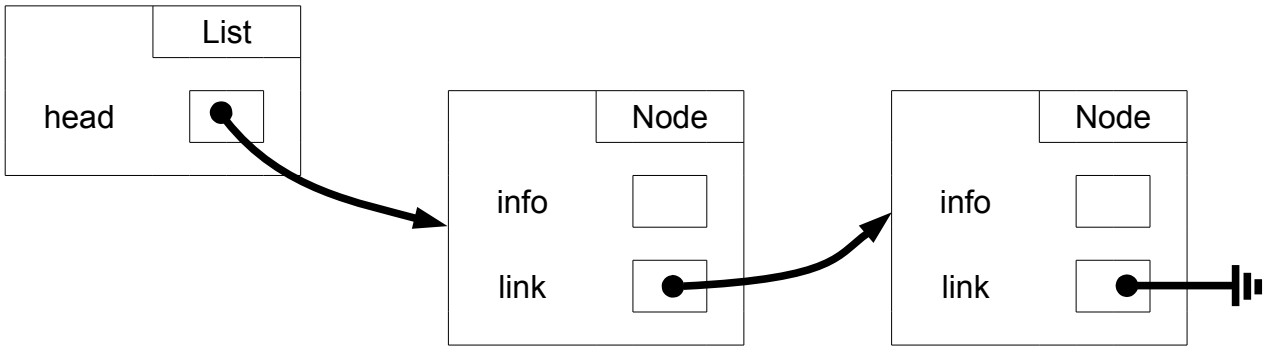
Dynamic Data Structures in Java – Linked Lists

An array or string is an example of a static data structure. Once it has been created, the size is fixed. There are ways to program around this limitation, but in general, the size is meant to be static. A dynamic data structure is intended to grow and shrink as the program executes, depending upon the data needs at the time.

One of the most common dynamic data structures is a list, or more specifically, a linear linked list. We say "linked" because each item (or node, or object) in the list is somehow connected to other items in the list. "Linear" indicates that the connections are only between adjacent items.

As a result, each node in the list must contain two broad pieces of information: (1) the desired information to be stored in the list, and (2) a connection to the next item in the list. Consider the following diagram.



This diagram indicates that there are two types of objects required for our linear linked list: (a) a single object of type **List**, which "points" to the first node of the list, and (b) multiple objects of type **Node**, which contain both the information fields (generically called "info" for now), and a link to the next node.

The link in the last node uses the symbol for an electrical ground, which has been commonly adopted in computer science to indicate a **null** reference (i.e., the link does not point to, or reference, anything). The **null** reference is, however, something we can test using boolean logic (e.g., if statements, looping conditions), so we can determine if we have reached the end of our list.

```
class List
{
      private Node head;
}

class Node
{
      int info;        // can be changed to anything we want
      Node link;       // reference to next node in list

      // constructor method
      Node (int i, Node n)
      {
            info = i;
            link = n;
      }
}
```

This first step in creating our **List** class presents a problem. Should the contents (fields) of the **Node** class be public or private? It is usually a good idea to keep the data fields (info) private, and for the **link** field, we definitely don't want other parts of the program to access it directly. Unfortunately, making this data private means the **List** class cannot access the **Node** fields, which is necessary for the list to function.

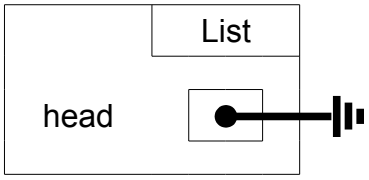
Java has the facility to work around this limitation by defining a class within a class. The *interior class* will be visible only to the *containing class* (essentially, the entire interior class is now private). Now only **List** methods will have access to the **Node** fields or methods (including the constructor).

```
class List
{
      private Node head;

      // inner class, visible only to the List class
      class Node
      {
            int info;       // can be changed to anything we want
            Node link;      // reference to next node in list

            // constructor method
            Node (int i, Node n)
            {
                  info = i;
                  link = n;
            }
      }
}
```

Creating a List

To create a new, empty list, with the **head** set to **null**, we use the default constructor (i.e., a constructor created automatically by Java) for the **List** class:

```
List myList = new List();        // create an empty list
```



Creating a Node

To insert a single node containing some information into this list, we need a method which will create a new **Node** object, store the information, and attach the Node to the head of the list.

```
public void addAtFront (int value)
{
      // create new Node, store data, reference previous head of list
      Node temp = new Node(value, head);
      // set head of list to be new Node
      head = temp;
}
```

Exercises Part 1:

1. Write an instance method printList() that will traverse the list and print each element.

2. Modify printList() with a message if the list is empty.

3. Write an instance method sum() that will traverse and sum all elements in the list.

4. Write an instance method deleteFirst() which removes the first node and leaves the rest of the list intact.

5. Write an instance method deleteLast() to remove the last node and leaves the rest of the list intact. Remember the last node should point to NULL.

6. Write an instance method countNodes() which returns the number of nodes.

7. Write an instance method addAtRear(value) which adds a node at the end of the list, which contains value and points to NULL.

Exercises Part 2:

1. Write an instance method, insert(int), which adds a new node in order (i.e., the info fields are in order). Note: assumes list is empty or currently in order, smallest to largest.

2. Write an instance method, delete(int), which removes the node with the specified value.

3. Write a boolean instance method, contains(int), which returns true if the list contains the specified value, otherwise it returns false.

4. Write an instance method, deleteAll(int), which removes ALL nodes with specified value.

5. Write an instance method, isOrderedIncreasing(), which checks if list is in order, small to large.

6. Write a boolean instance method, isIdentical(List), which compares current list object to the list object passed as a parameter.