

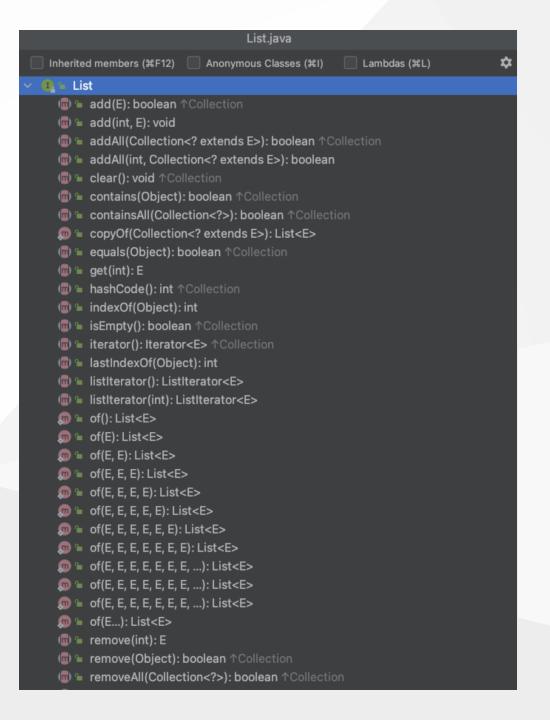
# **Collection List**

Rawlabs Academy

### List

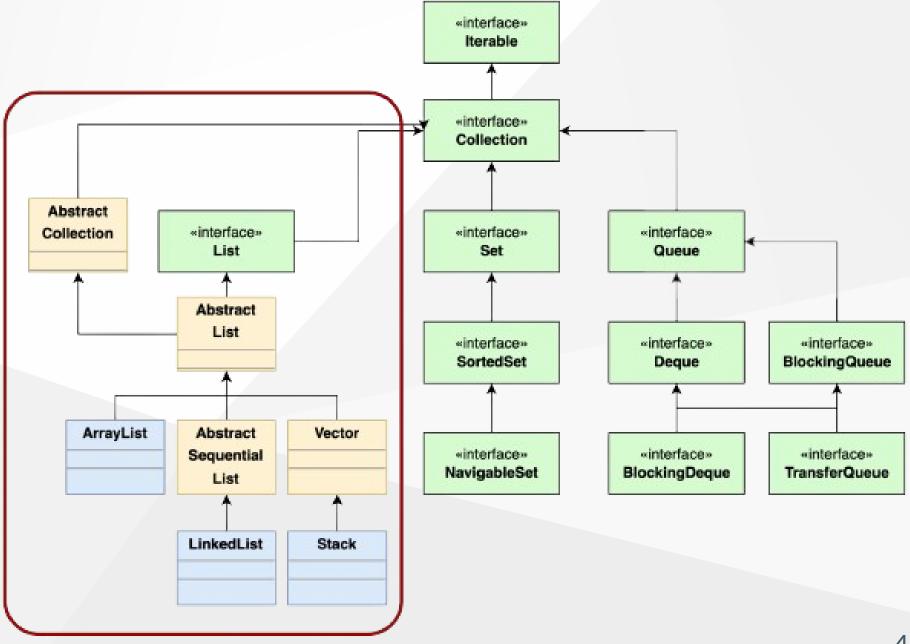
- The List interface provides a way to **store** and **ordered collection**.
- It is a **child interface** of **Collection**
- It is an ordered collection of object in which **duplicate values** can be stored.
- Since List preserves the insertion order, it allows **positional** access and insertion of elements.

# Methods List



### List

**Hierarchy** 

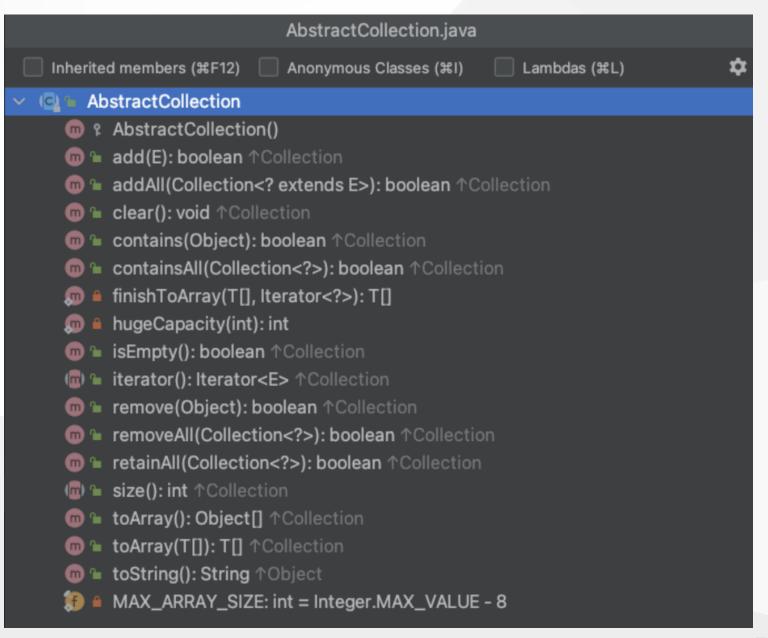


# Abstract Collection

It is used to implement an unmodifiable collection, for which one need to only extend this

AbstractCollection

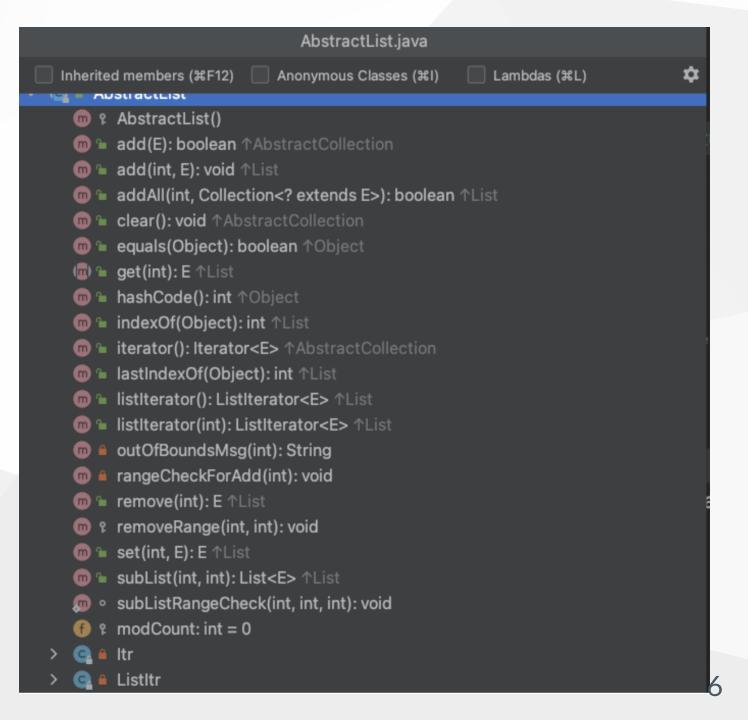
class and implement only the iterator and the size methods.



### AbstractList

This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface baked by Random Access data store (such an array). For sequential access data (such as linked list), AbstractSequentialList should be used in

preference to this class.



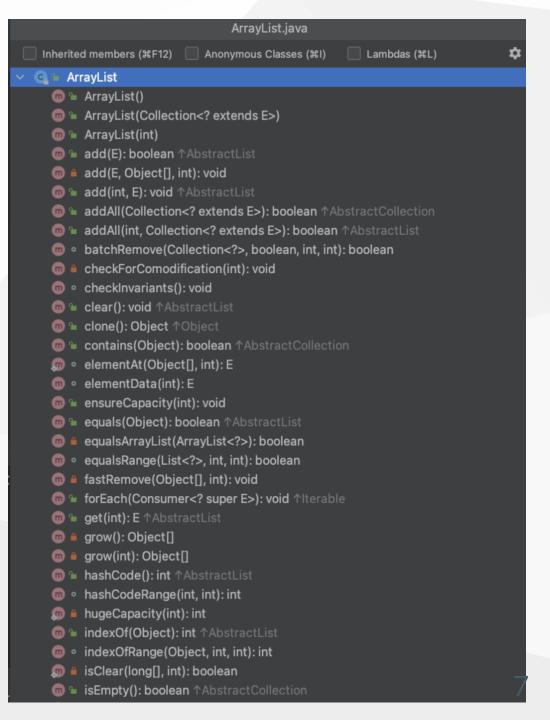
# ArrayList

Its provides us with **dynamic arrays** in java. Though, it may be slower than standard arrays but can be helpful in programs where **lots of manipulation** in the array needed.

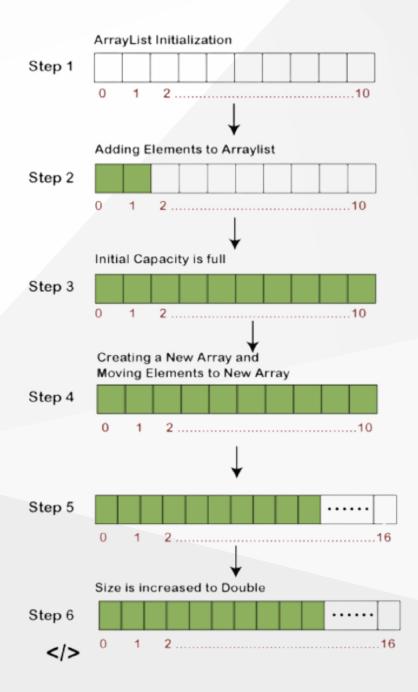
40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

<- Array Indices

Array Length = 9
First Index = 0
Last Index = 8



# ArrayList Work



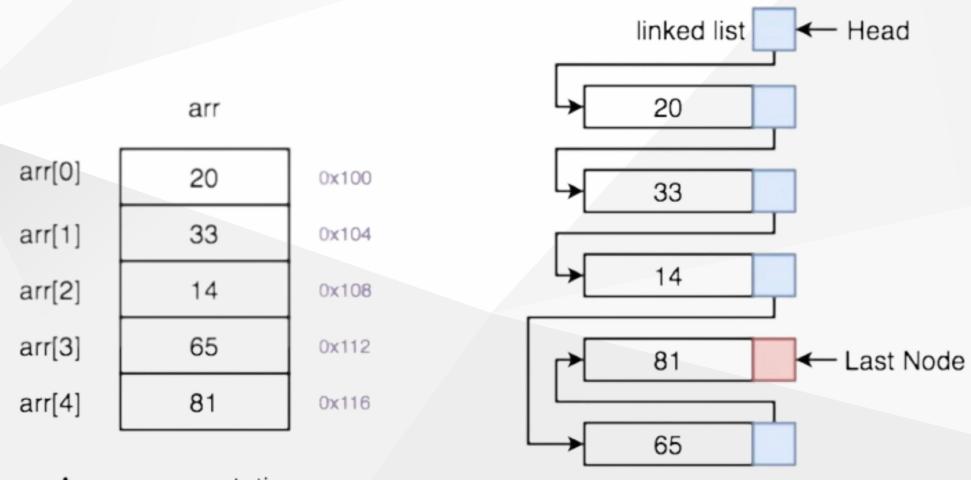
### LinkedList

- LinkedList consists of nodes where each node contains data and a reference to the next node in the list
- Unlike an array, data is not stored in one contigous block of memory and does not have a fixed size
- Instead, it consists of multiple blocks of memory at different addresses



#### LinkedList m LinkedList() • LinkedList(Collection<? extends E>) m = add(E): boolean ↑AbstractList m = add(int, E): void ↑AbstractSequentialList m addAll(Collection<? extends E>): boolean ↑AbstractCollection m addAll(int, Collection<? extends E>): boolean ↑AbstractSequentialList m = addFirst(E): void ↑Deque m = addLast(E): void ↑Deque m a checkElementIndex(int): void m a checkPositionIndex(int): void m = clear(): void ↑AbstractList m = clone(): Object ↑Object m = contains(Object): boolean ↑AbstractCollection @ • element(): E ↑Deque m = get(int): E ↑AbstractSequentialList m = getFirst(): E ↑Deque m = getLast(): E ↑Deque m indexOf(Object): int ↑AbstractList m isElementIndex(int): boolean m a isPositionIndex(int): boolean m = lastIndexOf(Object): int ↑AbstractList o linkBefore(E, Node<E>): void m | linkFirst(E): void m • linkLast(E): void m = listIterator(int): ListIterator<E> ↑AbstractSequentialList o node(int): Node<E> m • offer(E): boolean ↑Deque m • offerFirst(E): boolean ↑Deque m • offerLast(E): boolean ↑Deque m a outOfBoundsMsg(int): String 📵 🦆 peek(): E 个Deque

### Representation ArrayList vs LinkedList



Array representation

# **ArrayList Example**

```
public class Main {
    public static void main(String[] args) {
        List<String> addresses = new ArrayList<>();
        addresses.add("Milan");
        addresses.add("London");
        addresses.add("Guatemala");
        addresses.add("London");
        System.out.println(addresses.get(2));
```

### LinkedList Example

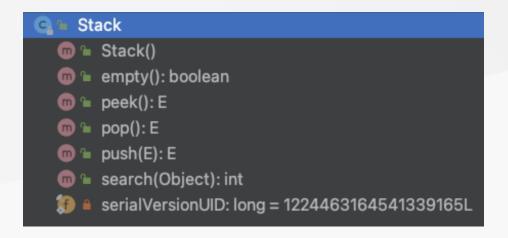
```
public class Main {
    public static void main(String[] args) {
        List<String> addresses = new LinkedList<>();
        addresses.add("Milan");
        addresses.add("London");
        addresses.add("Guatemala");
        addresses.add("London");
        System.out.println(addresses.get(2));
```

### **Immutable List Example**

```
public class Main {
    public static void main(String[] args) {
        List<String> addresses = new LinkedList<>();
        var immutableAddresses = Collections.unmodifiableList(addresses);
        immutableAddresses.add("Texas");
        addresses.add("Milan");
        addresses.add("London");
        addresses.add("Guatemala");
        addresses.add("London");
        System.out.println(addresses.get(2));
```

### Stack

- The Stack class represents a last-in-first-out
   (LIFO) stack of objects
- It extends class Vector with 5 operations that allow a vector to be treated as a stack
- The usual **push** and **pop** operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top
- A more complete and consistent set of LIFO stack operations is provoided by the Deque interface and its implementations, which should be used in preference to this class.



### **Stack Example**

```
public class Main {
    public static void main(String[] args) {
        Stack<String> addresses = new Stack<>();
        addresses.push("Milan");
        addresses.push("London");
        addresses.push("Guatemala");
        addresses.push("London");
        System.out.println(addresses.pop());
```

### **Task - Array Merge**

Make a program to combine 2 arrays, then insert several objects into the array in the middle with the index entered.

# **Task - Play with Parking Area**

It is known that there is a parking lot that only contains 1 motorcycle in each row. Make a program to manage the parking lot so that it fills the farthest parking lot with the parking gate first and the motorbike closest to the parking gate can exit first.