# Collections List Interface Framework

## 1. Collection <E> interface (E – the type of elements in the collection)

a) If we want to represent a group of individual objects as single entity then we should go for Collection.

b) Collection interface acts as the root interface of the collections framework & it contains most commonly used methods for any collection object.

c) Collection interface is typically used to pass Collections around & manipulate them where maximum generality is desired.

d) The JDK doesn't provide any direct implementations of Collection interface.

e) **Methods in Collection interface**

1. **boolean add (E e) :** To add an object to the collection
2. **boolean addAll (Collection<? Extends E> c) :** To add a group of objects in the Collection

3. **boolean remove (Object o) :** removes a particular object from the collection
4. **boolean removeAll (Collection<?> c) :** removes a group of objects from the collection

5. **boolean retainAll (Collection<?> c) :** except a particular group of objects, remove all other objects

6. **void clear() :** removes all the objects/elements from the collection

7. **boolean contains (Object o) :** checks a particular object is available or not
8. **boolean contains (Collection<?> c) :** checks a group of objects available or not

9. **boolean isEmpty () :** checks if the collection is empty or not
10. **int size () :** returns the number of objects in the collection

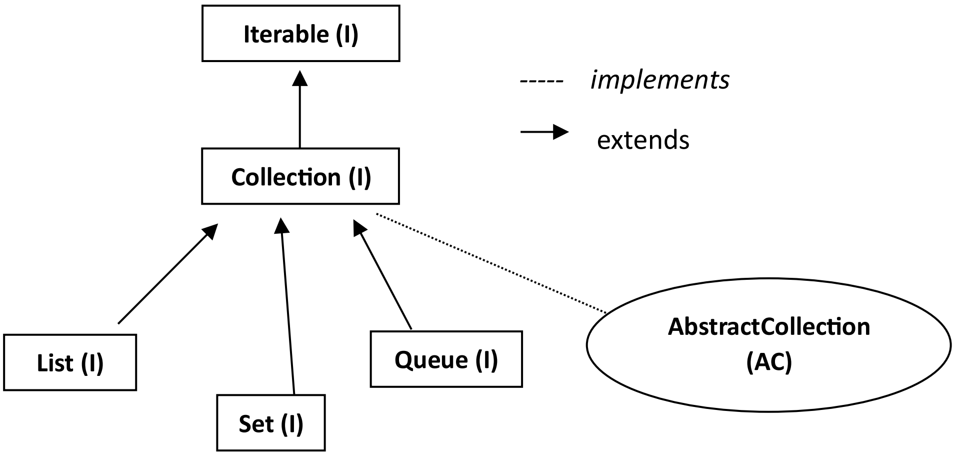
11. **Object [] toArray () :** returns an array containing all of the elements in the collection

12. **Iterator<E> iterator () :** returns an iterator over the elements in the collection i.e., to get object one by one from the collection

13. **default boolean removeIf (Predicate<? super E> filter) :** removes all of the elements of this collection that satisfy the given predicate.

14. **default Stream<E> parallelStream () :** returns a possibly parallel Stream with this collection as its source.
15. **default Stream<E> stream () :** returns a sequential stream with this collection as its source

16. **default Spliterator<E> spliterator () :** creates a Spliterator over the elements in this collection



## 2. List <E> interface extends Collection<E> (E – the type of elements in this list)

**[Ordered Collection]**

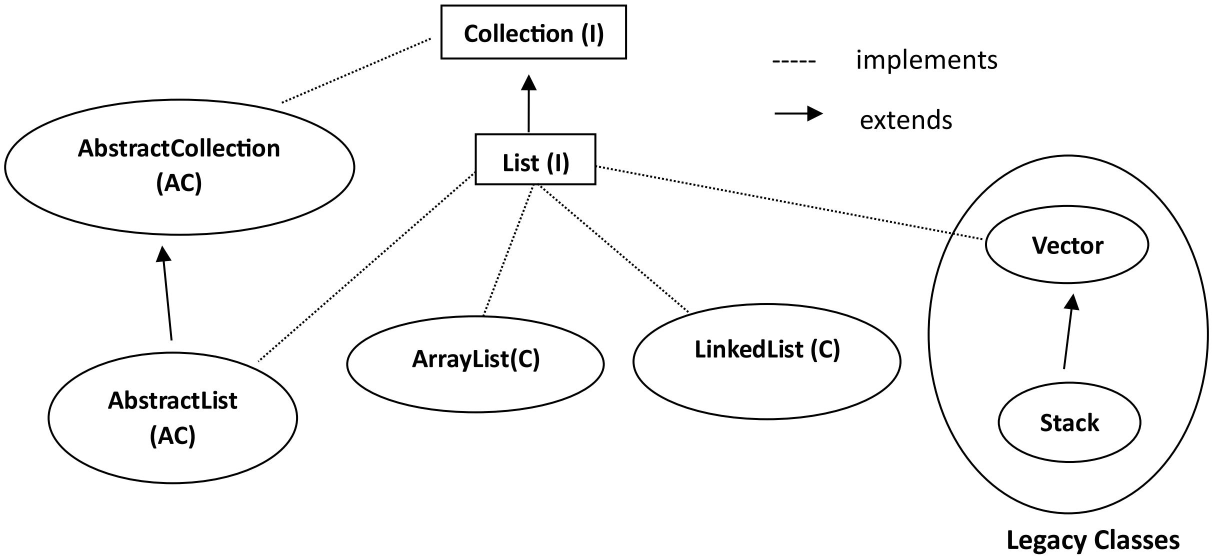
**a)** Child interface of Collection interface

**b)** If we want to represent a group of individual objects as single entity where duplicates are allowed & insertion order must be preserved then we should go for List interface.

**c)** We can preserve insertion order via index & we can differentiate duplicate objects by using index, hence index will play very important role in List.

**d)** **Methods in List interface**

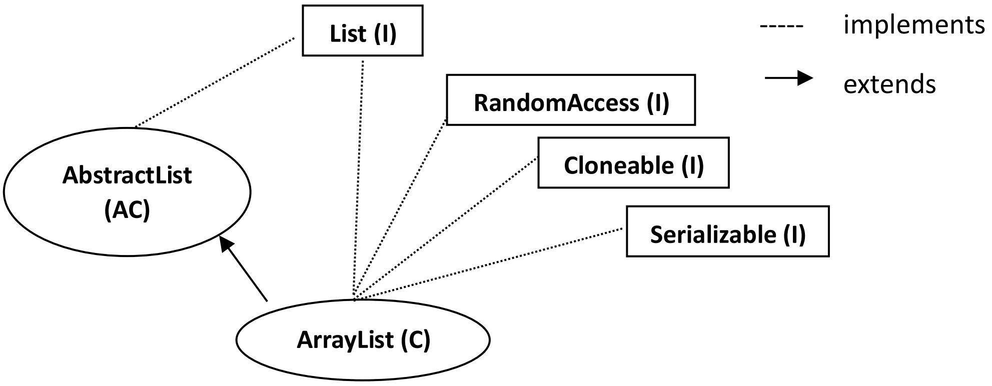
1. **void add (int index, E element) :** adds a particular object at a particular index.
2. **boolean addAll (int index, Collection<? Extends E> c) :** adds a group of objects started from this index onwards.
3. **E get (int index) :** returns the element at the specified position in this list.
4. **E set (int index, E element) :** replaces the element at the specified position in this list with the specified element.
5. **E remove (int index) :** removes a specified index object.
6. **int indexOf (Object o) :** returns index of first occurrence of specified object.
7. **int lastIndexOf (Object o) :** returns the last index of occurrence of specified object.
8. **ListIterator<E> listIterator ()** : returns a list iterator over the elements in this list (in proper sequence).
9. **ListIterator<E> listIterator (int index) :** returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
10. **List<E> subList (int fromIndex, int toIndex) :** returns a view of the portion of this list between the specified fromIndex (inclusive) & toIndex (exclusive).
11. **default void sort (Comparator<? super E> c) :** sorts this list according to the order induced by the specified Comparator.
12. **default void replaceAll (UnaryOperator<E> operator) :** replaces each element of this list with the result of applying the operator to that element.



# List Interface implemented classes

**2.1 ArrayList<E> class** extends **AbstractList<E>** implements **List<E>, RandomAccess,** Cloneable, Serializable
**2.2 LinkedList<E> class** extends **AbstractSequentialList<E>** implements **List<E>, Deque<E>,** Cloneable, Serializable
**2.3 Vector<E> class** extends **AbstractList<E>** implements **List<E>, RandomAccess,** Cloneable, Serializable
**2.4 Stack<E> class** extends **Vector<E>**

## 2.1 ArrayList<E> class extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable



**a)** The underlying data structure is Resizable array or Growable array.
**b)** Duplicates objects are allowed & insertion order is preserved.
**c)** Heterogenous objects are allowed (except TreeSet & TreeMap, everywhere heterogenous objects are allowed)
**d)** Null insertion is possible.
**e)** By default, ArrayList is non – synchronized but we can get synchronized version of **ArrayList** object by using **synchronizedList ()** method of Collections class.

**public static List synchronizedList (List l)**

e.g.,

**ArrayList l = new ArrayList ();**
**List l1 = Collections.synchronizedList (l);**

**f) Constructors in ArrayList class**

1. **ArrayList l = new ArrayList ()** : creates an empty ArrayList object with **default initial capacity 10**. Once it reaches its max capacity then a new ArrayList object will be created with formula

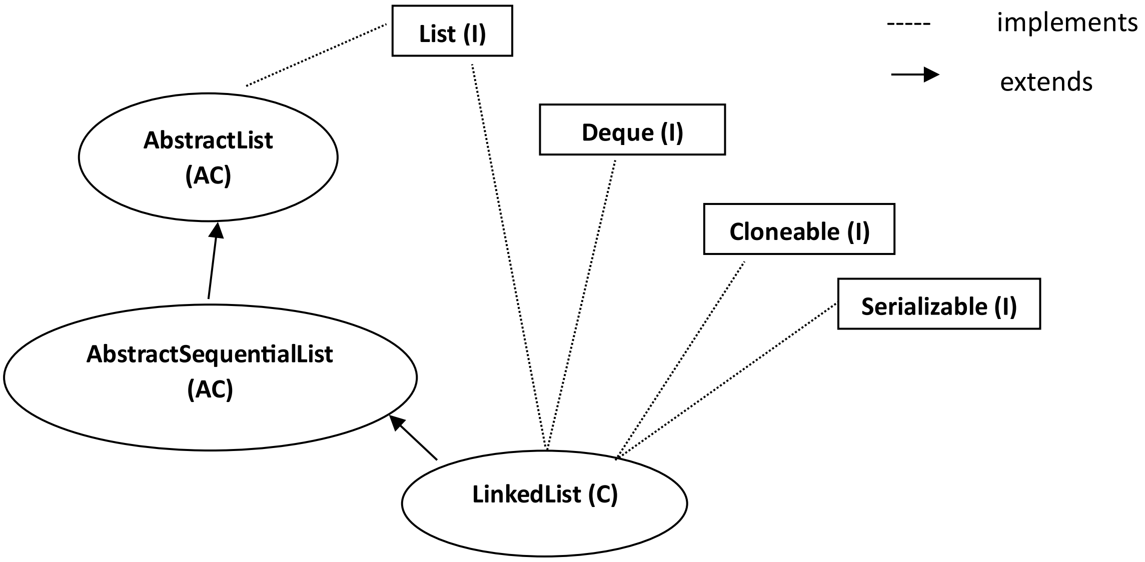
   **New capacity = (Current Capacity * 3/2) + 1**

2. **ArrayList l = new ArrayList (int initialCapacity)** : constructs an empty list with the specified initial capacity.
3. **ArrayList l = new ArrayList (Collection<? extends E> c)**
   - creates an equivalent ArrayList object for the given Collection.
   - This constructor meant for interconversion between Collection objects.

**g)** ArrayList is the best choice if our frequent operation is retrieval operation (because ArrayList implements RandomAccess interface which is a marker interface)

**h)** ArrayList is the worst choice if our frequent operation is insertion or deletion in middle (because of shift operation)

## 2.2 LinkedList<E> class extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable



**a)** The underlying data structure is doubly – linked list & deque.
**b)** Duplicate objects are allowed & Insertion order is preserved.
**c)** Null insertion is possible.
**d)** By default, LinkedList is non – synchronized but we can get synchronized version same as ArrayList.

**e)** **Constructors in LinkedList class**

1. **LinkedList l = new LinkedList ()** : creates an empty linked list object.
2. **LinkedList l = new LinkedList (Collection c)** : creates an equivalent LinkedList object for the given collection.

**f)** **Methods in LinkedList<E>** class (Usually we can use LinkedList to develop Stack & Queues. To provide support for this requirement, LinkedList class defines following specific methods)

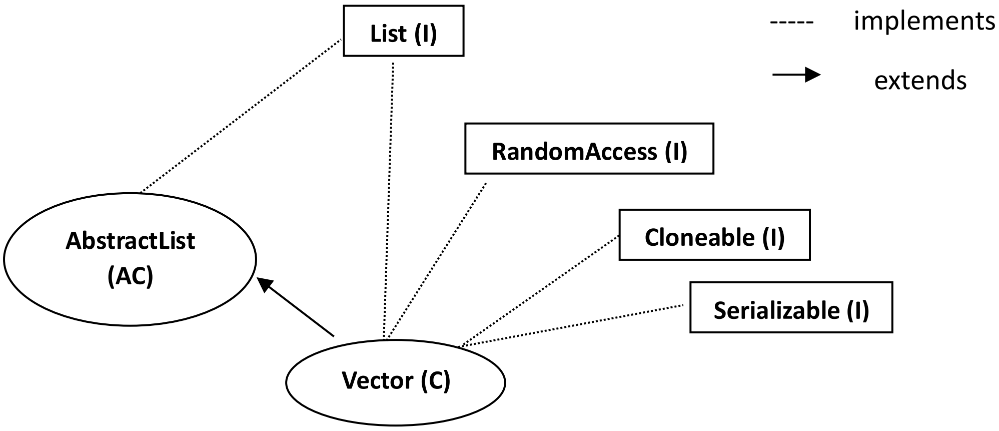
1. **void addFirst (E e) :** inserts the specified element at the beginning of this list.
2. **void addLast (E e) :** inserts the specified element to the end of this list.

3. **E getFirst () :** returns the first element in this list.
4. **E getLast () :** returns the last element in this list.

5. **E removeFirst () :** removes & returns the first element from this list.
6. **E removeLast () :** removes & returns the last element from this list.

**g)** LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.
**h)** LinkedList is the worst choice if our frequent operation is retrieval operation.

**2.3 Vector<E> class** extends **AbstractList<E>** implements **List<E>, RandomAccess,** Cloneable, Serializable



**a)** The underlying data structure is Resizable array or Growable array.
**b)** Everything is same as ArrayList<E> except Vector object are synchronized.

**c) Constructors in Vector class**

1. **Vector v = new Vector ()**
   - Creates an empty Vector object with default initial capacity 10.
   - Once vector reaches its max capacity, then a new Vector object will be created with new capacity using formula

     **New capacity = Current capacity * 2**

2. **Vector v = new Vector (int initial_capacity)**
3. **Vector v = new Vector (int initial_capacity, int increment_capacity)** : creates an empty Vector object with specified initial_capacity & how much increment in size of vector is specified by increment_capacity.
4. **Vector v = new Vector (Collection c)**

**d) Vector specific methods**

1. **void addElement (E obj) :** adds the specified component to the end of this vector, increasing its size by one.

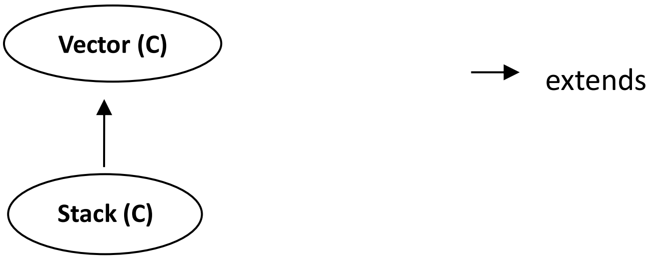
2. **boolean removeElement (Object obj) :** removes the first (lowest-indexed) occurrence of the argument from this vector.
3. **void removeElementAt (int index) :** deletes the component at the specified index.
4. **void removeAllElements () :** removes all components from this vector & sets its size to zero.

5. **E firstElement () :** returns the first component(the item at index 0) of this vector.

6. **E lastElement () :** returns the last component of the vector

7. **E elementAt (int index) :** returns the component at the specified index.

## 2.4 Stack<E> class extends Vector<E>



**a)** Stack class is specially designed class for LIFO (Last In First Out) order.

**b) Constructor in Stack class: Stack s = new Stack ();**

**c) Methods in Stack class**

1. **E push (E item) :** pushes an item onto the top of this stack.

2. **E pop () :** removes the object at the top of this stack & returns that object as the value of this function.

3. **E peek () :** looks at the object at the top of this stack without removing it from the stack.

4. **boolean empty () :** tests if this stack is empty.

5. **int search (Object o) :** returns offset if the element is available otherwise returns -1.
   - Offset means position from the top.