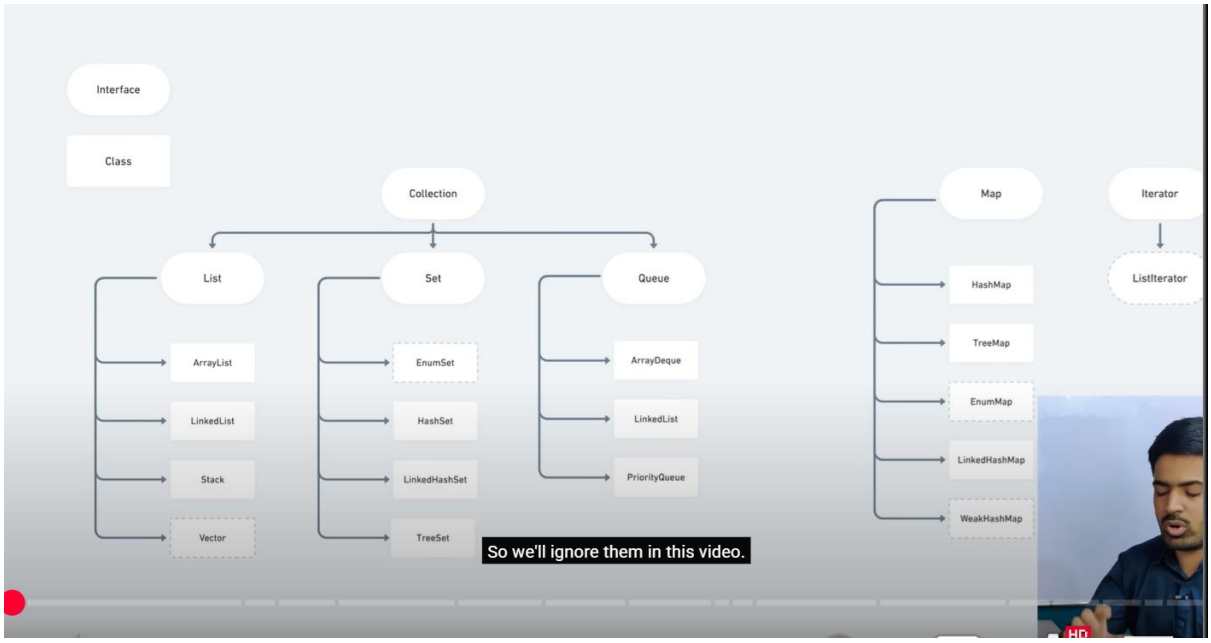
Java Collection Framework



1. List Interface

a. ArrayList

`ArrayList` is a **resizable array**, part of **java.util** package.

Unlike regular arrays in Java (`int[] arr = new int[5]`), an `ArrayList` **grows and shrinks** dynamically as elements are added or removed.

It **maintains the insertion order** (i.e., the order in which elements were added).

It **allows duplicate elements**.

It's a part of the **List** interface (i.e., it implements `List<E>`).

✅ Declaration and Initialization

```
import java.util.ArrayList;

public class Demo {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>(); // creating an empty
list of strings
    }
```

```
}
```

You can also specify an initial capacity:

```
ArrayList<Integer> numbers = new ArrayList<>(10); // optional
```

✅ Basic Operations

```
ArrayList<String> fruits = new ArrayList<>();

// Add elements
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Mango");

// Get element by index
System.out.println(fruits.get(1)); // Banana

// Set (replace) an element
fruits.set(1, "Orange"); // Replaces Banana with Orange

// Remove element
fruits.remove("Apple"); // by value
fruits.remove(0); // by index

// Size of ArrayList
System.out.println(fruits.size());

// Check if it contains an element
System.out.println(fruits.contains("Mango")); // true

// Clear all elements
fruits.clear();

// Check if empty
System.out.println(fruits.isEmpty());
```

✅ Iterating through an ArrayList

```
ArrayList<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");

// Using for loop
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}

// Using enhanced for-loop
for (String item : list) {
    System.out.println(item);
}

// Using forEach with lambda (Java 8+)
list.forEach(item -> System.out.println(item));
```

```
//extra code
✅ Full Example with Comments
import java.util.ArrayList;

public class ArrayListMethodsDemo {
    public static void main(String[] args) {
        // Create main list and add elements
        ArrayList<String> mainList = new ArrayList<>();
        mainList.add("A");
        mainList.add("B");
        mainList.add("D");

        System.out.println("Original mainList: " + mainList); // [A, B, D]

        // 1️⃣ add(index, element) → Insert "C" at index 2
        mainList.add(2, "C");
        System.out.println("After add(2, \"C\"): " + mainList); // [A, B,
C, D]

        // 2️⃣ addAll(collection) → Add list2 at the end
        ArrayList<String> list2 = new ArrayList<>();
        list2.add("E");
        list2.add("F");

        mainList.addAll(list2);
        System.out.println("After addAll(list2): " + mainList); // [A, B,
C, D, E, F]

        // 3️⃣ addAll(index, collection) → Insert list3 at index 1
        ArrayList<String> list3 = new ArrayList<>();
        list3.add("X");
        list3.add("Y");

        mainList.addAll(1, list3);
        System.out.println("After addAll(1, list3): " + mainList); // [A,
X, Y, B, C, D, E, F]
    }
}
```

```
📜 Output:
Original mainList: [A, B, D]
After add(2, "C"): [A, B, C, D]
After addAll(list2): [A, B, C, D, E, F]
After addAll(1, list3): [A, X, Y, B, C, D, E, F]
```

---

b. Stack

c. In Java, `Stack` is a **class** that is part of the **`java.util` package**.
d. It **extends `Vector`**, and behaves like a **stack data structure** (LIFO — Last In, First Out).
e. You **push** items onto the stack and **pop** them off from the top.

📦 Common Methods of `Stack`

Method	Description
`push(E item)`	Adds an item to the top of the stack
`pop()`	Removes and returns the top item
`peek()`	Returns (but doesn't remove) top item
`isEmpty()`	Checks if the stack is empty
`search(Object o)`	Returns 1-based position from top

✅ Example Program: Stack Usage

```
import java.util.Stack;

public class StackDemo {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();

        // 🔼 Push elements onto the stack
        stack.push("Apple");
        stack.push("Banana");
        stack.push("Cherry");

        System.out.println("Stack after pushes: " + stack);  // [Apple,
Banana, Cherry]

        // 👀 Peek at top element
        System.out.println("Top element (peek): " + stack.peek()); //
Cherry

        // ⬇️ Pop the top element
        String removed = stack.pop();
        System.out.println("Popped element: " + removed);          // Cherry
        System.out.println("Stack after pop: " + stack);           //
[Apple, Banana]

        // ❓ Check if stack is empty
        System.out.println("Is stack empty? " + stack.isEmpty()); // false

        // 🔍 Search for an element (1-based position from top)
        System.out.println("Position of 'Apple': " +
stack.search("Apple")); // 2
        System.out.println("Position of 'Banana': " +
stack.search("Banana")); // 1
        System.out.println("Position of 'Cherry': " +
stack.search("Cherry")); // -1 (already removed)
    }
}
```

c. LinkedList

The **LinkedList** class in Java is **a very versatile data structure**. Because of its underlying **doubly linked list** structure, it can act both like:

- A **List** (ordered, indexed collection — like ArrayList)
- A **Queue/Deque** (FIFO, LIFO, or double-ended operations)

So, it **implements multiple interfaces**:

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, Serializable
```

✅ **Because It Implements List<E>, It Supports:**

- Index-based access: get(index), add(index, element), remove(index)
- Ordered elements (in insertion order)
- Allows duplicates

🔁 Think of LinkedList as a **list of items you can iterate through in order**, just like ArrayList.

✅ **Because It Implements Deque<E> (which extends Queue<E>), It Supports:**

- Queue operations: offer(), poll(), peek() (FIFO)
- Stack operations: push(), pop() (LIFO)
- Double-ended operations: add/remove from both front and rear (addFirst(), addLast(), etc.)

🔁 So, LinkedList can also behave like:

- A **Queue** → for tasks like buffering, printing jobs, etc.
- A **Stack** → for undo history, backtracking, etc.

Java Example – peek() and poll() with Queue

```
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import java.util.Deque;

public class LinkedListDemo {
    public static void main(String[] args) {
        // ✅ LinkedList as a List
        List<String> list = new LinkedList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");
        list.add(1, "Blueberry");  // Insert at index 1
        System.out.println("As List: " + list); // [Apple, Blueberry,
Banana, Cherry]

        // ✅ LinkedList as a Queue (FIFO)
        Queue<String> queue = new LinkedList<>();
```

```
        queue.offer("Task1");
        queue.offer("Task2");
        queue.offer("Task3");
        System.out.println("\nAs Queue:");
        while (!queue.isEmpty()) {
            System.out.println("Next in line (peek): " +
queue.peek()); // Look at the head
            System.out.println("Processing (poll): " + queue.poll());
// Remove the head
        }

        // ✅ LinkedList as a Stack (LIFO)
        Deque<String> stack = new LinkedList<>();
        stack.push("Page1");
        stack.push("Page2");
        stack.push("Page3");
        System.out.println("\nAs Stack:");
        while (!stack.isEmpty()) {
            System.out.println("Top of stack (peek): " +
stack.peek()); // Look at the top
            System.out.println("Going back to (pop): " +
stack.pop());  // Remove from top
        }
    }
}
```

🔍 **Output Will Look Like:**

```
As List: [Apple, Blueberry, Banana, Cherry]

As Queue:
Next in line (peek): Task1
Processing (poll): Task1
Next in line (peek): Task2
Processing (poll): Task2
Next in line (peek): Task3
Processing (poll): Task3

As Stack:
Top of stack (peek): Page3
Going back to (pop): Page3
Top of stack (peek): Page2
Going back to (pop): Page2
Top of stack (peek): Page1
Going back to (pop): Page1
```
