list. Your method should throw `NoSuchElementException` when the original list is empty. ✓

11. Write a method

```
public int size(ListNode head)
```

that returns the number of nodes in the list referred to by `head`:

(a) using a `for` loop
(b) using recursion.

12. `head` is the fist node of a non-empty linked list. Write a method

```
public void add(ListNode head, Object value)
```

that appends a new node holding `value` at the end of the list. ✓

13. Fill in the blanks in the method below that takes the list referred to by `head`, builds a new list in which nodes have the same information but are arranged in reverse order, and returns the head of the new list. The original list remains unchanged. Your solution must use a `for` loop (not recursion).

```
public ListNode reverseList(ListNode head)
{
  ListNode node, newNode, newHead = null;

  for ( ____________________________________________ )
  {
      ____________________________________________
      ...
  }

  return newHead;
}
```

14. Write a method

```
public ListNode concatenateStrings(ListNode head)
```

that takes the list referred to by `head`, builds a new list, and returns its head. The original list contains strings. The *k*-th node in the new list should contain the concatenation of all the strings from the original list from the first node up to and including the *k*-th node. For example, if the original list contains strings `"A"`, `"B"`, `"C"`, the new list should contain strings `"A"`, `"AB"`, `"ABC"`.

15.■ Write a method

```
public ListNode mix(ListNode head1, ListNode head2)
```

that takes two lists of equal length and makes a new list, alternating values from the first and the second list. For example, if `head1` refers to the list `"A"`, `"B"` and `head2` refers to the list `"1"`, `"2"`, the combined list is `"A"`, `"1"`, `"B"`, `"2"`. The method returns the head of the combined list.

16. Write a method that is similar to the one in Question 15, but works with arguments of the type `java.util.List` and returns a `java.util.LinkedList`:

```
public LinkedList mix(List list1, List list2)
```

Use iterators on `list1` and `list2` and the `add` method for adding values to the combined list. ✓

17.■ Write a method

```
public ListNode rotate(ListNode head)
```

that takes a linked list referred to by `head`, splits off the first node, and appends it at the end of the list. The method should accomplish this solely by rearranging links: do not allocate new nodes or move objects between nodes. The method should return the head of the rotated list.

18.■ A list referred to by `head` contains strings arranged alphabetically in ascending order. Write a method

```
public ListNode insertInOrder(ListNode head, String s)
```

that inserts `s` into the list, preserving the order. If `s` is already in the list, it is not inserted. The method should return the head of the updated list.

19.■ Write a method

```
public ListNode middleNode(ListNode head)
```

that returns the middle node (or one of the two middle nodes) of a linked list. Design this method using no recursion and only one loop.

20.◆ Let us say that a string matches a pattern (another string) if the pattern is at least as long as the string and for every non-wildcard character in the pattern the string has the same character in the same position. (The wildcard character is `'?'`.) For example, both `"New York"` and `"New Jersey"` match the pattern `"New ???????"`. Write a method

```
public ListNode moveToBack(ListNode head, String pattern)
```

that takes a list of strings referred to by `head` and moves all the strings that match `pattern` to the end of the list, preserving their order. Your method must work by rearranging links; do not allocate new nodes or use temporary arrays or lists. The method should return the head of the updated list.

21. (a) Write a reasonably efficient method

```
public ArrayList copyToArrayList(LinkedList list)
```

that copies a given `LinkedList` into a new `ArrayList` and returns the copy. (Note that the `ArrayList` class has a constructor that lets you specify the initial capacity of the list.) ✓

(b) The same as in Part (a) but in reverse: write a method that copies from an `ArrayList` to a `LinkedList`

```
public LinkedList copyToLinkedList(ArrayList list)
```

Use subscripts for the `ArrayList` and the efficient `addLast` method for the `LinkedList`.

22.■ A list contains n+1 `Double` values a_0, ..., a_n. Write a method

```
public double sum2(List list)
```

that calculates $a_0a_1 + a_0a_2 + ... + a_0a_n + a_1a_2 + ... + a_{n-1}a_n$ — the sum of products for all pairs of values. Do not use subscripts.