
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS390 Foundamental Programming Practices (FPP)
Professor Paul Corazza

Lecture 8:
The List Data Structure

Wholeness of the Lesson

The List ADT is one of the most general data types, capable of supporting most needs for storing a collection of objects in memory. Different implementations of this data type provide optimizations for different operations – such as insert, delete, find – that are typically supported by Lists. Lists give expression to the natural tendency of pure intelligence to express itself through a sequential unfoldment.

Outline of Topics

- Array Lists and including sort and search
- The LIST ADT and list operations
- Linked Lists – singly linked, headers, doubly linked, circular linked
- Generic types for lists before and after jse5.0
- Using Iterators instead of for loops
- The List interface in Java and the AbstractList class
- The "for each" construct and the `forEach` method
- `Collections.sort`, `Collections.binarySearch`, and `RandomAccess`
- `Comparators`
- Synchronizing List data

A Growable Array

- *Arrays Are Very Efficient* 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.

- *Arrays Inconvenient Because of Fixed Length.* Arrays are inconvenient sometimes because it is necessary to commit to a fixed array size before adding elements. If the number of elements then exceeds the array size, a new larger array must be created to accommodate the new elements, and old elements have to be copied into the new array. There are similar problems involved in removing elements and in inserting elements into a specified position.

- *ArrayList.* A convenient data structure that saves the explicit effort of recopying. Here, all the work required to copy over elements into a new array for insert, remove, and adding operations is encapsulated in the class.

- Example: `MyStringList` in `lesson8.demo.mystringlist`

Array Operations Can Be Included in An Array List's Set of Methods

- We consider two operations: *sorting* and *searching a sorted array*
- There are many sorting algorithms; Java provides a sorting routine as part of its API. We will consider a simple one for illustration.
- `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.
  - [Recall Lab4-2]

MinSort for Arrays

- *In-Place MinSort*. MinSort can be implemented without an auxiliary array. This is done by performing a swap after each min value is found. Here is the code:

```
//arr is given as input
int[] arr;
public void sort(){
    if(arr == null || arr.length <=1) return;
    int len = arr.length;
    for(int i = 0; i < len; ++i){
        //find position of min value from arr[i] to arr[len-1]
        int nextMinPos = minpos(i,len-1);

        //place this min value at position i
        swap(i, nextMinPos);
    }
}


void swap(int i, int j){
     String temp = strArray[i];
     strArray[i] = strArray[j];
     strArray[j] = temp;
 }
```

(continued)

Exercise: Include a version of `MinSort` in `MyStringList`. Since `Strings` will be compared instead of `int's`, you will need to use the `compareTo` method (Lesson 7)

Searching a Sorted Array with Binary Search

- If an array `arr` of integers is already sorted, we can search for a given integer `testVal` in a very efficient way using the binary search strategy described in Lab 4-3:

  Let `mid = arr[arr.length/2]` (the value in the middle position of the array).

  - If `testVal == mid`, return true
  - Else if `testVal < mid`, search for `testVal` in the left half of the array
  - Else if `testVal > mid`, search for `testVal` in the right half of the array

  The strategy of repeatedly cutting the size of the search domain by a factor of 2 makes this algorithm highly efficient. It does NOT work if the array is not already sorted.

Here is the code for binary search, applied to sorted arrays.

```
int[] anArray;  //given as input and set as instance variable
public boolean search(int val) {
   boolean b = recSearch(0, anArray.length-1, val);
   return b;
}

private boolean recSearch(0 a, int b, int val) {
   int mid = (a + b)/2;
   if(anArray[mid] == val) return true;
   if(a > b) return false;
   if(val > anArray[mid]) {
      return recSearch(mid + 1, b, val);
   }
   return recurse(a,mid - 1,val);
}
```

- Exercise: Implement a version of binary search in `MyStringList`.

  Hint: You will replace == with `equals` and < with `compareTo` when working with Strings

Inefficiencies of Array List `insert, add, remove` Operations

- If in using an Array List, the operations `remove, insert`, and `add` are used predominantly, performance is not optimal because of repeated resizing and other steps that require array copying. For such purposes, another implementation of "List" is better.

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:

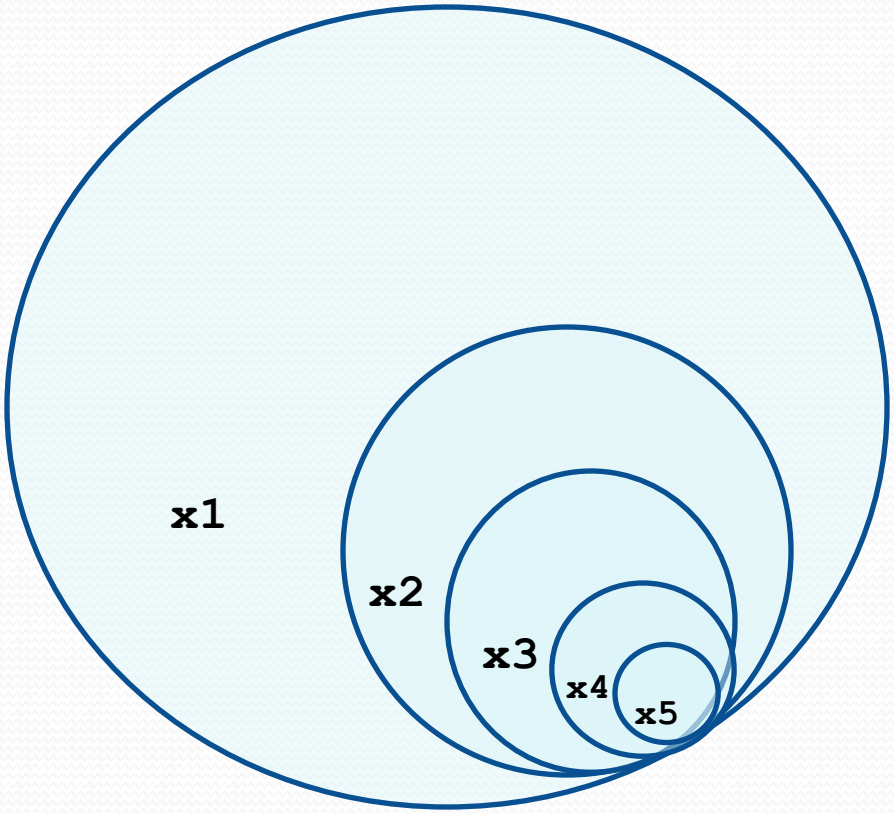
find(Object o)	returns position of first occurrence
findKth(int pos)	returns element based on index
insert(Object o, int pos)	inserts object into specified position
remove(Object o)	removes object
printList()	outputs all elements
makeEmpty()	empties the List

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

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

`LinkedList` Implementation of LIST: Concept of a NODE

- Can store data values in nested nodes rather than by index in an array-based structure



rather than. . .

Nodes

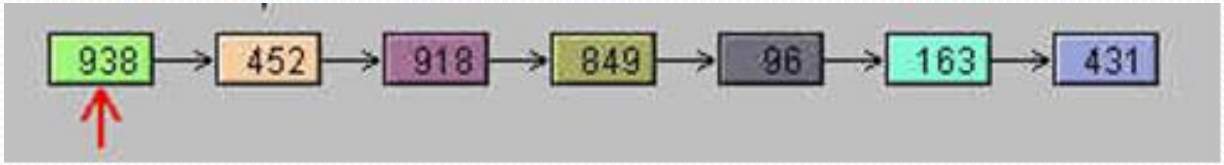
```
public class Node {
    String data;
    Node node;

    @Override
    public String toString() {
        return data + " ";
    }
}
```

Performing a search on a Node populated with values

```
Node startNode = null;
boolean search(String s) {
    if(s == null) return false;
    Node next = startNode;
    while(next != null) {
        String t = next.data;
        if(s.equals(t)) {
            return true;
        }
        next = next.node;
    }
    return false;
}
```

`LinkedList` Implementation of List

- The Need: Improve performance of *insert, remove, add,* and avoid the cost of resizing incurred by the array implementation.



- A `LinkedList` consists of Nodes. In addition to storing data, a Node contains a link to the next Node (which may be null).

- ***Operations*** – Placed *inside* the Linked List instead of being executed from an external class.
  - *search* requires traversing the Nodes via links, starting at the first Node (as in previous slide).
  - *insert* requires traversing the Nodes to locate position and adjusting links
  - *remove* requires doing a *find*, and when the object is found, the *previous* object has to be located so that it can be linked to the *next* object

Implementing *remove* Operation

- ***Finding the previous node during remove operation.***
  - Could invoke a routine to go back to the beginning and locate the previous Node
  - *remove* method could maintain a reference to previous Node
  - Could implement as a doubly linked list, where previous Node as well as next Node are stored in the Node

```
void removeNode(String s) {
    if(s == null) return;
    if(startNode != null && startNode.data.equals(s)){
        startNode = startNode.node;
        return;
    }
    Node previous = startNode;
    Node next = startNode.node;
    while(next != null) {
        if(s.equals(next.data)) {
            previous.node = next.node;
            return;
        }
        previous = next;
        next = next.node;
    }
}
```

`remove` in a Singly Linked List

Linked Lists with Headers

Headers.

- A header is a Node that contains no data, can never be removed, and has a link to first Node.
- Sometimes used to make `remove` operation uniform for all Nodes (removing first Node no longer a special case)
- In a doubly linked list, header's previous Node is always null.

```
Node header = null; //contains no data, cannot be removed

void removeNode(String s) {
    if(s == null) return;
    Node next = header.node;
    Node previous = header;

    //No special case for removing first node
    while(next != null) {
        if(s.equals(next.data)) {
            previous.node = next.node;
            return;
        }
        previous = next;
        next = next.node;
    }
}
```

`remove` in a Singly Linked List with header

Doubly Linked List with Header

```
public class MyStringLinkedList {
  Node header;
  MyStringLinkedList(){
      header = new Node(null,null, null);
  }
  //adds to the front of the list
  public void add(String item){
      Node n = new Node(header.next, header, item);
      if(header.next != null){
          header.next.previous = n;
      }
      header.next = n;

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

Question: How to implement the `remove` operation?

- See Demo (`lesson8.demo.DemoLinkedList`) and the lab on `MyStringLinkedList` (prog8-2)

Circular Linked Lists

- In a circular LinkedList, the last element has a link to the first
- If a header is used, the last element links to the header
- If the LinkedList is doubly linked, and has a header, `header.previous` points to the last element as well
- Making a doubly linked list circular cuts the search time for the operations `insert(Object o, int pos)` and `findKth` in half.

Main Point

The List ADT captures the abstract notion of a "list"; it specifies certain operations that any kind of list should support (for example, *find*, *findKth*, *insert*, *remove*), without specifying the details of implementation.

Different concrete implementations of this abstract data type (such as Array Lists and Linked Lists) meet the contract of the List ADT using different implementation strategies. Likewise, pure awareness is an abstraction of individual awareness; each individual provides a specific, concrete realization of pure consciousness.

Genericising 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); //downcast necessary
```

Example: MyObjectLinkedList

```
public class MyObjectLinkedList {
  Node header;
  MyObjectLinkedList (){
      header = new Node(null,null, null);
  }
  public void add(Object item){
      Node n = new Node(header.next,header,item);
      if(header.next != null){
          header.next.previous = n;
      }
      header.next = n;

  }
      . . .

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

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

Java's Approach (before jdk 1.5)

- Before j2se5.0, Java provided versions of these two kinds of Lists having implementations similar to the above.

- *ArrayList*. This is an array-backed list that accepts any type of object, like MyObjectList above.

  Usage:

```
ArrayList list = new ArrayList();
list.add("Bob");
list.add("Sally");

String name = (String)list.get(1);
```

- *LinkedList*. This is a linked list that accepts any type of object, like MyObjectLinkedList above. Usage is like that of an ArrayList; however, as in our earlier examples, LinkedList performs better for insert and delete operations, and never needs to be resized.

  Usage:

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

Using Java's Lists With Primitives

- Lists in Java are designed to aggregate *objects*, not primitives.
- Java provides "wrapper" classes for all the primitives

primitive	wrapper class
`int`	`Integer`
`short`	`Short`
`byte`	`Byte`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

- Autoboxing allows you to use lists with primitives transparently

```
List list = new ArrayList();
list.add(5);  //5 converted to Integer type
```

Iterating Through Elements in a List (pre-j2se5.0)

- Java's lists can be traversed in one of two ways:

```
//mimic loop through an array
String next = null;
for(int i = 0; i < list.size(); ++i) {
    next = (String)list.get(i);
    //do something with next
}

//use an Iterator
String next = null;
Iterator iterator = list.iterator();
while(iterator.hasNext()) {
    next = (String)iterator.next();
    //do something
}
```

- `Iterator` is a Java interface with two methods: `hasNext()` and `next()`. Every type of list implemented in Java provides access to an `Iterator` instance. We can do the same for `MyStringList`:

```
class MyStringList implements Iterable {
      //. .   . .
  public Iterator iterator() {
      return new MyIterator();
  }
  private class MyIterator implements Iterator {
      private int position;
      MyIterator(){
         position = 0;
      }
      public boolean hasNext() {
         return (position < size);
      }

      public Object next() throws IndexOutOfBoundsException {
         if(!hasNext()) throw new IndexOutOfBoundsException();
         return strArray[position++];
      }

      public void reset() {
         position = 0;
      }

      /** optional -- not necessary to implement */
      public void remove() {
         // not implemented

      }

  }
```

```
public static void main(String[] args){
   MyStringList l = new MyStringList();
   l.add("Bob");
   l.add("Steve");
   l.add("Susan");
   l.add("Mark");
   l.add("Dave");
   Iterator iterator = l.iterator();
   while(iterator.hasNext()){
      System.out.println(iterator.next());
   }
}
```

See Demo – `lesson8.demo.MyStringList`

Prefer Iterators to `for` Loops

- For `ArrayLists`, either approach to iterating through elements is OK, but for `LinkedLists`, the

  `get(int pos)`

  operation is very slow, so the `Iterator` approach is preferable.

- All lists in Java implement the interface *List*. The declared operations are identical to the operations we find in `ArrayList` and `LinkedList` – here is a partial catalogue of them:

```
void add(Object ob);
Object get(int pos);
boolean remove(Object ob);
int size();
```

List ADT in the Java Library

- All lists in Java implement the interface *List*. The declared operations are identical to the operations in `ArrayList` and `LinkedList` – here is a partial catalogue:

```
void add(Object ob);
Object get(int pos);
boolean remove(Object ob);
int size();
```

- When creating your own implementation of `List`, instead of implementing all the methods in the `List` interface, you can use default implementations provided by the `AbstractList` class. This class requires only that you provide your own implementation of `get(int i)`. Other common methods (`add`, `remove`, `set`) usually need to be overridden (by default they throw an `UnsupportedOperationException`).

  A big advantage to using `AbstractList` as a superclass for your list implementations is that it provides a default implementation of `Iterator` and `ListIterator`.

Example: Extending `AbstractList`

```
//declare your list to extend AbstractList
public class MyStringList extends AbstractList { ...  }

public class Test {
     public static void main(String[] args){
            MyStringList l = new MyStringList();
            l.add("Bob");
            l.add("Steve");
            l.add("Susan");
            l.add("Mark");
            l.add("Dave");
            //uses the implementation in AbstractList
            Iterator iterator = l.iterator();
            while(iterator.hasNext()){
                  System.out.println(iterator.next());
            }
     }
}
```

Programming to the Interface

- Always type your lists as `List` (as implementers of the `List` interface)
  - Supports polymorphism
  - Adds flexibility to your implementation

Example: Start with `ArrayList`:

```
List myList = new ArrayList();
myList.add("Bob");
myList.add("Dave");
```

Later, decide to switch to `LinkedList`:

```
List myList = new LinkedList(); //one small change
myList.add("Bob");
myList.add("Dave");
```

To ensure optimal efficiency and flexibility, looping through a list should always be done with an Iterator (when using pre-j2se5.0 style of coding) (unless information about the indices is actually necessary)

```
//code works when ArrayList is replaced by LinkedList
//or any other implementor of the List interface
List myList = new ArrayList();
myList.add("Bob");
myList.add("Dave");
String next = null;
Iterator it = myList.iterator();
while(it.hasNext()) {
    next = (String)it.next();
}
```

Lists and Iteration in JSE5.0 and After

- To do away with the downcasting and support compiler type checking, the Java designers created *parametrized lists* in j2se5.0.

- An example of an undesirable aspect of old-style list (which parametrized lists fix) is the following:

```
public static void main(String[] args) {
    List list = new ArrayList();
    list.add("hello");
    list.add(1);

    Object[] ints = list.toArray();

    //No compiler warning for this
    //Produces runtime ClassCastException
    Integer x = (Integer)ints[0];
}
```

Runtime exception because there is no compiler checking of types in a collection.

- 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();

    . . .

}
```

```
//USAGE
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));
```

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>();
```

Warnings

- Rules for Java syntax forbid the creation (but not declaration) of an *array* of parametrized Lists:

```
//compiler error
List<String>[] arrayOfLists = new ArrayList<String>[10];

//Workaround: can use an ArrayList instead of an Array:
ArrayList<List<String>> listOfLists =
                              new ArrayList<List<String>>(10);

//The following works but produces a warning
List<String>[] arrayOfLists = new ArrayList[10];
```

- Subtypes of parametrized types may seem unexpected:

```
ArrayList<Manager> is a subtype of List<Manager>

ArrayList<Employee> is a subtype of List<Employee>

BUT
ArrayList<Manager> is NOT a subtype of List<Employee> or
even of ArrayList<Employee>
```

The `Iterable` Interface and "for each" Loops

- When you create your own type of list in Java, like `MyStringList`, you cannot use the "for each" construct without additional work. In order for this construct to be supported, your list class must implement the `Iterable` interface. This interface has just one method that must be implemented:

```
public Iterator iterator();
```

- Example:

```
class MyStringList implements Iterable {

        public Iterator iterator() {
                //see earlier code which implements Iterator
        }
        public static void main(String[] args){
                MyStringList list = new MyStringList();
                list add("Bob");
                list.add("Steve");
                list.add("Susan");
                list.add("Mark");
                list.add("Dave");
                //this works because Iterable has been implemented
                for(Object s : list){
                        System.out.println(s);
                }
        }
}
```

- Note that the `Iterable` interface is *automatically implemented* whenever your list class is a subclass of `AbstractList` (this class declares that it implements Iterable).

  Also, the `List` interface is declared to be a subinterface of `Iterable`, so if you declare your list class to be an implementer of `List`, it is required to implement `Iterable` interface (and no special declaration "implements Iterable" is needed).

- 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)` or `list.add(x)`.

- Examples:

```
//Java's List
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
```

```
//User-defined list that
//implements Iterable
MyStringList list = new
MyStringList();
list.add("Bob");
list. add("Carol");
list.add("Steve);

list.forEach(
     name ->
       System.out.println(name));

//output
Bob
Carol
Steve
```

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");
```

How Sorting Is Done in the Collections Class – the Role of `RandomAccess`

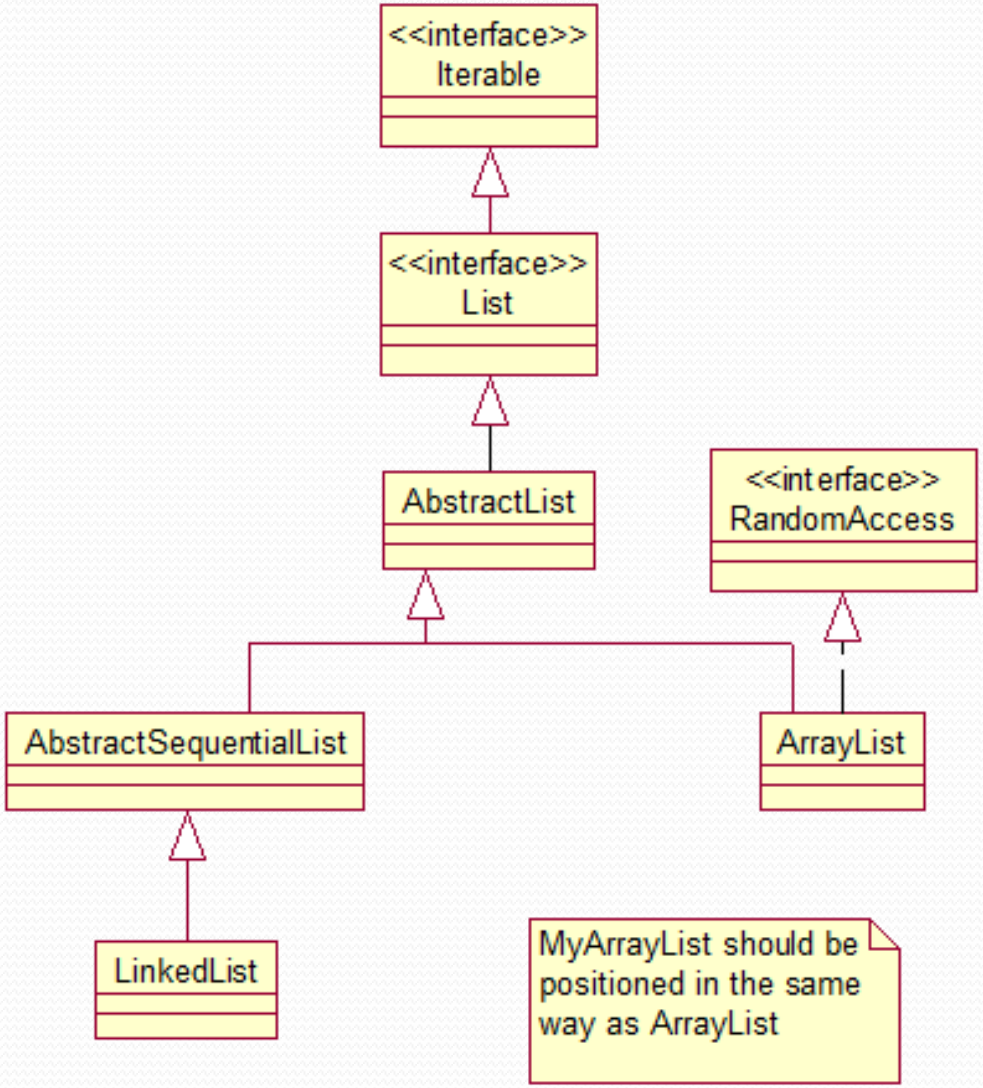
- *Arrays* can be sorted most efficiently because there is no cost for accessing array elements by index
- For collections in the Java Library that provide element access like arrays, the fastest sorting algorithms can be used. Such collections implement the `RandomAccess` interface.
- `RandomAccess` is a *tag interface* – it contains no methods. But when a collection implements `RandomAccess` and is passed to `Collections.sort`, the JVM passes the collection to a fast algorithm that relies on speedy element access.
- Lists that do not implement `RandomAccess` are sorted using another strategy that does not depend on the cost of element access.
- Example: `LinkedList`. Fast sorting algorithms in their very first step attempt to read the value contained near the middle of the collection. For a `LinkedList`, a traversal of half the list is therefore necessary just to get started. For an `ArrayList` (which implements `RandomAccess`), finding the middle element takes only one step.

How Searching Is Done in the Collections Class – the Role of `RandomAccess`

- Performing binary search on an array of sorted elements is extremely fast, but again depends on the fact that accessing elements by index is instantaneous
- The fast binary search algorithm is applied to collections that implement RandomAccess
- Since binary search also requires accessing the middle element in the very first step, when binary search is run on collections that do not implement RandomAccess, an ordinary left-to-right scan is performed.

Using `Collections` Library Functions with User-Defined Collections

- If you want to use `Collections.sort` or `Collections.binarySearch` on your own list:
  - Your list must implement the `List` interface (or extend `AbstractList`), since these `Collections` operations expect this.
  - If you want optimal performance of sorting and binary search, your list must also implement `RandomAccess`


<<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

- Implementing the Comparable interface allows you to sort a list of Employees with reference to one primary field – for instance, you could sort by name or by salary, but you do not have the option to change this primary field.

- A more flexible interface for such requirements is provided by the **Comparator** interface, whose only method is **compare()**. Like lists, in j2se5.0, `Comparators` are parametrized.

- The `compare(Object a, Object b)` 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 in the previous example 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);
   }
}
```

Keeping List Data Synchronized

- Coming soon…

Main Point

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 losing 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 (*prathibha*) , or by way of *ritam-bhara pragya*, is knowing the truth without steps – a kind of "random access" mode of gaining knowledge.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

All knowledge contained in point

1. An implementation of the abstract class AbstractSequentialList in Java (such as a LinkedList) results in a list that has only sequential access to its elements.
2. An implementation of the RandomAccess interface in Java (such as ArrayList and Vector) results in a list that has random access (and therefore, effectively, instantaneous access) to its elements.

---

3. **Transcendental Consciousness:** TC is the home of all knowledge. All knowledge has its basis in the unbounded field of pure consciousness.
4. **Wholeness moving within itself**: In Unity Consciousness, when the home of all knowledge has become fully integrated in all phases of life, it is possible to know anything, any particular thing, instantly.