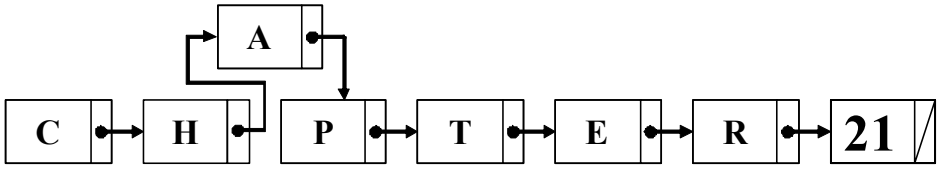

Lists and Iterators

21.1 Prologue

This chapter begins a series of chapters where we take a closer look at data structures. The Java collections framework provides a rich set of tools for incorporating various data structures into programs. In the previous chapter, we discussed a subset of these tools and when and how to use them. That's what most programmers need to know. However, we want to go a little beyond that and see how some of these data structures are implemented. We will proceed in a "do-it-yourself" manner, even though the Java library already has all the implementations. Of course we won't re-implement the library classes completely: we will just get a feel for how they work. We begin with lists.

The easiest implementation of a list is an array. As we've seen, 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 are held in *nodes* that can be scattered in various locations in memory, but each node contains a reference to the next one. The last node in the list points to nothing, so instead of a reference to the next node it holds `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.

As we know from Chapter 20, the `java.util.LinkedList` class implements a list as a doubly-linked list. In this chapter we will learn the basics of implementing our own linked lists: a singly-linked list, a linked list with a tail, a doubly-linked list, and a circular list.

21.2 Singly-Linked List

> **In a *singly-linked list*, each element is stored in a larger structure, called a *node*. In addition to the element's value, the node holds 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 21-1).

```
// Represents a node of a singly-linked list.

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

  public ListNode(Object v)
  {
    value = v;
    next = null;
  }

  public ListNode(Object v, ListNode nx)
  {
    value = v;
    next = nx;
  }

  public Object getValue() { return value; }
  public ListNode getNext() { return next; }

  public void setValue(Object v) { value = v; }
  public void setNext(ListNode nx) { next = nx; }
}
```

Figure 21-1. A class that represents a node in a singly-linked list (JM\Ch21\Lists\ListNode.java)

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 just a reference to an object, not the object itself. A reference 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.

We do not want to deal here with the rather complicated syntax for defining "generic" classes (which work with any specified type of objects). Therefore, the type of `value` in `ListNode` is `Object`, and we will use a cast to the appropriate type when we retrieve the value from a node.

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, constructors, 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 singly-linked list is accessed through a reference to its first node. This reference is held in a variable that we named `head`. When a program creates a linked list, it usually starts with an empty list — the `head` reference is `null`:

```
ListNode head = null;
```

To create the first node, we call `ListNode`'s constructor and store a reference to the new node in `head`:

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

This results in a list with one node (Figure 21-2).

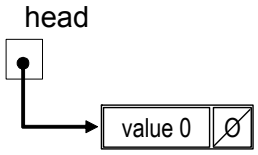

Figure 21-2. A linked list with only one node

The first node in a non-empty linked list is called the head 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 21-3.

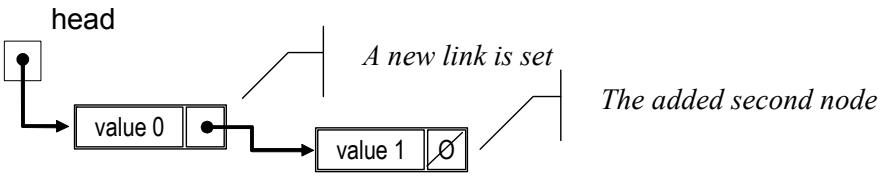

Figure 21-3. The second node is appended to the list

Now `head` refers to the first node of the list and `head.getNext()` returns a reference to the second node.

Diagrams like this one help us understand how links in a linked list are reassigned in various operations.

Figure 21-4 shows the field and constructors of the `SinglyLinkedList` class. (We are building this class just to learn how a linked list works; we will continue using the

much more advanced `java.util.LinkedList` class in our projects.) `SinglyLinkedList` has two fields, `ListNode head` and `int nodeCount`, and two constructors. The no-args constructor creates an empty list. The constructor with one parameter, `Object[] values`, creates a list that contains all elements from the array `values` in the same order. When we build this list, we keep a reference to the last node, `tail`, append a new node to the tail node, then update `tail`.

```
// Implements a singly-linked list.

import java.util.Iterator;

public class SinglyLinkedList
{
  private ListNode head;
  private int nodeCount;

  // Constructor: creates an empty list
  public SinglyLinkedList()
  {
    head = null;
    nodeCount = 0;
  }

  // Constructor: creates a list that contains
  // all elements from the array values, in the same order
  public SinglyLinkedList(Object[] values)
  {
    ListNode tail = null;
    for (Object value : values) // for each value to insert
    {
      ListNode node = new ListNode(value, null);
      if (head == null)
        head = node;
      else
        tail.setNext(node);
      tail = node;    // update tail
    }

    nodeCount = values.length;
  }
  ...
}
```

Figure 21-4. The constructors of the `SinglyLinkedList` class (`JM\Ch21\Lists\SinglyLinkedList.java`)

Figure 21-5 shows the resulting linked list.

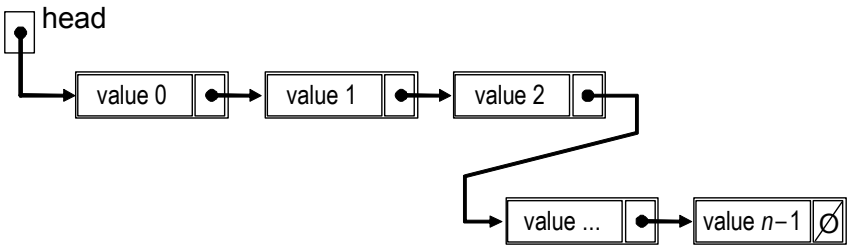

Figure 21-5. A singly-linked list

We can easily attach a new node at the head of the list. For example:

```
public void addFirst(Object value)
{
  ListNode node = new ListNode(value, null);
  node.setNext(head);
  head = node;
}
```

Or simply:

```
public void addFirst(Object value)
{
  head = new ListNode(value, head);
}
```

The above method creates a new node and sets its `next` field to the old value of `head`. It then sets `head` to refer to the newly created node (Figure 21-6).

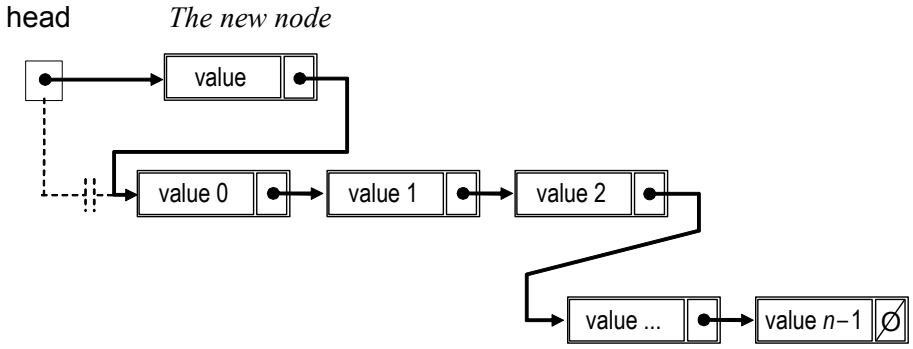

Figure 21-6. A new node inserted at the head of a linked list

But in an `addLast` method, you have to traverse the list to find the last node, then append the new node to the last one.

21.3 Traversals

It is easy to traverse a singly-linked list with a `for` loop that goes from one node to the next:

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

It is also not very hard to make iterators and "for each" loops work with our `SinglyLinkedList` class. To achieve this, we need to make our linked lists "iterable." Here is a recipe:

1. State that `SinglyLinkedList` implements `Iterable<Object>`. `Iterable<T>` is an interface in the `java.lang` package (built into Java). It specifies only one method:

   ```
   Iterator<T> iterator();
   ```

2. Add a method `iterator` to the `SinglyLinkedList` class. This method constructs an iterator for a `SinglyLinkedList`, passing the `head` of the list to the iterator's constructor as a parameter:

   ```
   public Iterator<Object> iterator()
   {
     return new SinglyLinkedListIterator(head);
   }
   ```

   The method returns the newly constructed iterator object.

3. Write the `SinglyLinkedListIterator` class, which implements `java.util.Iterator<Object>`. A simplified version of this class is shown in Figure 21-7.

```
// Implements an iterator for a SinglyLinkedList.

import java.util.Iterator;
import java.util.NoSuchElementException;

public class SinglyLinkedListIterator implements Iterator<Object>
{
  private ListNode nextNode;

  // Constructor
  public SinglyLinkedListIterator(ListNode head)
  {
    nextNode = head;
  }

  public boolean hasNext()
  {
    return nextNode != null;
  }

  public Object next()
  {
    if (nextNode == null)
      throw new NoSuchElementException();

    Object value = nextNode.getValue();
    nextNode = nextNode.getNext();
    return value;
  }

  // Not implemented.
  public void remove()
  {
    throw new UnsupportedOperationException();
  }
}
```

Figure 21-7. JM\Ch21\Lists\SinglyLinkedListIterator.java

In Figure 21-7 the iterator class is implemented as a regular public class. It would be stylistically more appropriate to make `SinglyLinkedListIterator` a *private inner class* in `SinglyLinkedList`, because `SinglyLinkedList` is the only class that uses this type of iterator. We have avoided inner classes in this book, because, from the OOP standpoint, the relationship between an object of an inner class and an object of its embedding outer class is not well defined. In particular, the code in an inner class has access to the private fields and methods of the embedding class. But

while inner classes are philosophically confusing, their syntax is straightforward. Here, just replace `public class` with `private class` and embed the whole definition of `SinglyLinkedListIterator` inside the `SinglyLinkedList` class body. Something like this:

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class SinglyLinkedList implements Iterable<Object>
{
  < ... fields, constructors, and methods >

  private class SinglyLinkedListIterator implements Iterator<Object>
  {
    ...
  }
}
```

A more elaborate iterator implementation would have the `remove` method implemented (which means keeping track of not just the current node but also the two previous ones). It would also check that the list has not been modified outside the iterator.

21.4 *Lab:* Implementing a Singly-Linked List

Finish the `SinglyLinkedList` class (`SinglyLinkedList.java` in `JM\Ch21\Lists`). Implement all of the `java.util.List` methods shown in Figure 20-4 on page 488 (for elements of the `Object` type only) except the two `listIterator` methods. Also implement a `toString` method: it should return a string in the same format as the library classes that implement `List`. Test your class thoroughly.

21.5 Linked List with a Tail

For a singly-linked list, adding an element at the beginning is an $O(1)$ operation, while adding an element at the end is an $O(n)$ operation (where n is the number of nodes), because you have to traverse the whole list to find the last node. (Incidentally, for arrays, it is exactly the opposite.) With a small modification, however, we can make appending an element at the end efficient, too. All we have to do is keep track of a reference to the last node (Figure 21-8) and update it appropriately when we add or remove nodes. Such a structure is called a *linked list with a tail*.

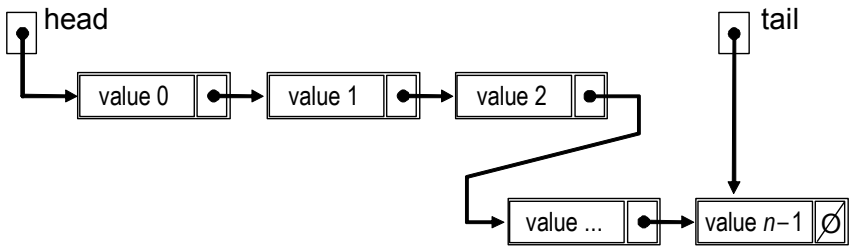

Figure 21-8. A singly-linked list with a tail

Write a class `LinkedListWithTail`. Define two fields: `head` and `tail`. `tail` should refer to the last node of the list. Provide a no-args constructor that creates an empty list, setting both `head` and `tail` to `null`. Implement the `isEmpty`, `peek`, `add`, and `remove` methods specified by the `java.util.Queue<Object>` interface. (Make sure that all these methods run in $O(1)$ time.)

21.6 Doubly-Linked List and Circular List

We have already mentioned doubly-linked lists in Chapter 20 (Figure 20-6 on page 490). In a doubly-linked list, each node holds references to both the next node and the previous node (Figure 21-9).

```
// Represents a node of a doubly-linked list.
public class ListNode2
{
  private Object value;
  private ListNode2 previous;
  private ListNode2 next;

  public ListNode2(Object v)
  {
    value = v;
    next = null;
    previous = null;
  }

  public ListNode2(Object v, ListNode2 pr, ListNode2 nx)
  {
    value = v;
    previous = pr;
    next = nx;
  }

  public Object getValue()  { return value; }
  public ListNode2 getPrevious() { return previous; }
  public ListNode2 getNext() { return next; }

  public void setValue(Object v) { value = v; }
  public void setPrevious(ListNode2 pr) { previous = pr; }
  public void setNext(ListNode2 nx) { next = nx; }
}
```

Figure 21-9. A class that represents a node of a doubly-linked list (JM\Ch21\Lists\ListNode2.java)

The advantage of a doubly-linked list over a singly-linked list is that we can traverse it in both directions, and we have direct access to the last node. A disadvantage is that it takes more space than a singly-linked list.

Like a linked-list with a tail, a doubly-linked list can be implemented with two fields, `head` and `tail`, that would hold references to the first and last nodes of the list, respectively. In an empty list, both `head` and `tail` are `null`.

To add a value at the end of the list, we have to write something like this:

```
public void addLast(Object value)
{
  ListNode2 newNode = new ListNode2(value, tail, null);
  if (head == null)
  {
    head = newNode;
    tail = newNode;
  }
  else
  {
    tail.setNext(newNode);
  }
  tail = newNode;
}
```

❖ ❖ ❖

In a *circular* doubly-linked list, the "next" link in the last node refers back to the first node, and the "previous" link in the first node refers to the last node (Figure 21-10).

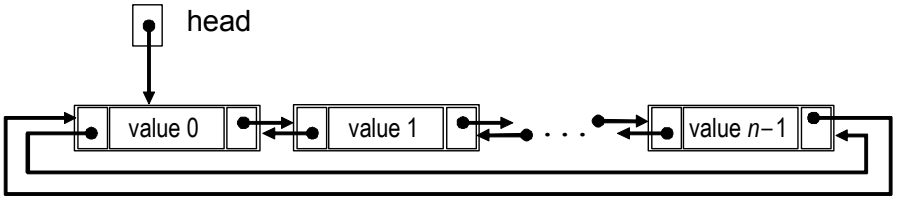

Figure 21-10. A doubly-linked circular list

In a circular list we do not need to keep a separate reference to the tail because we can access it as `head.getPrevious()`.

❖ ❖ ❖

In a doubly-linked list with separate references to the head and tail, we have to handle the case of an empty list separately. It is possible to streamline the code if we combine the `head` and `tail` references in one dummy node, which does not hold any valid value. Let's call this node `header`. Our list then becomes circular (Figure 21-11) and always has at least one node. This is how `java.util.LinkedList` is implemented.

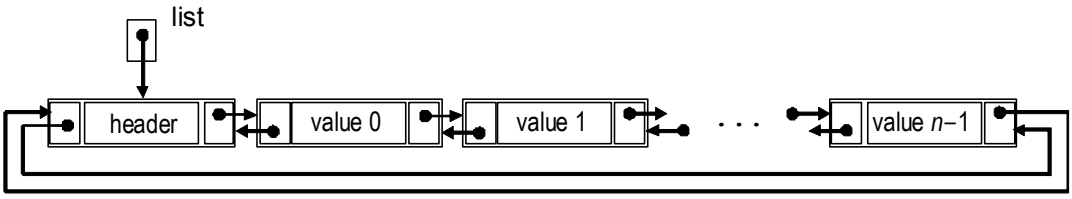

Figure 21-11. A circular doubly-linked list with a header node

In this implementation, an empty list has only one node, `header`, and its `next` and `previous` fields refer to `header` itself:

```
// Constructs an empty list:
public DoublyLinkedListWithHeader()
{
  header = new ListNode2(null);
  header.setPrevious(header);
  header.setNext(header);
}
```

With a header node, the code for methods that insert and remove elements is a little shorter, because we don't have to handle the case of an empty list separately. For example, we can write a method to insert an element after a given node:

```
// Inserts value after node.
public void addAfter(ListNode2 node, Object value)
{
  ListNode2 newNode = new ListNode2(value, node, node.getNext());
  node.getNext().setPrevious(newNode);
  node.setNext(newNode);
}
```

This code also works for inserting the first or the last node. For example:

```
public void addLast(Object value)
{
  addAfter(header.getPrevious(), value);
}
```