Arrays are limited in the sense they have a fixed size
- Resizing as needed must be done carefully and is not efficient
- A **linked structure** is the primary alternative to an array-based implementation of a collection
- **Initial** implementation idea: a class **Person** as part of a community

```java
public class Person{
    private String name;
    private String address;
    private Person next;
    // and whatever else
}
```

- Using only this one class, a linked structure of “nodes” is created
  - One Person object contains a link to another Person object
  - This second object contains a reference to a third Person, etc.
  - This type of object is sometimes called **self-referential**
  - This kind of relationship forms the basis of a **linked list**
- Has capacity limited only by memory in the computer
  - Is a **dynamic structure** because its size grows and shrinks as needed to accommodate the number of elements stored

- A simple linked list is only one kind of linked structure
- In a **doubly linked list**, each node in the list stores both a reference to the next element and a reference to the previous one
- Java’s library implementation uses doubly linked lists
- Using multiple links you can create non-linear structures

- A node may be inserted or deleted at **any location**
  - at the front of the list,
  - among the interior nodes, or
  - at the end of the list

- There are a few basic techniques when managing nodes on the list, no matter what the list is used to store

- Special care must be taken when dealing with the **first or last node** in the list so that the reference to the entire list is maintained appropriately
A flaw in the Person class is that the self-referential Person class must be designed so that it "knows" it may become a node in a linked list. This violates the goal of separating the implementation details from the parts of the system that use the collection. We better separate the details of the linked list structure from the elements that the list stores. Better: define a separate node class LinearNode that serves to link the elements together.

```java
/** LinearNode.java
 * @author Java Foundations
 */
package javafoundations;

public class LinearNode<T> {
    private LinearNode<T> next;
    private T element;
    // Creates an empty node
    public LinearNode() {
        next = null;
        element = null;
    }
    // Creates a node storing the specified element
    public LinearNode(T elem) {
        next = null;
        element = elem;
    }
```
public LinearNode<T> getNext() {
    return next;
}

public void setNext (LinearNode<T> node) {
    next = node;
}

public T getElement() {
    return element;
}

public void setElement (T elem) {
    element = elem;
}

// Four people
Person p1 = new Person("Ashley", "1600 Pennsylvania Ave");
Person p2 = new Person("Stella", "106 Central Street");
Person p3 = new Person("Jean", "123 Sesame Street");
Person p4 = new Person("Christine", "4 Yawkey Way");

LinearNode<Person> front; // Refers to first node in list
LinearNode<Person> temp; // Refers to new node as it is // being added to list

front = new LinearNode<Person>(p4); // List contains one node

// Add new node to front of list
front = temp; // Update "front"
// Add new node to front of list
temp = new LinearNode<Person>(p3); // Create new node
front = temp; // Update "front"

// Add new node to front of list
temp = new LinearNode<Person>(p2); // Create new node
temp.setNext(front); // Add to front of list
front = temp; // Update "front"

// Add new node to front of list
temp = new LinearNode<Person>(p1); // Create new node
temp.setNext(front); // Add to front of list
front = temp; // Update "front"

public static int getLength_rec(LinearNode<Person> L) {
    if (L == null) return 0;
    else return 1 + getLength_rec(L.getNext());
}

// Returns the number of nodes in list L. Uses a loop.
public static int getLength_loop(LinearNode<Person> L) {
    int count = 0;
    for (LinearNode<Person> current = L; current != null; current = current.getNext()) {
        count++;
    }
    return count;
}

// If the index is valid, returns the element in the list at the
// specified index. If the index is invalid, e.g., less than 0 or
// greater than or equal to the list length, then "null" is returned.
public static Person getPersonAtIndex(LinearNode<Person> L, int index)
{
    int count = 0;
    for (LinearNode<Person> current = L; current != null;)
    {
        if (count == index)
            return current.getElement();
        current = current.getNext();
        count++;
    }
    return null;
}
String Representation

```java
// Return a String representation of the specified list L.
public static String listToString(LinearNode<Person> L) {
}
```

Stack Implementation using Java’s LinkedList <E>

- The ADT LinkedList can be used to represent the items in a stack

```java
public void push(E item)
public E pop() throws NoSuchElementException
public E element() // same as peek
```

Methods for Adding LinkedList Nodes

```java
public void add(int index, E element)
    indices change automatically!
    if index > size() throws IndexOutOfBoundsException
public void addFirst (E element)
public void addLast (E element)
aka: public void add (E element)
```

```java
LinkedList<String> L1 = new LinkedList<String>();
L1.add(0, "eggs");
L1.addFirst("milk");
L1.addLast("bread");
L1.add(2, "chicken");
System.out.println("contents of L1: " + L1);
```
Methods for Getting List Items

public E getFirst ()
public E getLast ()
public E get (int index)

LinkedList<String> L2 = new LinkedList<String>();
for (int i = L1.size()-1; i >= 0; i--)
    L2.add(L1.get(i));
System.out.println("contents of L2: "+ L2);

What does L2 contain?

Methods for Setting and Removing Items

public E set (int index, E element)
public E remove (int index)
public E removeFirst ()
public E removeLast ()
L1.remove(1);
L1.set(2, "beer");
L1.set(1, L1.removeFirst());
L1.addFirst(L1.getLast());
L1.add(1, "butter");
System.out.println("new contents of L1: "+ L1);
new contents of L1: (what happened to the beer?)