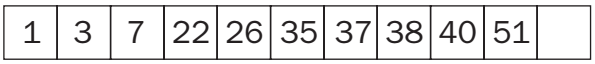
Notes on Linked Data Structures
adapted by Mike Clancy from notes by Jonathan Shewchuk

Linked lists

Consider how we might store a collection of int values. One way uses an array:



Here is code that implements a collection as an array.

```
public class Collection {
   int myItems[ ];
   int myCount;

   public Collection ( ) {
      items = new int[10];                        // The number "10" is arbitrary.
      myCount = 0;
   }

   public void insertItem (int newItem, int location) {
      if (myCount == myItems.length) {
         int a[ ] = new int[2*myItems.length]; // Alloc new array, twice as big.
         for (int k=0; k<myCount; k++) {
            a[k] = myItems[k];                 // Copy items to the bigger array.
         }
         myItems = a;              // Replace the too-small array with the new one.
      }
      for (int k=myCount; k>location; k--) {        // Shift items to the right.
         myItems[k] = myItems[k-1];
      }
      myItems[location] = newItem;
      myCount++;
   }
}
```

This representation has disadvantages for certain uses of the collection. For example, arrays have a fixed length that can't be changed. If we guessed wrong about how many values would be stored when we constructed the array, we might encounter a situation where we want to add a value, but the array is full. We then have to allocate and fill a whole new array.

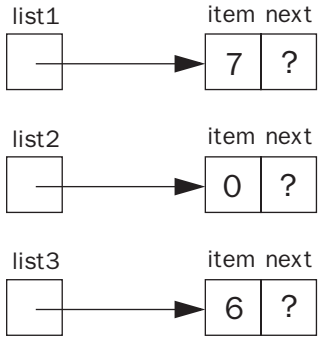
The values displayed above are stored in increasing order. If we want to insert a value at the beginning or middle of the array, we have to slide a bunch of values over one place to make room. This shifting of elements can be expensive.

We can avoid these problems by choosing a linked representation for a collection. A linked list is made up of *nodes*. Each node has two components, an item and a link—a reference to the next node in the list. (See the earlier section "Boxes and Arrows" in this document.) Not surprisingly, a node is an object in Java:

```
class ListNode {                          // ListNode is a recursive type.
   int item;
   ListNode next;                         // Here we use ListNode before
}                                         //  we've finished defining it.
```

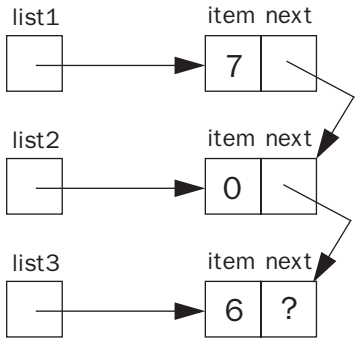
Let's make some ListNodes.

```
ListNode list1 = new ListNode ( );
ListNode list2 = new ListNode ( );
ListNode list3 = new ListNode ( );
list1.item = 7;
list2.item = 0;
list3.item = 6;
```



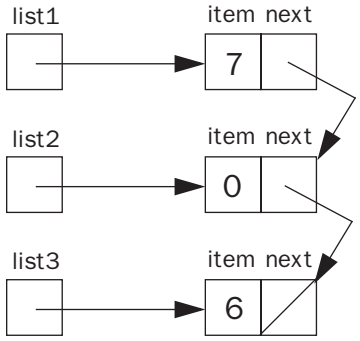
Now let's link them together.

```
list1.next = list2;
list2.next = list3;
```



(Note that next contains a reference to a ListNode object, so the arrow may be drawn to point anywhere on the object.) What about the last node? We need a reference that doesn't reference anything. In Java, this is called null and is drawn with a slash.

```
list3.next = null;
```



To simplify programming, let's add some constructors.

```
ListNode (int item) {
   this.item = item;
   next = null;
}
```

```
ListNode (int item, ListNode next) {
   this.item = item;
   this.next = next;
}
```

These constructors allow easy construction of a short linked list.

```
ListNode list1 = new ListNode(1, new ListNode(2, new ListNode(3)));
```

Inserting an item immediately after a given node is a constant-time operation, and the list can keep growing until memory runs out. The next method inserts a new item into the list immediately after this.

```
public void insertAfter (int item) {
   next = new ListNode (item, next);
}
```

A disadvantage of a linked list is that, to access an item in the middle of the list, we must go through all the preceding nodes. The method below recursively finds the nth item in a list whose first item, numbered #0, is this.

```
public ListNode nth (int position) {
   if (position == 0) {
      return this;
   } else if ((next == null) || (position < 0)) {
      return null;
   } else {
      return next.nth(position - 1);
   }
}
```

For greater generality, let's change ListNodes so that each node contains not an int, but a reference to any Java object. In Java, we can accomplish this by declaring a reference of type Object.

```
class ListNode {
   Object item;
   ListNode next;
}
```

A List class

There are two problems with using only list nodes to represent a collection.

- Suppose p and q are references to the same shopping list. Suppose we insert a new item at the beginning of the list thusly:

  ```
  p = new ListNode("soap", p);
  ```

  q doesn't point to the new item; q still points to the second item in p's list. If q goes shopping for p, it'll forget to buy soap.

- How do you represent an empty list? The obvious way is list = null. However, Java won't let you call a ListNode method—or any method—on a null object. One might wish to provide an isEmpty method:

```
public boolean isEmpty ( ) {
   return this == null;
}
```

  If, however, you call list.isEmpty( ) when list is null, you'll get a run-time error, even though list is declared to be a ListNode reference.

The solution is a separate Collection class, whose job is to maintain the head (first node) of the list. We will put most of the methods that operate on lists in the Collection class, rather than the ListNode class.

```
public class Collection {

   private ListNode myHead;
   private int myCount;

   public Collection ( ) {                    // Here's how to represent an empty list.
      myHead = null;
      myCount = 0;
   }

   public void insertFront (Object item) {
      myHead = new ListNode(item, myHead);
      myCount++;
   }
}
```

Now, when an item is inserted at the front of a Collection, every reference to that Collection can see the change. Another advantage of the Collection class is that it can keep a record of the Collection's size (number of ListNodes). Hence, the size can be determined more quickly than if the ListNodes had to be counted.

Note that the Collection class hides its data members, thereby hiding the implementation of the class as a linked list. If it turned out to be appropriate later to represent a collection as an array, the representation could be updated and none of the users of the Collection class would need to know. A related advantage of this information hiding is that the Collection methods can enforce two *invariant* properties:

- A list is never circularly linked; in a nonempty list, there is always a "tail" node whose next reference is null.
- A Collection's myCount variable is always correct.

Both these goals are accomplished by making sure that only the methods of the Collection class can change the lists' internal data structures. Collection ensures this by two means: (1) the fields of the Collection class (myHead and myCount) are declared private; (2) no Collection method returns a ListNode. The first rule is necessary so that an evil tamperer can't change the fields. The second rule prevents the evil tamperer from changing list items, truncating a list, or creating a cycle in a list.

Accessing ListNode information

We have so far not said anything about whether ListNodes and their fields are public or private. One way to define ListNode is as a public class in its own file. In order to restrict other clients of the ListNode class from tampering with links or stored items, we should define the ListNode fields as private. Then, however, for methods such as nth to be able to examine list elements, we would also need to provide accessor methods for the data stored in a ListNode. This seems like too much access. ListNodes should not be used without a list anyway.

A second option is to define the ListNode class in the same file as the Collection class. When one omits the keywords public, private, and protected from a class or field definition, one gets what's called *package visibility*. Two classes in the same file are also in the same package. (Packages are described in more detail in chapter 17 of *Head First Java*.) A ListNode could then be defined as follows:

```
class ListNode {
   Object item;
   ListNode next;

   // Constructors go here.
}
```

A third option, which we prefer, is to define the ListNode class *inside* the Collection class, restricting its access as much as possible. (ListNode is then an example of an *inner class*, described in detail in chapter 12 of *Head First Java*.) Figure 1 displays the code that results from this approach.

One final issue is the definition of a ListNode item as Object. This allows each list node to contain any object (recall that every class implicitly inherits from the Object class). It may not contain simple integers, characters, and so on, since these are not objects. Java, however, provides *wrapper classes* named Integer, Character, Boolean, etc. precisely for this purpose. All these wrapper classes provide methods to supply and access the simple value within:

`public Integer (int value);`
`public Character (char value);`
`public Boolean (boolean value);`
	a constructor with the appropriate simple argument

`public Integer (String s);` `public Boolean (String s);`	a constructor with a string argument

`public int intValue ( );`
`public char charValue ( );`
`public boolean booleanValue ( );`
	an access method to the stored simple value

`public boolean equals (Object obj);`	a comparison method
`public String toString ( );`	a method that converts the stored value to printable form

```
public class Collection {

   // Initialize an empty collection.
   public Collection ( ) {
      myHead = null;
      myCount = 0;
   }

   // Attach the given object to the front of the collection (element #0).
   public void insertFront (Object obj) {
      myHead = new ListNode (obj, myHead);
      myCount++;
   }

   // Return the element of the collection at the given position.
   // The first element is at position 0, the second at position 1, and so on.
   public Object nth (int position) {
      if (position >= myCount || position < 0) {
         return null;
      }
      ListNode p = myHead;
      for (int k=0; k<position; k++) {
         p = p.next;
      }
      return p.item;
   }

   private class ListNode {

      public ListNode (Object item) {
         this.item = item;
         next = null;
      }

      public ListNode (Object item, ListNode next) {
         this.item = item;
         this.next = next;
      }

      public Object item;
      public ListNode next;
   }

   private ListNode myHead;
   private int myCount;
}
```

Figure 1

Linked list implementation of a collection,
using an inner ListNode class

Consider now how the items from the list would be used. Sample code appears below. What expression should be assigned to sum in the next-to-last line?

```
Collection values = new Collection ( );
values.insertFront (new Integer (3));
values.insertFront (new Integer (1));
values.insertFront (new Integer (7));
int sum = 0;
for (int k=0; k<3; k++) {
   sum += the nth item from the list;
}
```

An obvious thing to guess is

```
sum += values.nth (k).intValue ( );
```

(Recall that the elements of the list are Integer objects; the actual stored int value must be accessed via the intValue method.) Unfortunately, this produces a compiler error:

```
Method intValue() not found in class java.lang.Object.
         sum += values.nth (k).intValue ( );
                                        ^
```

What's needed is a *cast*, to inform the Java compiler that the kind of Object reference that nth will return is a reference to an Integer. The corrected statement is

```
sum += ((Integer) values.nth (k)).intValue ( );
```

The extra set of parentheses are needed because the dot operator has higher precedence than the cast.

This pattern of processing is fairly common in Java. The java.util package provides a number of classes—e.g. ArrayList, LinkedList, and HashMap—for storing collections of Objects, and all are accessed in this way. Here's how the example just discussed would be coded using an ArrayList:

```
ArrayList values = new ArrayList ( );
values.add (new Integer (3));
values.add (new Integer (1));
values.add (new Integer (7));
int sum = 0;
for (int k=0; k<3; k++) {
   sum += ((Integer) values.get (k)).intValue ( );
}
```