Array-based lists.

Advantages: Very fast access of each item.
Disadvantages:
1. Insert item at beginning or middle; takes time proportional to length of array.
2. Array has a fixed length.

```java
public class List {
    int[] a;
    int lastItem;

    public List() {
        a = new int[10];
        lastItem = -1;
    }

    public void insertItem(int newItem, int location) {
        int i;

        for (i = lastItem; i >= location; i--)
            a[i + 1] = a[i];
        a[location] = newItem;
        lastItem++;
    }
}
```
public class List {
    int[] a;
    int lastItem;

    public List() {
        a = new int[10];
        lastItem = -1;
    }

    public void insertItem(int newItem, int location) {
        int i;
        if (lastItem + 1 == a.length) {
            int[] b = new int[2 * a.length];
            for (i = 0; i <= lastItem; i++) {
                b[i] = a[i];
            }
            a = b;
        }
        for (i = lastItem; i >= location; i--) {
            a[i + 1] = a[i];
        }
        a[location] = newItem;
        lastItem++;
    }
}
**LINKED LISTS** (a recursively linked data type)

Made up of nodes. Each node has:

- an item.
- a reference to next node in list. (Like "car" and "cdr".)

```java
public class ListNode {
    public int item;
    public ListNode next;
}
```

```java
ListNode l1 = new ListNode();
ListNode l2 = new ListNode();
ListNode l3 = new ListNode();
```

```java
l1.item = 7;
l2.item = 0;
l3.item = 6;
```

```java
l1.next = l2;
l2.next = l3;
l3.next = null;
```

```java
public ListNode(int i, ListNode n) {
    item = i;
    next = n;
}

public ListNode(int i) {
    this(i, null);
}
```

```java
ListNode l1 = new ListNode(7, new ListNode(0, new ListNode(6)));
```
Linked lists vs. array lists

Advantages of linked lists:

- Inserting item into middle of linked list takes constant time, if you have a ref to previous node.
- List can keep growing until memory runs out.

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

Disadvantage: Finding $n$th item of a linked list takes time proportional to $n$.
Start at head, walk $n-1$ nodes.

Lists of Objects | Reference any object.

```java
public class ListNode {
    public Object item;
    public ListNode next;
}
```
A List class

Problems with SListNode:

1. Insert new item at beginning of list:
   
   ```java
   X = new SListNode("soap", X);
   ```

2. How do you represent an empty list?
   
   ```java
   X = null;
   X.insertAfter(item);  // Run-time error.
   ```

Solution: Separate SList class maintain the head of the list.
public class SList {
    private SListNode head;
    private int size;

    public SList() {
        head = null;
        size = 0;
    }

    public void insertFront(Object Item) {
        head = new SListNode(item, head);
        size++;
    }
}
The "public" and "private" keywords

Private method or field: invisible & inaccessible to other classes.

Why?

1. To prevent data from being corrupted by other classes.
2. You can improve the implementation without causing other classes to fail.
public class Date {
    private int day;
    private int month;

    private void setMonth(int m) {
        month = m;
    }

    public Date(int month, int day) {
        [Implementation with error−checking code here.]
    }
}

public class EvilTamperer {
    public void tamper() {
        Date d = new Date(1, 1, 2006);
        d.day = 100; // Foiled !!!!
        d.setMonth(0); // Foiled again, !!!
    }
}
The interface of a class: prototypes for public methods, plus description of their behaviors. 

Abstract Data Type (ADT): A class with well-defined interface, but implementation details are hidden from other classes.

Invariant: A fact about a data structure that is always true. "A Date object always stores a valid date." Enforced by allowing access only through method calls.

Not all classes are ADTs! Some classes are data storage units, no invariants; field can be public.
The SLList ADT

Another advantage of SLList class: it enforces 2 invariants.
1. "size" is always correct.
2. A list is never circularly linked.

Both goals accomplished because only SLList methods can change the lists. SLList ensures this:
1. The fields of SLList (head & size) are "private".
2. No method of SLList returns an SLListNode.
Doubly-Linked Lists

Inserting/deleting at front of list is easy, end of list takes a long time.

```java
class DLListNode {
    Object item;
    DLListNode next;
    DLListNode prev;
}
```

class DLList {
    private DLListNode head;
    private DLListNode tail;
}

3

`head`

Insert & delete items at both ends in constant running time.

Removes the tail node (at least 2 items in DLList):

```java
    tail.prev.next = null;
    tail = tail.prev;
```

**Sentinel**: A special node that does not represent an item.
DLList invariants (with sentinel):

1. For any DLList d, d.head != null.
2. For any DLListNode x, x.prev != null.
3. For any DLListNode x, x.next != null.
4. if x.next == y, then y.prev == x.
5. if x.prev == y, then y.next == x.
6. A DLList’s "size" variable is # of DLListNode, NOT counting sentinel, accessible from sentinel by sequence of "next"s.

```java
class DLList {
    public long size() {
        return size;
    }
}
```

Empty DLList: Sentinel’s prev & next fields point to itself.

```java
public void removeBack() {
    if (head.prev != head) {
        head.prev = head.prev.prev;
        head.prev.next = head;
        size--;
    }
}
```