

# Lesson 8

## The List Data Structure:



Sequential Unfoldment of  
Natural Law

# Wholeness Statement



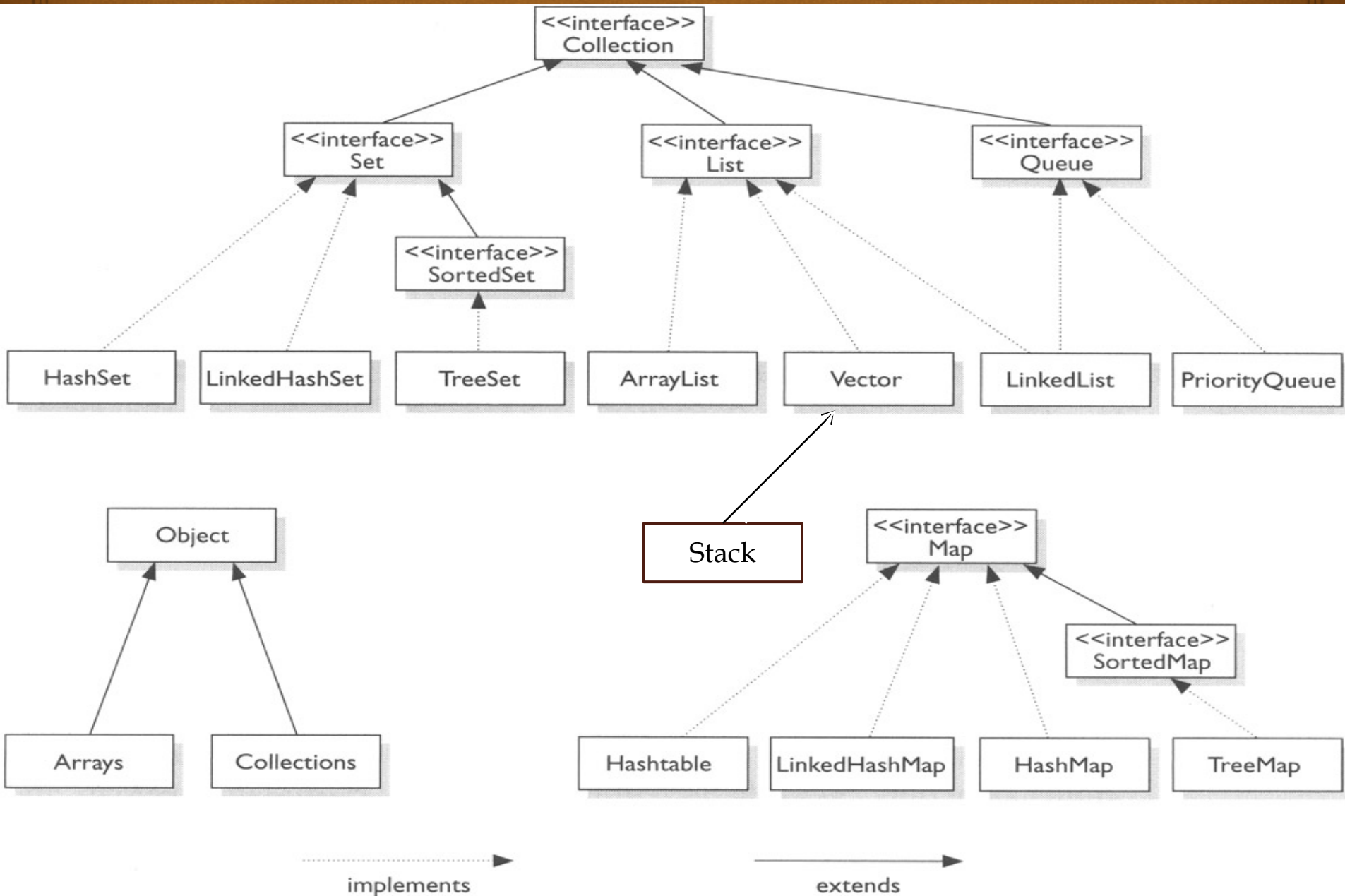
Lists are the way of collecting objects of the same type. Everything in creation is the same. The unity of everything is really the more powerful quality.



# DAY - 1



# Java Collections framework interface and class hierarchy



# List

A list is a popular data structure to store data in sequential order.

For example, a list of students, a list of available rooms, a list of cities, and a list of books, etc. can be stored using lists.

The common operations on a list are usually the following:

- Retrieve an element from this list.
- Insert a new element to this list.
- Delete an element from this list.
- Find how many elements are in this list.
- Find if an element is in this list.
- Find if this list is empty.

# List



There are two ways to implement a list.

❧ ArrayList

❧ LinkedList

# A Growable Array

- ❧ Arrays are data structures that provide "random access" to elements – to find the *i*th entry, there is no need to traverse the elements prior to the *i*th in order to locate the *i*th entry.
- ❧ Initially, an array, say data of `Object[]` type, is created with a specified size.
- ❧ When inserting a new element into the array, first ensure there is enough room in the array. If not, create a new array with the size as twice as the current one.
- ❧ Copy the elements from the current array to the new array. The new array now becomes the current array.

Demo : `MyStringList.java`



# ArrayList

## ArrayList

- It is a class in the standard Java libraries that can hold any type of object
- an object that can grow and shrink while your program is running (unlike arrays, which have a fixed length once they have been created)
- In general, an **ArrayList** serves the same purpose as an array.
- Insert and remove operations of a List
- Automatically enlarges array
- Allows insertion and removal of elements anywhere in the array



# Using the `ArrayList` Class

- ✧ In order to make use of the `ArrayList` class, it must first be imported

```
import java.util.ArrayList;
```

- ✧ An `ArrayList` is created and named in the same way as object of any class, except that you specify the base type as follows:

```
ArrayList<BaseType> aList =  
    new ArrayList<BaseType>( );
```

# Primitives

☞ Sun Java provides “wrapper” classes for all primitive types

☞ Allows them to be stored in a list

☞ int → Integer

☞ short → Short

☞ byte → Byte

☞ long → Long

☞ float → Float

☞ double → Double

☞ char → Character

☞ boolean → Boolean

# Primitives and Lists In J2SE5.0

☞ supports automatic conversion between primitives and wrappers; this is called *autoboxing*.

```
int[] ints = {1, 3, 4};  
List<Integer> list = new ArrayList<Integer>();  
for(int i = 0; i < ints.length; ++i) {  
    list.add(ints[i]);  
}
```

```
// no extraction of primitive necessary  
int x = list.get(1);
```

# Creating an ArrayList

- ✧ An initial capacity can be specified when creating an **ArrayList**
- ✧ The following code creates an **ArrayList** that stores objects of the base type **String** with an initial capacity of 20 items

```
ArrayList<String> list = new  
ArrayList<String>(20);
```

- ✧ Specifying an initial capacity does not limit the size to which an **ArrayList** can eventually grow
- ✧ Note that the base type of an ArrayList is specified as a *type parameter*
- ✧ *Initial Capacity of array list is 10. It's not a size.*



# Adding elements to an **ArrayList**

✧ The **add** method is used to add an element at the “end” of an **ArrayList**

```
public boolean add(Object o);  
list.add("something");
```

✧ The method name **add** is overloaded

**void add(int index, Object element)**

This version that allows an item to be added at any currently used index position or at the first unused position

# How many elements?

- ❧ The **size** method is used to find out how many indices already have elements in the **ArrayList**

```
int howMany = list.size();
```

- ❧ The **set** method is used to replace any existing element, and the **get** method is used to access the value of any existing element

```
list.set(index, "something else");  
String thing = list.get(index);
```

- ❧ **size** is NOT capacity

- ❧ size is the number of elements currently stored in the ArrayList

- ❧ Capacity is the maximum number of elements which can be stored. Capacity will automatically increase as needed

# ArrayList code Example

```
// Note the use of Integer, rather than int
public static void main( String[ ] args)
{
    ArrayList<Integer> myInts = new ArrayList<Integer>(25);
    System.out.println( "Size of myInts = " + myInts.size());
    for (int k = 0; k < 10; k++)
        myInts.add( 3 * k );
    myInts.set( 6, 44 );
    System.out.println( "Size of myInts = " + myInts.size());
    for (int k = 0; k < myInts.size(); k++)
        System.out.print( myInts.get( k ) + ", " );
}

// output
Size of myInts = 0
Size of myInts = 10
0, 3, 6, 9, 12, 15, 44, 21, 24, 27
```

# Methods in the Class **ArrayList with Example**

Lesson-8-ListMethods.doc



# ArrayList Inefficiencies



- ❧ If in using an Array List, the operations remove, insert, and add are used predominantly, performance is not optimal because of the repeated resizing and other array copying that are needed. For such purposes, another implementation of "List" is necessary.
- ❧ "List" is known as an *abstract data type* (ADT) – consisting of a sequence of objects and operations on them.

# More Array Operations

- ❧ Sorting (Minsort)
  - ❧ FindMax(), FindMin()
- ❧ Searching a sorted array
  - ❧ Linear Search
  - ❧ Binary Search

# MinSort



- ❧ *MinSort* uses the following approach to perform sorting an array *A* of integers.
  - ❧ Start by creating a new array *B* that will hold the final sorted values
  - ❧ Find the minimum value in *A*, remove it from *A*, and place it in position 0 in *B*.
  - ❧ Place the minimum value of the remaining elements of *A* in position 1 in array *B*.
  - ❧ Continue placing the minimum value of the remaining elements of *A* in the next available position in *B* until *A* is empty.

# Binary Search on a Sorted Array

- ❧ Calculate the middle position of the array
  - ❧ `mid = arr.length/2;` (Binary Search)
- ❧ `if (target < arr[mid])`
  - search the lower half of the array
  - else search the upper half of the array
- ❧ Repeatedly cut the search domain in half until the target value is located (or it is known that the target is not in the array)
- ❧ From Collections we can use predefined method *`binarySearch()`*. Here *`stringArray`* is a collection and *"Java"* is the key value.
  - ❧ `int x = Collections.binarySearch(stringArray, "Java");`
  - ❧ `System.out.println("String \"Java\" is in the position of: " + x);`



# Iterator

- ❧ It enables you to cycle through a collection, obtaining or removing elements.
- ❧ An interface in Java with three methods
  - ❧ `hasNext()`
  - ❧ `next()`
  - ❧ `remove()`
- ❧ For ArrayLists, any approach to iterating through elements is ok,
- ❧ but for LinkedLists, the `get(int pos)` operation is very slow, so the Iterator or for each approach is preferable

# Iterable and Iterator interface



```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

# Iterators

Java's lists can be traversed as :

## 1. Loop through the list

```
Object next = null;
for(int i = 0; i < list.size(); ++i) {
    next = list.get(i);
    //do something with next
}
```

## 2. Use an Iterator

```
Object next2 = null;
ArrayList<String> list = new ArrayList<String>(
    Arrays.asList("Hello", "Welcome", "Java", "Object", "Array",
"String", "Inheritance"));
Iterator it = list.iterator();
while(it.hasNext()) {
    nextitem = it.next();
    System.out.println(nextitem);
}
```

## The Iterable Interface and “for each” Loops

New in Java 8: A default method `forEach` was added to the `Iterable` interface. Consequently, any Java library class that implements `Iterable`, as well as any user-defined class that implements `Iterable`, has automatic access to this new method.

The `forEach` method takes a lambda expression of the form `x -> function(x)` where `function(x)` does not return a value, like `System.out.println(x)`.

```
List<String> javaList = new ArrayList<>();  
javaList.add("Bob");  
javaList.add("Carol");  
javaList.add("Steve");  
  
javaList.forEach( name -> System.out.println(name));
```

//output

Bob

Carol

Steve



# Demo code



- ❧ Array.java
- ❧ Array1.java
- ❧ ArrayListDemo.java
- ❧ TallySale.java

# Comparable and Comparator Interface



# Comparing Objects for Sorting and Searching

1. Java supports sorting of many types of objects. To sort a list of objects, it is necessary to have some “ordering” on the objects. For example, there is a natural ordering on numbers and on Strings. But what about a list of Employee objects?
2. In practice, we may want to sort business objects in different ways. An Employee list could be sorted by name, salary or hire date.

Employee Class		
Name	Hire Date	Salary
Joe Smith	11/23/2000	50000
Susan Randolph	2/14/2002	60000
Ronald Richards	1/1/2005	70000



3. To accomplish this, you specify your own ordering on a class using the **Comparator** interface, whose only method is **compare()**.



4. The **compare()** method is expected to behave in the following way (so it can be used in conjunction with the Collections API):

For objects *a* and *b*,

- ❧ **compare(a,b)** returns a negative number if *a* is “less than” *b*
- ❧ **compare(a,b)** returns a positive number if *a* is “greater than” *b*
- ❧ **compare(a,b)** returns 0 if *a* “equals” *b*



5. If compare is not used in a “sensible” way, it will lead to unexpected results when used by utilities like Collections.sort.

The compare contract It must be true that:

---

a is “less than” b if and only if b is “greater than” a  
if a is “less than” b and b is “less than” c, then a must be “less than” c.

It *should* also be true that the Comparator is *consistent with equals*; in other words:

`compare(a,b) == 0` if and only if `a.equals(b)`

If a Comparator is not consistent with equals, problems can arise when using different container classes. For instance, the `contains` method of a Java List uses `equals` to decide if an object is in a list. However, containers that maintain the order relationship among elements check whether the output of `compare` is 0 to implement `contains`.

❧ The **Comparable** interface defines the **compareTo** method for comparing objects.



```
package java.lang;  
  
public interface Comparable<E>{  
    public int compareTo(E o);  
}
```

❧ The **compareTo** method determines the order of this object with the specified object **o** and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than object **o**.

# Comparator Interface

❧ **Comparator** can be used to compare the objects of a class that doesn't implement **Comparable**.



❧ To do so, define a class that implements the **java.util.Comparator<T>** interface.

❧ The **Comparator<T>** interface has one method.

**public int** compare(T element1, T element2);

Returns a negative value if **element1** is less than **element2**,  
a positive value if **element1** is greater than **element2**,  
and zero if they are equal.



To know about :

## Comparable and Comparator Interface

Refer the demo code

ComparableDemo.java ( Comparable Interface)

ArrayListSort.java(Comparator Interface)

lesson8comparator.employee package

lesson8comparator.lambda package





# DAY - 2



*Best Practice* Different kinds of lists provide different advantages. LinkedLists are a superior choice when many inserts and deletions are expected, or when the number of add operations would force too many `resize()` operations in an ArrayList. If the requirement is instead for repeated access by index, ArrayList is preferable.

# Linked List

❧ Motivation:

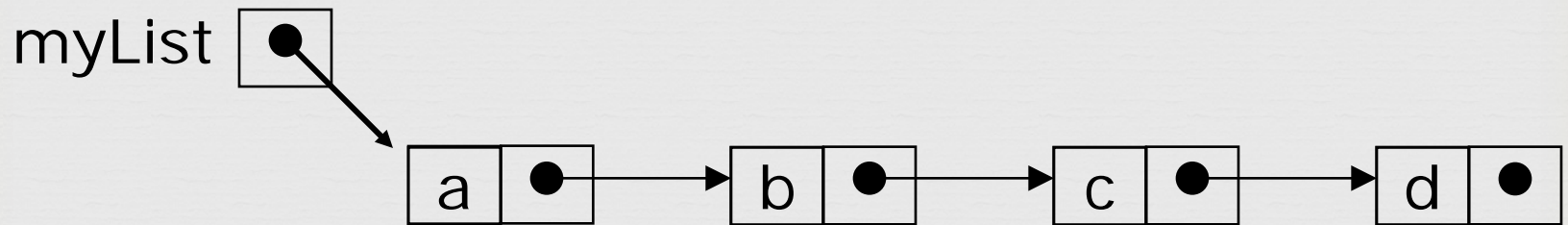


- ❧ The other approach is to use a linked structure. A linked structure consists of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list.
- ❧ eliminate the need to resize the array
- ❧ grows and shrinks exactly when necessary
- ❧ efficient handling of insertion or removal from the middle of the data structure
- ❧ random access is not often needed

# Anatomy of a linked list

⌘ A linked list consists of:

⌘ A sequence of **nodes**



Each node contains a **value**  
and a **link** (pointer or reference) to some other node

The last node contains a **null link**

The list may have a **header**



# More terminology



⌘ A node's **successor** is the next node in the sequence

⌘ The last node has no successor

⌘ A node's **predecessor** is the previous node in the sequence

⌘ The first node has no predecessor

⌘ A list's **length** is the number of elements in it

⌘ A list may be **empty** (contain no elements)

# Types of Linked List

- ❧ **Singly Linked List** : Each element of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the **head** of the list.
- ❧ **Doubly Linked List** : has two references, one to the next node and another to previous node.
- ❧ **Circular linked list**: A linked list whose last node has reference to the first node.

# Example

## Creation of Linked List

---

```
LinkedList <Integer>list = new  
                                LinkedList<Integer>();  
list.add(10);  
list.add(20);  
int size = list.size();
```

Lesson8-ListMethods.doc

# Implementation

Two ways

1. Java Collections – Linked List
2. Create by your own without Collection



# User Defined Linked List



# Nodes



- A linked list is composed of nodes linked together; each node contains the data
- find and get – traverse the nodes
- insert – inserts a new node by changing the links
- remove – removes a node by changing the links

```
public class MyObjectLinkedList {  
    Node header;
```

```
    public void insert(Object element, int pos){
```

```
        ...
```

```
    }
```

```
    public void remove(int pos){
```

```
        ...
```

```
    }
```

```
    public Object get(int pos){
```

```
        ...
```

```
    }
```


```
    public int find(Object element){
```

```
        ...
```

```
    }
```

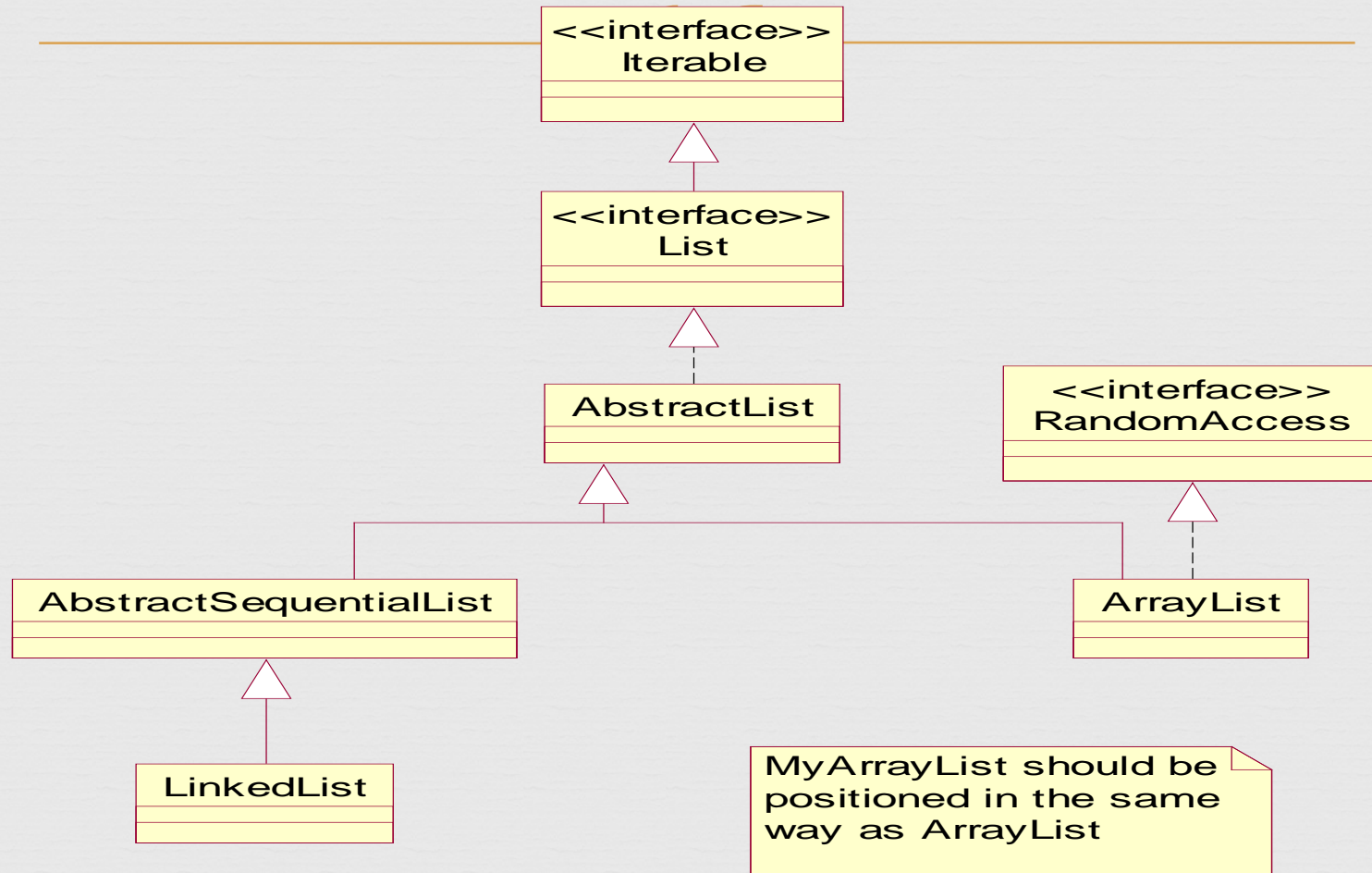


```
class Node {  
    Object value;  
    Node next;  
    Node previous;  
    Node(Node next, Node previous, Object value){  
        this.next = next;  
        this.previous = previous;  
        this.value = value;  
    }  
}
```





Linked lists is lack in *random access*



# Demo code



- ❧ LinkedListExample.java
- ❧ MailList.java
- ❧ MyStringLinkedList.java

# Main Point



Linked lists are much more efficient than arrays when many insertions or deletions need to be made to random parts of the list. Nature always functions with maximum efficiency and minimum effort.

# *Good Programming Practice*

---

Different kinds of lists provide different advantages.

LinkedLists are a superior choice when many inserts, deletions are expected, or when the number of add operations would force to many `resize()` operations in an `ArrayList`.

If the requirement is instead for repeated access by index, `ArrayList` is preferable.



# Main Points



An Array List encapsulates the random access behavior of arrays, and incorporates automatic resizing and optionally may include support for sorting and searching. Using a style of sequential access instead, Linked Lists improve performance of insertions and deletions, but at the cost of fast element access by index.

Random and sequential access provide analogies for forms of gaining knowledge. Knowledge by way of the intellect is always sequential, requiring steps of logic to arrive at an item of knowledge. Knowing by intuition, or by way of *ritam-bhara pragya*, is knowing the truth without steps – a kind of “random access” mode of gaining knowledge.