Lesson 8
The List Data Structure:

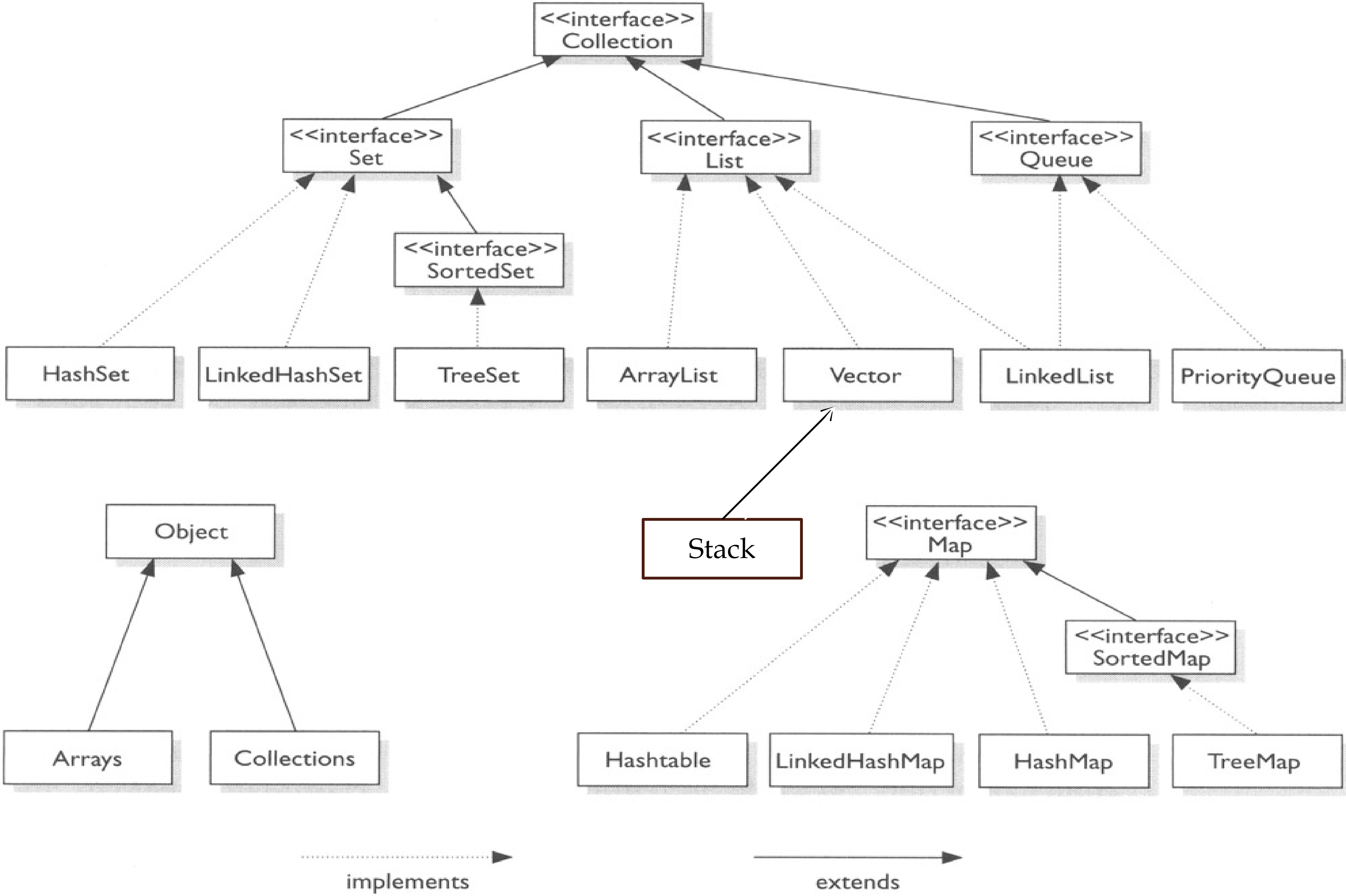
Sequential Unfoldment of Natural Law

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

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)

The LIST Abstract Data Type

- "List" is known as an *abstract data type* (ADT) – consisting of a sequence of objects and operations on them.

- Typical operations:

find(Object o)	returns position of first occurrence
findKth(int pos)	returns element based on index
insert(Object o, int pos)	inserts object into specified position
remove(Object o)	removes object
printList()	outputs all elements
makeEmpty()	empties the List

- Other operations are sometimes included, like "contains".

- Can be implemented in more than one way. Array List is one such implementation.

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.

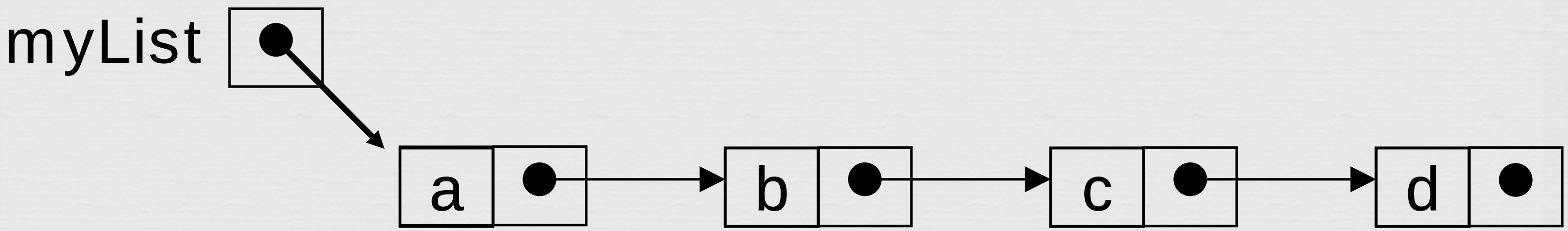
Best Practice Different kinds of lists provide different advantages. LinkedLists are a superior choice when many inserts and deletions are expected, or when the number of add operations would force too many resize() operations in an ArrayList. If the requirement is instead for repeated access by index,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

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;
	}
}
```

Day - 2

Searching and Sorting Lists

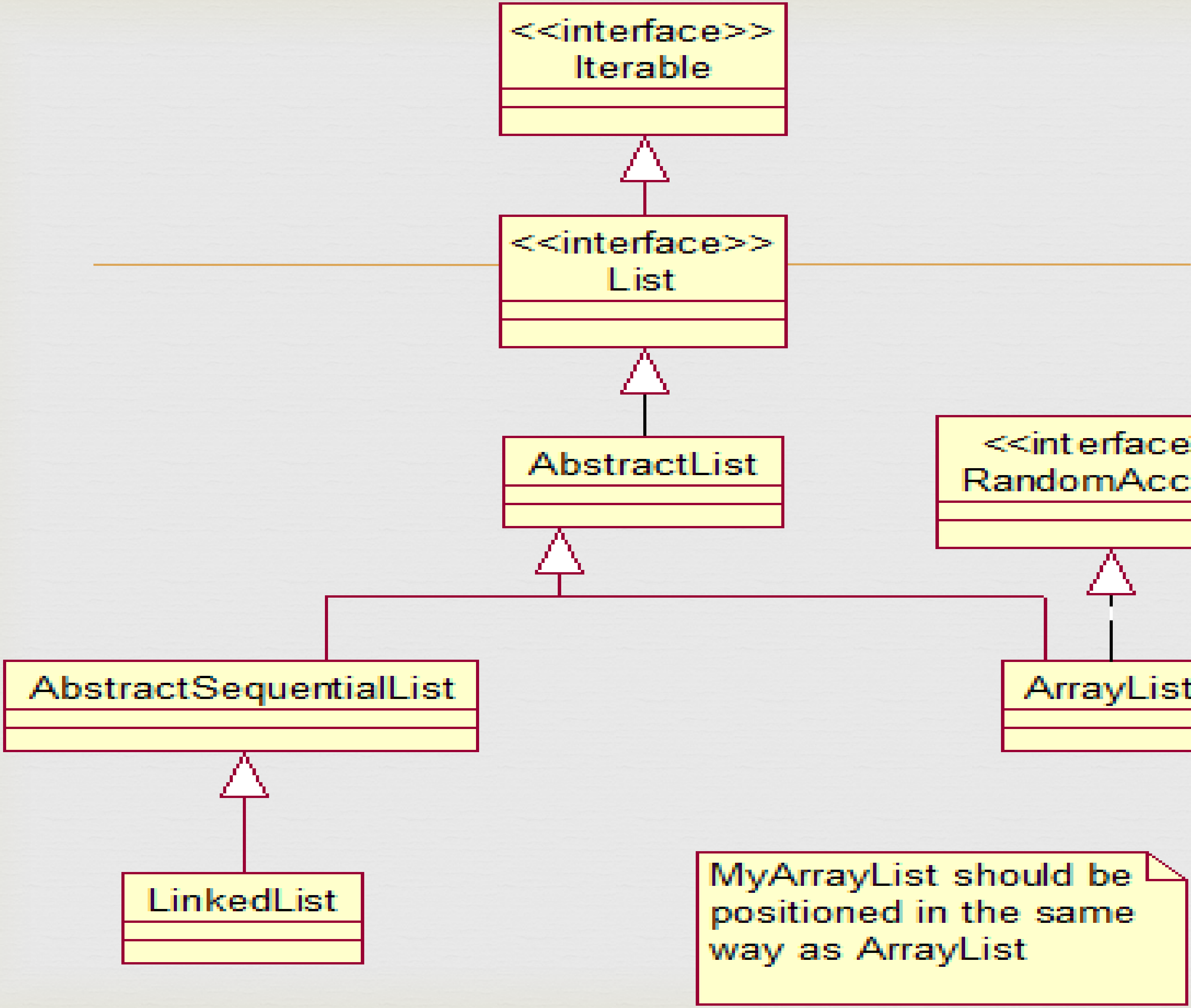
- Java provides `sort` and `binarySearch` methods for all of its lists (and other types of collections), by way of the `Collections` class.

```
List<String> myList = new ArrayList<String>();
//populate it with a long list of first names, and then….
Collections.sort(myList);
int pos = Collections.binarySearch(myList,"Dave");
```

- As you will see in the labs for this lesson, sorting and searching are accomplished in different ways for different lists. It is possible to rewrite `MinSort` in the context of linked lists so that it is approximately as efficient as the `MinSort` for array lists. However, this is not true for binary search. Any known binary search implementation on linked lists is no more efficient than just doing a "find" operation. [This fact is part of the motivation for the invention of Binary Search Trees, which we will discuss later.]

- The reason is that linked lists lack *random access*, so finding the value in the middle of the list is a costly operation. For this reason, Sun's `ArrayList` implements a "tag or marker interface" `RandomAccess`.
- Then, when you call the `binarySearch` method on `Collections` for an `ArrayList`, the method recognizes that the list implements `RandomAccess`, and therefore uses a divide and conquer algorithm. But if you pass in a `LinkedList`, a slower algorithm is usually used (since no faster algorithm exists).

- If you want to use the `Collections` `binarySearch` method on your own array-based list, your list must implement the `List` interface, and, to ensure that the `binarySearch` implementation is efficient, it must also implement the `RandomAccess` interface.


<<interface>>
Iterable
<<interface>>
List
AbstractList
<<interface>>
RandomAccess
AbstractSequentialList
ArrayList
LinkedList
MyArrayList should be positioned in the same way as ArrayList

Comparing Objects for Sorting and Searching

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

Sorting

- To accomplish this, you specify your own ordering on a class using the **Comparator** interface, whose only method is **compare()**. Like lists, in j2se5.0, `Comparators` are parameterized.

- 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

- 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 (like `TreeSet` – more on this one later) check whether the output of compare is 0 to implement `contains`.

```
// Assumes Employee contains just name and hireDate as
// instance variables
public class NameComparator implements Comparator<Employee> {
     //is this implementation consistent with equals?
     public int compare(Employee e1, Employee e2) {
             return e1.getName().compareTo(e2.getName());
     }
}
public class EmployeeSort {
     public static void main(String[] args) {
             new EmployeeSort();
     }
  public EmployeeSort() {
     Employee[] empArray =
             {new Employee("George", 1996,11,5),
              new Employee("Dave", 2000, 1, 3),
              new Employee("Richard", 2001, 2, 7)};
     List<Employee> empList = Arrays.asList(empArray);
     Comparator<Employee> nameComp =
                             new NameComparator();
     Collections.sort(empList, nameComp);
     System.out.println(empList);
  }
}
public class Employee {
     private String firstName;
     private Date hireDate;
     // . . .
}
```

Question: How can the comparator be made consistent with equals?

Solution:

```
public class NameComparator implements
                      Comparator<Employee> {
    // consistent with equals
    public int compare(Employee e1, Employee e2) {
          String name1 = e1.getName();
          String name2 = e2.getName();
          Date hireDate1 = e1.getHireDay();
          Date hireDate2 = e2.getHireDay();
          if(name1.compareTo(name2) != 0) {
                return name1.compareTo(name2);
          }
          //in this case, name1.equals(name2) is true
          return hireDate1.compareTo(hireDate2);
    }
}
```

Genericizing the Objects Stored in a List

- One difficulty with our examples of Lists – MyStringList and MyStringLinkedList – is that they don’t work if the objects we wish to store are not Strings.

- *Unsatisfactory Solution:* Rewrite the List code for each type as the need arises. E.g. MyEmployeeList, MyIntegerList, MyAccountList. . .

- *A Better Solution:* Could create a List that stores elements of type Object.

Example: MyObjectList

```
public class MyObjectList {
        private final int INITIAL_LENGTH = 4;
        private Object[] objArray;
        private int size;

        public MyObjectList() {
                objArray = new Object[INITIAL_LENGTH];
                size = 0;
        }
        public void add(Object ob){
                if(size == objArray.length) resize();
                objArray[size++] = s;
        }
        . . .
}

//USAGE
MyObjectList list = new MyObjectList();
list.add("Bob");
list.add("Sally");
String name = (String)list.get(1);
```

Drawback : Once retrieve a element with specific type, always casting is required.
Pre Java-5 uses object type to support generic version in its API, but also have the same drawback

Usage:

```
ArrayList list = new ArrayList();
list.add("Bob");
list.add("Sally");
String name = (String)list.get(1);
```

Generics Support

- From j2se5.0 on, Lists include a generic parameter. Here are declarations from the Java library:

```
class ArrayList<E> implements List<E> {
    ArrayList<E>() {
            …
    }
}

class LinkedList<E> implements List<E> {
    LinkedList<E>() {
            …
    }
}

interface List<E> {
    void add(E ob);
    E get(int pos);
    boolean remove(E ob);
    int size();

    . . .

}
```

Generics Support

- `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

- List API in Java are designed to aggregate *objects,* not primitives.
- Java provides "wrapper" classes for all primitive types that help in this case.
- 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

- 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);
```

Inferred Types in JSE 7 and After:

When creating an instance of a parametrized type, the parameter can be dropped in the construction step:

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

is the same as:

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

//Examples

```
List<String> list = new ArrayList<String>();
list.add("Bob");
list.add("Sally");
String name = list.get(0); //no downcast required

//iterate using for each construct – no downcasting needed
for(String s : list) {
    //do something with s
}

//any class type can be used as a parameter
List<Employee> empList = new LinkedList<Employee>();
empList.add(new Employee("Bob", 40000, 1996, 12, 2));
empList.add(new Employee("Dave", 50000, 2000, 11, 15));

//clumsy runtime exceptions are now replaced by
    //compiler errors
 List<Integer> list = new ArrayList<Integer>();
 list.add(new Integer(1));
 list.add(new Integer(3));
 //list.add("5"); //compiler won't allow this
 Integer[] listArr =
           (Integer[])list.toArray(new Integer[3]);

 System.out.println(Arrays.toString(listArr));
```

Example

- **Creation of Linked List[API]**

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

Lesson8-ListMethods.doc

Good Programming Practice

- ***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.

  Sometimes you won't know which type of list will be the best choice. And even if you have a preference at the beginning of a project, the need may change as development proceeds. For these reasons, the best way to create a list is to use the principle of *Programming to the Interface:*

```
//start with an ArrayList
List myList = new ArrayList();
myList.add(“Bob”);
myList.add(“Dave”);
```

  Later, if you need to switch to a LinkedList, only one change in the code is necessary:

```
List myList = new LinkedList();
myList.add(“Bob”);
myList.add(“Dave”);
```

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();
}
```

Refer : MyStringListWithIterator for the own implementation of Iterator

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

Refer : Lesson-8-ListMethods.doc to know more about List API methods and examples

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(mind full of Rhythm) , is knowing the truth without steps – a kind of "random access" mode of gaining knowledge.