Collections
List

Objectives

- At the end of this topic, you should be able to
  - Understand the Java Collection framework.
  - Explain List and its operations.
  - Use the concrete implementations of List in the collection .
  - Perform List Traversal.
  - Find index of element in a List

Collection

- A collection is a container object that groups multiple elements into a single unit
- Very useful as we can
  - store,
  - retrieve and
  - manipulate data

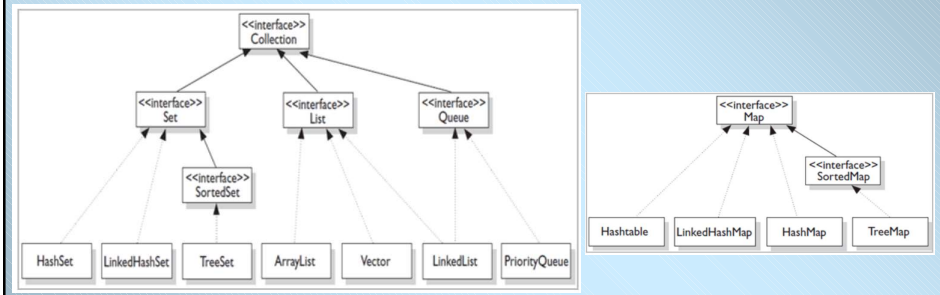
Java Collections Framework

- a unified architecture for representing and manipulating collections

- provides several types of collections, such as *sets, lists, queues* and *maps* in the form of interfaces and also their implementations in the form of concrete classes

Java Collections Framework hierarchy

List, Set and Queue are subinterfaces of Collection.



The Collection Interface

The Collection interface is the root interface for manipulating a collection of objects.

«interface» java.lang.Iterable<E>	
+iterator(): Iterator<E>	Returns an iterator for the elements in this collection.
+forEach(action: Consumer<? super E>): default void	Performs an action for each element in this iterator.

«interface» java.util.Collection<E>	
+add(e: E): boolean	Adds a new element e to this collection.
+addAll(c: Collection<? extends E>): boolean	Adds all the elements in the collection c to this collection.
+clear(): void	Removes all the elements from this collection.
+contains(o: Object): boolean	Returns true if this collection contains the element o.
+containsAll(c: Collection<?>): boolean	Returns true if this collection contains all the elements in c.
+equals(o: Object): boolean	
+isEmpty(): boolean	Returns true if this collection contains no elements.
+remove(o: Object): boolean	Removes the element o from this collection.
+removeAll(c: Collection<?>): boolean	Removes all the elements in c from this collection.
+retainAll(c: Collection<?>): boolean	Retains the elements that are both in c and in this collection.
+size(): int	Returns the number of elements in this collection.
+toArray(): Object[]	Returns an array of Object for the elements in this collection.
+toArray(a: T[]): T[]	Returns an array of the T[] type.

What is a List?

- A list is a popular data structure to store data
- Defined an ordered collection of elements.
- Can control the position where a new element is inserted in the list.
- Can use index to retrieve elements from the list.
- Allows duplicates.

Conceptual View of a List

Linear List

$L = (e_0, e_1, e_2, e_3, \ldots, e_{n-1})$

relationships

e_0 is the zeroth (or front) element

e_{n-1} is the last element

e_i immediately precedes e_{i+1}

List Operations

Useful set of operations:

- Add an element at a specified index
- Remove and return element with given index
- Determine list size
- Get element with given index
- Traversing the list
- Determine the index of an element

Add Element

Assume L = (a,b,c,d,e,f,g)

Add an element at a specified index

- add(0,h)
  - modified list L = (h,a,b,c,d,e,f,g)
  - index of a,b,c,d,e,f, and g increase by 1
- add(2,h)
  - modified list L = (a,b,h,c,d,e,f,g)
  - index of c,d,e,f, and g increase by 1
- add(10,h) = error
- add(-6,h) = error

Add Element

Assume L = (a,b,c,d,e,f,g)

Add new entry at the end

add(h)

modified list L = (a,b,c,d,e,f,g,h)

- No need to specify the index

Remove Element

Assume L = (a,b,c,d,e,f,g)

Remove and return element with given index

remove(2) returns c

and L becomes (a,b,d,e,f,g)

index of *d,e,f,* and *g* decrease by *1*

remove(-1) = error

remove(20) = error

List Size and Get Element

Assume L = (a,b,c,d,e)

Determine list size

size = 5

Get element with given index

get(0) = a
get(2) = c
get(4) = e
get(-1) = error
get(9) = error

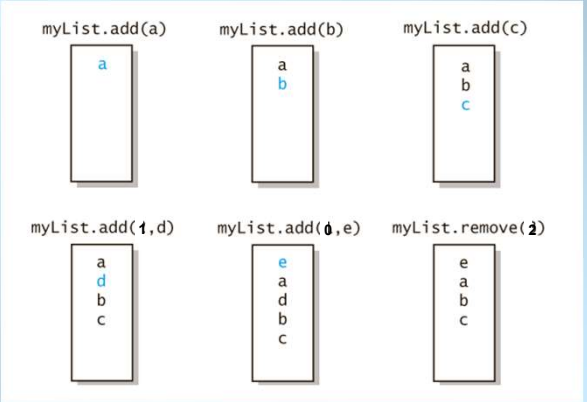
Find Index of List Element

Assume L = (a,b,d,b,a)

Determine the index of an element

indexOf(d) = 2
indexOf(a) = 0
indexOf(z) = -1
indexOf(b) = 1
lastIndexOf(b) = 3

Example



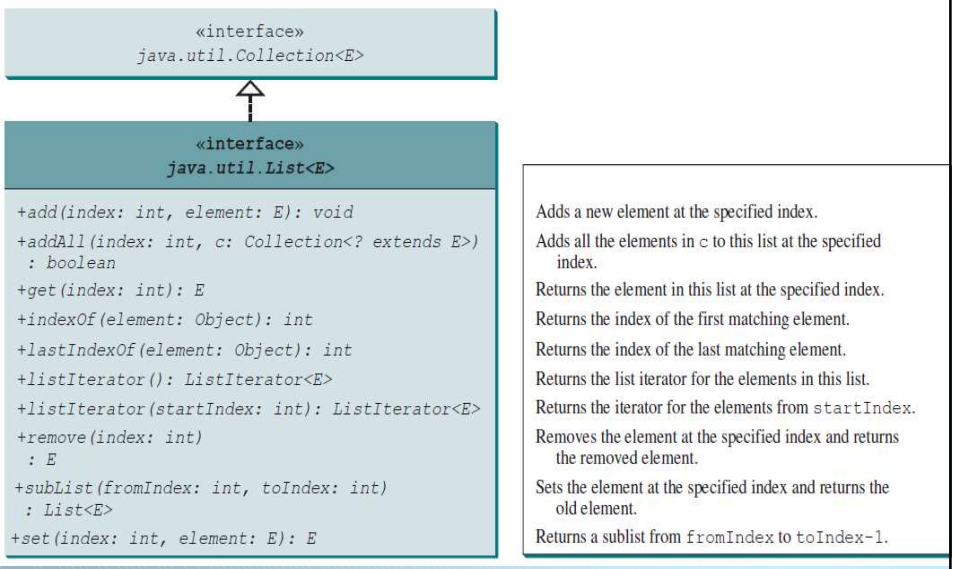
The effect of list *add(x),add(index,x), remove(index)* operations on an initially empty list

The List interface

The List interface adds position-oriented operations, It contains some methods such as:

```
boolean add(E element) //inherited
boolean add(int index, E element)
boolean remove(Object o) //inherited
E remove(int index)
int size()                //inherited
E get(int index)
int indexOf(Object o)
int lastIndexOf(Object o)
```

The List Interface



List Implementations

Several concrete classes implement the **`List`** interface such as:

- **`ArrayList`**
- **`Vector`**
- **`LinkedList`**

They are categorised based on the type of data structures used to implement the lists:

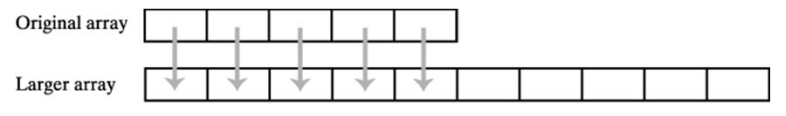
- Array (**`ArrayList & Vector`**)
- Sequence of linked data or *linked nodes* (**`LinkedList`**)

Array–based List

- Store elements in array
- The array is dynamically created. If the capacity of the array is exceeded, a larger new array is created and all the elements from the current array are copied to the new array (costly operation)
- Add & Remove require moving elements (costly operation except item at the end of list)
  - Eg method add(index,x), remove(x) are inefficient
- Each element can be directly accessed (fast access)
    - Eg. method get(), set() are efficient

Dynamic Array Expansion

- An array has a fixed size
- When array becomes full, move its contents to a larger array
  - create a new array eg. twice the current size
  - copy data from original to new location
  - the new array now becomes the current array.

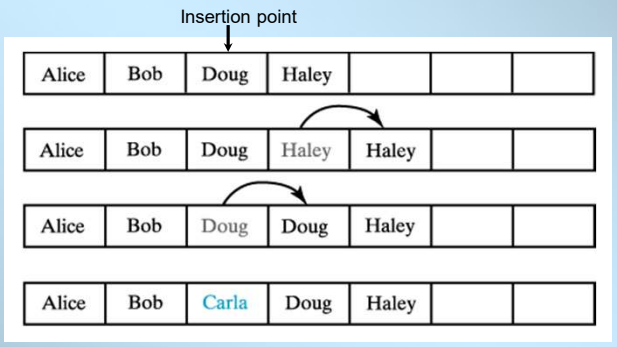


`List add()` Methods

- `add(x)` method adds new item x at the end of the list
  - Assign x at end
  - Increment `size` of list by 1
- `add(index,x)` method adds new item x in mid-list (at position index)
  - Requires shifting ahead from last element until element at index
  - Assign x at index
  - Increment `size` of list by 1

Adding Items in Mid-list

eg. add(2, “Carla”)



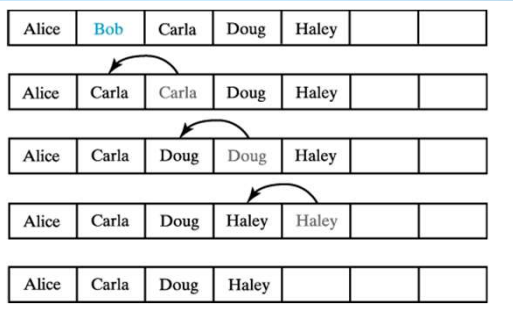
Making room to insert Carla as third entry in an array.

The `remove()` Method

- Must shift existing entries to avoid gap in the array
  - Except when removing last entry
  - In all cases, decrease size by 1
- Method must also handle error situation
  - When position specified in the remove is invalid
  - When remove() is called and the list is empty
  - Invalid call returns null value

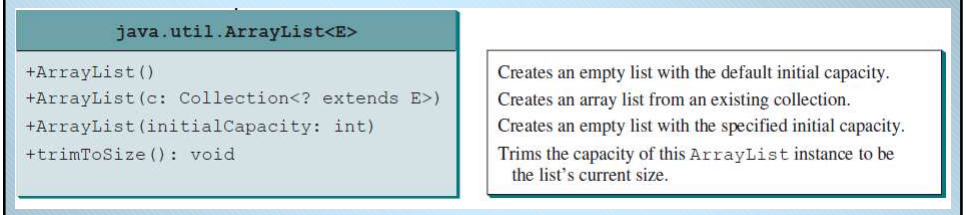
Removing a List Entry

eg. remove("Bob") or remove(1).



Removing Bob by shifting array entries.

The ArrayList Class



Using `ArrayList` Class

To create an ArrayList object:

List <Element Type> listvar = new ArrayList <>( );

OR

ArrayList <Element Type> listvar = new ArrayList <>( );

e.g.

```
List<Integer>   numList = new ArrayList <>();
List<String>    names   = new ArrayList <>();
ArrayList<Student>  students = new ArrayList <>(10);
```

Using `ArrayList` Class

Once the ArrayList object is created, you can invokes all the operations by calling the related methods, e.g.:

```
numList.add(10); numList.add(12); numList.set(1,11);
names.add("Ali"); names.add(0, "Abu");
students.add(new Student("Siti", 10001));
students.add(new Student("John", 10002));
System.out.println(students.size());
System.out.println(names.get(1));
System.out.println(names.indexOf("Abu"));
System.out.println(names.remove(1));
System.out.println(names.indexOf("Ali"));
```

The Vector Class

In Java 2, Vector is the same as ArrayList, except that Vector contains the synchronized methods for accessing and modifying the vector. None of the new collection data structures introduced so far are synchronized.

If you don't need synchronization, use ArrayList since it work faster than Vector

The Vector Class

java.util.Vector<E>	
+Vector()	Creates a default empty vector with initial capacity 10.
+Vector(c: Collection<? extends E>)	Creates a vector from an existing collection.
+Vector(initialCapacity: int)	Creates a vector with the specified initial capacity.
+Vector(initCapacity: int, capacityIncr: int)	Creates a vector with the specified initial capacity and increment.
+addElement(o: E): void	Appends the element to the end of this vector.
+capacity(): int	Returns the current capacity of this vector.
+copyInto(anArray: Object[]): void	Copies the elements in this vector to the array.
+elementAt(index: int): E	Returns the object at the specified index.
+elements(): Enumeration<E>	Returns an enumeration of this vector.
+ensureCapacity(): void	Increases the capacity of this vector.
+firstElement(): E	Returns the first element in this vector.
+insertElementAt(o: E, index: int): void	Inserts o into this vector at the specified index.
+lastElement(): E	Returns the last element in this vector.
+removeAllElements(): void	Removes all the elements in this vector.
+removeElement(o: Object): boolean	Removes the first matching element in this vector.
+removeElementAt(index: int): void	Removes the element at the specified index.
+setElementAt(o: E, index: int): void	Sets a new element at the specified index.
+setSize(newSize: int): void	Sets a new size in this vector.
+trimToSize(): void	Trims the capacity of this vector to its size.

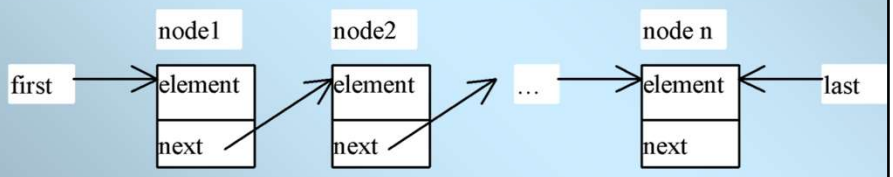
Using the Vector Class

```
Vector vector = new Vector();
System.out.println("Before adding elements:");
System.out.println("Capacity = "+vector.capacity());
System.out.println("Size = "+vector.size());
vector.addElement(1);
vector.addElement(2);
vector.addElement(3);
vector.insertElementAt(4,1);
System.out.println("After adding elements:");
System.out.println("Capacity = "+vector.capacity());
System.out.println("Size = "+vector.size());
```

OUTPUT:
Before adding elements:
Capacity = 10
Size = 0
After adding elements:
Capacity = 10
Size = 4

Linked-based List

- Implemented using Linked List
  - A linked list consists of a chain/sequence of nodes
  - Each node is an object that contains two attributes:
    - Element – store the element
    - Next – store reference to its next neighbor
      (Null if this is the last node in the sequence)



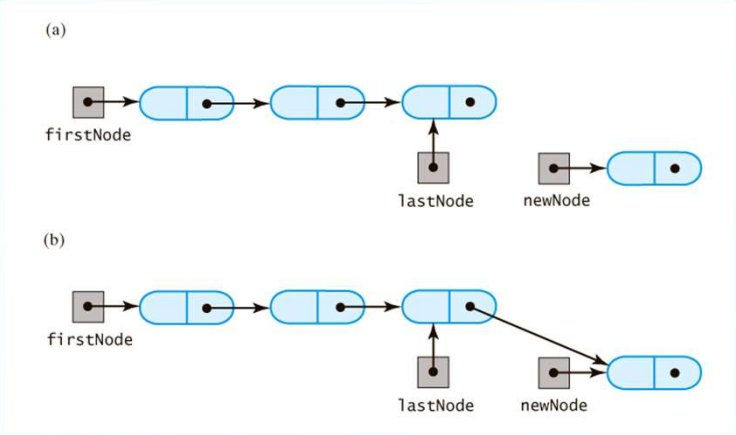
Pros and Cons of using Linked List

- The chain (list) can grow as large as necessary
- Can add and remove nodes without shifting existing entries

But …

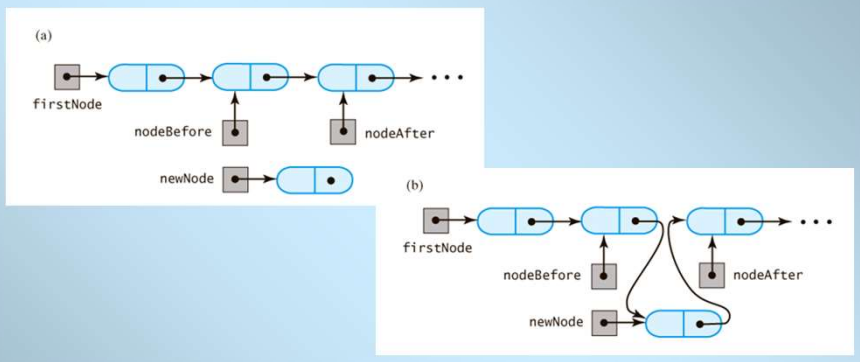
- Must traverse a chain to determine where to make addition/deletion
- Retrieving an entry requires traversal
  (As opposed to direct access in an array)

add(x) - Adding to the end List



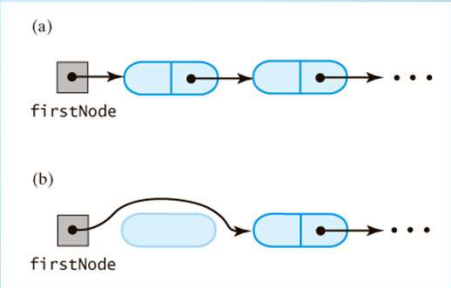
A chain of nodes (a) just prior to adding a node at the end; (b) just after adding a node at the end.

add(index,x) – Adding item in mid-List



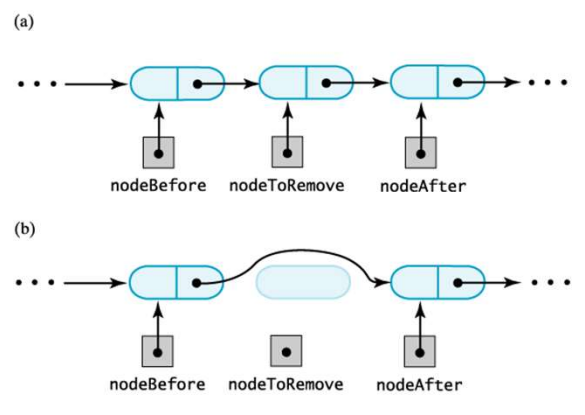
A chain of nodes (a) prior to adding node between adjacent nodes; (b) after adding node between adjacent nodes

The Method `remove`



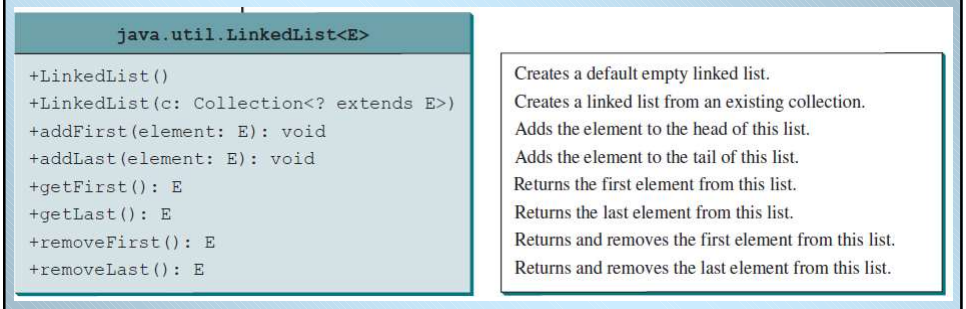
A chain of nodes (a) prior to removing first node; (b) after removing the first node

The Method `remove`



A chain of nodes (a) prior to removing interior node; (b) after removing interior node

The `LinkedList` Class



Using `LinkedList` Class

To create an LinkedList object:

List <Element Type> listvar = new LinkedList <>( );

OR

LinkedList <Element Type> listvar = new LinkedList <>( );

eg.

```
List <Integer>  numList = new LinkedList <Integer>();
List <String>   names   = new LinkedList <String>();
LinkedList <Student>  students = new LinkedList
<Student>();
```

List traversal

Iterator enables us to iterate over or traverse a collection

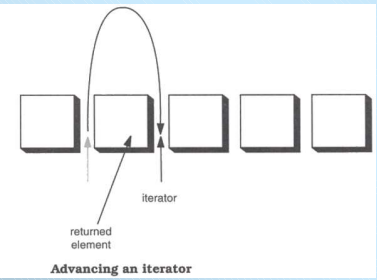
`Iterator` is an interface (in **`java.util`**)

To get the iterator object, call the **`iterator()`** method that will return an implementation of these methods

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

Iterator Interface

- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to next() “reads” an element from the collection
  - Then you can use it or remove it



Advancing an iterator

Iterator for List traversal

Iterator Approach:

```
Iterator<Element_type> it =
list.iterator();
while (it.hasNext())
   System.out.println(it.next());
```

Java 5 Feature

List Traversal

Using a JDK 1.5, you can use enhanced "for each" loop without using an iterator, as follows:

```
for (Element_type element : list)
  System.out.print(element + " ");
```

List Traversal with **ArrayList**

This example creates an array list with specified capacity, and inserts elements into the specified location & remove an element in the list.

Finally, the example traverses the list using the iterator.

Using **ArrayList** Class

```
ArrayList<Integer> aList = new ArrayList<Integer>();
aList.add(1);  // 1
aList.add(2);  // 1 -> 2
aList.add(3);   // 1 -> 2 -> 3
aList.add(1);   // 1 -> 2 -> 3 -> 1
aList.add(0,10); // 10 -> 1 -> 2 -> 3 -> 1
aList.add(3,30);  // 10 -> 1 -> 2 -> 30 -> 3 -> 1
aList.remove(1);  // 10 -> 2 -> 30 -> 3 -> 1
aList.set(3,40); // 10 -> 2 -> 30 -> 40 -> 1
Iterator<Integer> listItr = aList.iterator();
while (listItr.hasNext()) {
  System.out.print(listItr.next() + "");
}
```

OUTPUT: **10 2 30 40 1**

List Traversal with **LinkedList**

```
LinkedList<String> lList = new LinkedList<String>();
lList.add("red");
lList.add(0,"green");
lList.add("blue");
lList.add(1,"yellow");
lList.remove(lList.size()-1);
Iterator<String>listItr = lList.iterator();
while (listItr.hasNext()) {
  System.out.println(listItr.next());
}
```

Output:

green
yellow
red

List Traversal with **LinkedList**

```
LinkedList<String> lList = new LinkedList<String>();
lList.add("red");
lList.add(0,"green");
lList.add("blue");
lList.add(1,"yellow");
lList.remove(lList.size()-1);
for (String element : lList) {
  System.out.println(element);
}
```

Output:

green
yellow
red

Finding index of element in a List

indexOf() returns the index of the first matching element in this list,
or -1 if the element is not found

lastIndexOf() returns the index of the last matching element in this list,
or -1 if the element is not found

The equals() method is used to compare for equality

Finding index of element in a List

If elements are of user-defined class type, need to override the equals() in the class.

Example (for a Person class):

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj.getClass()!= this.getClass()) {
          return false;
    }
    Person p = (Person)obj;
    return this.name.equalsIgnoreCase(p.name) &&
    this.age == p.age ;
}
```