**Collections List Interface Framework**

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

1. If we want to represent a group of individual objects as single entity then we should go for Collection.
2. Collection interface acts as the root interface of the collections framework & it contains most commonly used methods for any collection object.
3. Collection interface is typically used to pass Collections around & manipulate them where maximum generality is desired.
4. The JDK doesn’t provide any direct implementations of Collection interface.
5. **Methods in Collection interface**
6. **boolean add (E e) :** To add an object to the collection
7. **boolean addAll (Collection<? Extends E> c) :** To add a group of objects in the Collection
8. **boolean remove (Object o) :** removes a particular object from the collection
9. **boolean removeAll (Collection<?> c) :** removes a group of objects from the collection
10. **boolean retainAll (Collection<?> c) :** except a particular group of objects, remove all other objects
11. **void clear() :** removes all the objects/elements from the collection
12. **boolean contains (Object o) :** checks a particular object is available or not
13. **boolean contains (Collection<?> c) :** checks a group of objects available or not
14. **boolean isEmpty () :** checks if the collection is empty or not
15. **int size () :** returns the number of objects in the collection
16. **Object [] toArray () :** returns an array containing all of the elements in the collection
17. **Iterator<E> iterator () :** returns an iterator over the elements in the collection i.e., to get object one by one from the collection
18. **default boolean removeIf (Predicate<? super E> filter) :** removes all of the elements of this collection that satisfy the given predicate.
19. **default Stream<E> parallelStream () :** returns a possibly parallel Stream with this collection as its source.
20. **default Stream<E> stream () :** returns a sequential stream with this collection as its source
21. **default Spliterator<E> spliterator () :** creates a Spliterator over the elements in this collection

**Iterable (I)**

*----- implements*

extends

**Collection (I)**

**Queue (I)**

**List (I)**

**Set (I)**

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

**[Ordered Collection]**

1. Child interface of Collection interface
2. 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.
3. We can preserve insertion order via index & we can differentiate duplicate objects by using index, hence index will play very important role in List.
4. **Methods in List interface**
5. **void add (int index, E element) :** adds a particular object at a particular index.
6. **boolean addAll (int index, Collection<? Extends E> c) :** adds a group of objects started from this index onwards.
7. **E get (int index) :** returns the element at the specified position in this list.
8. **E set (int index, E element) :** replaces the element at the specified position in this list with the specified element.
9. **E remove (int index) :** removes a specified index object.
10. **int indexOf (Object o) :** returns index of first occurrence of specified object.
11. **int lastIndexOf (Object o) :** returns the last index of occurrence of specified object.
12. **ListIterator<E> listIterator ()** : returns a list iterator over the elements in this list (in proper sequence).
13. **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.
14. **List<E> subList (int fromIndex, int toIndex) :** returns a view of the portion of this list between the specified fromIndex (inclusive) & toIndex (exclusive).
15. **default void sort (Comparator<? super E> c) :** sorts this list according to the order induced by the specified Comparator.
16. **default void replaceAll (UnaryOperator<E> operator) :** replaces each element of this list with the result of applying the operator to that element.

**Collection (I)**

**List (I)**

----- implements

extends

**Legacy Classes**

**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

**List (I)**

----- implements

extends

**RandomAccess (I)**

**Cloneable (I)**

**Serializable (I)**

1. The underlying data structure is Resizable array or Growable array.
2. Duplicates objects are allowed & insertion order is preserved.
3. Heterogenous objects are allowed (except TreeSet & TreeMap, everywhere heterogenous objects are allowed)
4. Null insertion is possible.
5. 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);**

1. **Constructors in ArrayList class**
2. **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**

1. **ArrayList l = new ArrayList (int initialCapacity)** : constructs an empty list with the specified initial capacity.
2. **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.

1. ArrayList is the best choice if our frequent operation is retrieval operation (because ArrayList implements RandomAccess interface which is a marker interface)
2. 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

----- implements

**List (I)**

extends

**Deque (I)**

**Cloneable (I)**

**Serializable (I)**

1. The underlying data structure is doubly – linked list & deque.
2. Duplicate objects are allowed & Insertion order is preserved.
3. Null insertion is possible.
4. By default, LinkedList is non – synchronized but we can get synchronized version same as ArrayList.
5. **Constructors in LinkedList class**
6. **LinkedList l = new LinkedList ()** : creates an empty linked list object.
7. **LinkedList l = new LinkedList (Collection c)** : creates an equivalent LinkedList object for the given collection.
8. **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)
9. **void addFirst (E e) :** inserts the specified element at the beginning of this list.
10. **void addLast (E e) :** inserts the specified element to the end of this list.
11. **E getFirst () :** returns the first element in this list.
12. **E getLast () :** returns the last element in this list.
13. **E removeFirst () :** removes & returns the first element from this list.
14. **E removeLast () :** removes & returns the last element from this list.
15. LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.
16. 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

----- implements

**List (I)**

extends

**RandomAccess (I)**

**Cloneable (I)**

**Serializable (I)**

1. The underlying data structure is Resizable array or Growable array.
2. Everything is same as ArrayList<E> except Vector object are synchronized.
3. **Constructors in Vector class**
4. **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**

1. **Vector v = new Vector (int initial\_capacity)**
2. **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.
3. **Vector v = new Vector (Collection c)**
4. **Vector specific methods**
5. **void addElement (E obj) :** adds the specified component to the end of this vector, increasing its size by one.
6. **boolean removeElement (Object obj) :** removes the first (lowest-indexed) occurrence of the argument from this vector.
7. **void removeElementAt (int index) :** deletes the component at the specified index.
8. **void removeAllElements () :** removes all components from this vector & sets its size to zero.
9. **E firstElement () :** returns the first component(the item at index 0) of this vector.
10. **E lastElement () :** returns the last component of the vector
11. **E elementAt (int index) :** returns the component at the specified index.

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

extends

1. Stack class is specially designed class for LIFO (Last In First Out) order.
2. **Constructor in Stack class:** **Stack s = new Stack ();**
3. **Methods in Stack class**
4. **E push (E item) :** pushes an item onto the top of this stack.
5. **E pop () :** removes the object at the top of this stack & returns that object as the value of this function.
6. **E peek () :** looks at the object at the top of this stack without removing it from the stack.
7. **boolean empty () :** tests if this stack is empty.
8. **int search (Object o) :** returns offset if the element is available otherwise returns -1.

* Offset means position from the top.