Sure! Let’s do a **deep dive into the List interface and its main implementing classes** in Java, in a **detailed, interview-ready way**.

**1. List Interface in Java**

**Definition**

* List is an **ordered collection** (also called a **sequence**) in Java.
* Allows **duplicate elements**.
* Elements can be **accessed by index** (position in the list).
* Part of the **Java Collections Framework**: java.util.List

**Key Methods**

* add(E e) – Adds an element to the list.
* add(int index, E e) – Adds element at a specific index.
* get(int index) – Returns element at specified index.
* set(int index, E e) – Updates element at a given index.
* remove(int index) – Removes element at a given index.
* indexOf(Object o) – Returns the first index of an element.
* lastIndexOf(Object o) – Returns the last index of an element.
* size() – Returns the number of elements.

**Properties**

| **Property** | **Description** |
| --- | --- |
| Order | Maintains insertion order |
| Duplicates | Allowed |
| Null Elements | Yes, depends on implementation |
| Random Access | Some implementations like ArrayList support O(1) access, LinkedList is O(n) |
| Thread Safety | Not synchronized by default; can use Collections.synchronizedList() |
| Implementing Classes | ArrayList, LinkedList, Vector, Stack |

**2. Main Implementing Classes of List**

**2.1 ArrayList**

* **Package:** java.util
* **Underlying Data Structure:** Resizable array
* **Properties:**
  + Maintains **insertion order**.
  + **Duplicate elements allowed**.
  + **Random access is fast (O(1))**.
  + **Insertion/deletion in middle is slow (O(n))**.
  + Not synchronized (use Collections.synchronizedList() for thread safety).
  + Can store null elements.
* **Use Cases:** Dynamic array scenarios, frequent read/access operations.

**Example:**

import java.util.ArrayList;

import java.util.List;

public class ArrayListDemo {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Alice");

list.add("Bob");

list.add("Charlie");

list.add("Bob"); // duplicate allowed

System.out.println(list);

}

}

**Output:**

[Alice, Bob, Charlie, Bob]

**2.2 LinkedList**

* **Package:** java.util
* **Underlying Data Structure:** Doubly linked list
* **Properties:**
  + Maintains **insertion order**.
  + **Duplicate elements allowed**.
  + **Efficient insertion/deletion at start, end, or middle (O(1) if pointer available)**.
  + **Random access is slow (O(n))**.
  + Implements **Deque interface** → can be used as **queue or stack**.
  + Not synchronized; can store null elements.
* **Use Cases:** Frequent insertion/deletion, queue/deque implementation.

**Example:**

import java.util.LinkedList;

import java.util.List;

public class LinkedListDemo {

public static void main(String[] args) {

LinkedList<Integer> ll = new LinkedList<>();

ll.add(10);

ll.add(20);

ll.addFirst(5); // add at beginning

ll.addLast(30); // add at end

System.out.println(ll);

}

}

**Output:**

[5, 10, 20, 30]

**2.3 Vector**

* **Package:** java.util
* **Underlying Data Structure:** Resizable array
* **Properties:**
  + Maintains **insertion order**.
  + **Duplicate elements allowed**.
  + **Synchronized (thread-safe)**.
  + Random access fast: O(1).
  + Insertion/deletion in middle slow: O(n).
  + Legacy class (from Java 1.0), replaced by ArrayList in most modern code.
* **Use Cases:** Thread-safe dynamic array (rare in modern applications).

**Example:**

import java.util.Vector;

public class VectorDemo {

public static void main(String[] args) {

Vector<String> vector = new Vector<>();

vector.add("A");

vector.add("B");

vector.add("C");

System.out.println(vector);

}

}

**Output:**

[A, B, C]

**2.4 Stack**

* **Package:** java.util
* **Underlying Data Structure:** Extends Vector
* **Properties:**
  + Implements **LIFO** (Last In First Out).
  + Synchronized (thread-safe).
  + Supports push(), pop(), peek(), empty() methods.
  + Legacy class; modern alternative: **Deque implementations** (ArrayDeque).
* **Use Cases:** Undo/redo, expression evaluation, stack operations.

**Example:**

import java.util.Stack;

public class StackDemo {

public static void main(String[] args) {

Stack<Integer> stack = new Stack<>();

stack.push(10);

stack.push(20);

stack.push(30);

System.out.println(stack.pop()); // 30

System.out.println(stack.peek()); // 20

}

}

**Output:**

30

20

**Summary Table of List Implementations**

| **Class** | **Underlying DS** | **Duplicates** | **Nulls Allowed** | **Thread-Safe** | **Random Access** | **Insert/Delete Efficiency** | **Use Case** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ArrayList | Resizable Array | Yes | Yes | No | Fast (O(1)) | Slow in middle (O(n)) | Frequent read, dynamic array |
| LinkedList | Doubly LinkedList | Yes | Yes | No | Slow (O(n)) | Fast at ends (O(1)) | Frequent insertion/deletion, queue/deque |
| Vector | Resizable Array | Yes | Yes | Yes | Fast (O(1)) | Slow in middle (O(n)) | Legacy thread-safe dynamic array |
| Stack | Vector | Yes | Yes | Yes | Fast (O(1)) | N/A (LIFO) | LIFO stack operations |

💡 **Interview Tip:**

* Know **when to use ArrayList vs LinkedList**: ArrayList → frequent read/random access, LinkedList → frequent insertion/deletion.
* Mention **Vector/Stack are legacy classes**.
* Be ready to **write small code examples** for each.

Great! Next, let’s dive into the **Set Interface and its main implementing classes in Java** in **detailed, interview-ready format**.

**2. Set Interface in Java**

**Definition**

* Set is a **collection that does not allow duplicate elements**.
* Part of the **Java Collections Framework**: java.util.Set
* **Main purpose:** Store **unique elements** only.

**Key Methods**

* add(E e) – Adds element if not already present.
* remove(Object o) – Removes specified element.
* contains(Object o) – Checks if element exists.
* size() – Returns number of elements.
* isEmpty() – Checks if set is empty.
* iterator() – Returns an iterator to traverse elements.

**Properties**

| **Property** | **Description** |
| --- | --- |
| Order | Depends on implementation (HashSet: unordered, LinkedHashSet: insertion order, TreeSet: sorted order) |
| Duplicates | Not allowed |
| Null Elements | Depends on implementation (HashSet/LinkedHashSet: one null, TreeSet: none for natural ordering) |
| Thread Safety | Not synchronized by default; can use Collections.synchronizedSet() |
| Implementing Classes | HashSet, LinkedHashSet, TreeSet |

**Main Implementing Classes of Set**

**2.1 HashSet**

* **Package:** java.util
* **Underlying Data Structure:** Hash table
* **Properties:**
  + Unordered collection.
  + **No duplicates allowed**.
  + Allows **one null element**.
  + Not synchronized.
  + Fast access (add/remove/contains ~ O(1) on average).
* **Use Cases:** Unique elements, fast lookup, mathematical set operations.

**Example:**

import java.util.HashSet;

import java.util.Set;

public class HashSetDemo {

public static void main(String[] args) {

Set<String> set = new HashSet<>();

set.add("Apple");

set.add("Banana");

set.add("Cherry");

set.add("Apple"); // duplicate ignored

System.out.println(set);

}

}

**Output:**

[Apple, Banana, Cherry] // order not guaranteed

**2.2 LinkedHashSet**

* **Package:** java.util
* **Underlying Data Structure:** Hash table + linked list
* **Properties:**
  + Maintains **insertion order**.
  + **No duplicate elements**.
  + Allows **one null element**.
  + Not synchronized.
* **Use Cases:** Unique elements where **iteration order matters**.

**Example:**

import java.util.LinkedHashSet;

import java.util.Set;

public class LinkedHashSetDemo {

public static void main(String[] args) {

Set<String> lhs = new LinkedHashSet<>();

lhs.add("Apple");

lhs.add("Banana");

lhs.add("Cherry");

lhs.add("Apple"); // duplicate ignored

System.out.println(lhs);

}

}

**Output:**

[Apple, Banana, Cherry] // insertion order maintained

**2.3 TreeSet**

* **Package:** java.util
* **Underlying Data Structure:** Red-Black tree (self-balancing binary search tree)
* **Properties:**
  + Elements are **sorted in natural order** (or using a comparator).
  + **No duplicates allowed**.
  + Null not allowed (for natural ordering).
  + Not synchronized.
  + Operations like add/remove/search take **O(log n)**.
* **Use Cases:** Sorted sets, range queries, elements in order.

**Example:**

import java.util.Set;

import java.util.TreeSet;

public class TreeSetDemo {

public static void main(String[] args) {

Set<Integer> ts = new TreeSet<>();

ts.add(30);

ts.add(10);

ts.add(20);

System.out.println(ts);

}

}

**Output:**

[10, 20, 30] // automatically sorted

**Summary Table of Set Implementations**

| **Class** | **Underlying DS** | **Order** | **Duplicates** | **Nulls Allowed** | **Thread-Safe** | **Use Case** |
| --- | --- | --- | --- | --- | --- | --- |
| HashSet | Hash table | Unordered | No | Yes (1 null) | No | Fast lookup, unique elements |
| LinkedHashSet | Hash table + linked list | Insertion order | No | Yes (1 null) | No | Unique elements with predictable iteration |
| TreeSet | Red-Black tree | Sorted order | No | No | No | Sorted unique elements, range queries |

💡 **Interview Tip:**

* Know **when to use HashSet vs LinkedHashSet vs TreeSet**.
* Emphasize **ordering, null handling, and duplicates**.
* Can mention **performance differences** (HashSet faster than TreeSet for basic operations).

If you want, I can **next cover the Queue interface and its classes with the same detailed approach**.

Do you want me to do that?

Great! Let’s move to the **Queue interface and its main implementing classes in Java**, explained in **detail for interview purposes**.

**3. Queue Interface in Java**

**Definition**

* Queue is a **collection designed for holding elements prior to processing**.
* Part of java.util package.
* Typically **FIFO (First-In-First-Out)** order, but some implementations like PriorityQueue have different ordering.

**Key Methods**

* add(E e) – Inserts element (throws exception if full).
* offer(E e) – Inserts element (returns false if full).
* remove() – Removes head element (throws exception if empty).
* poll() – Removes head element (returns null if empty).
* element() – Retrieves head element (throws exception if empty).
* peek() – Retrieves head element (returns null if empty).

**Properties**

| **Property** | **Description** |
| --- | --- |
| Order | Typically FIFO; may vary (PriorityQueue) |
| Duplicates | Allowed |
| Null Elements | Depends on implementation (PriorityQueue: null not allowed) |
| Thread Safety | Not synchronized by default; use ConcurrentLinkedQueue or synchronized wrapper |
| Implementing Classes | PriorityQueue, ArrayDeque, LinkedList |

**Main Implementing Classes of Queue**

**3.1 PriorityQueue**

* **Package:** java.util
* **Underlying Data Structure:** Priority heap (min-heap by default)
* **Properties:**
  + Elements ordered by **natural ordering** or **custom comparator**.
  + **Duplicates allowed**.
  + Null not allowed.
  + Not synchronized.
  + **Insertion/removal O(log n)** due to heap structure.
* **Use Cases:** Task scheduling, sorting by priority, min/max heap operations.

**Example:**

import java.util.PriorityQueue;

import java.util.Queue;

public class PriorityQueueDemo {

public static void main(String[] args) {

Queue<Integer> pq = new PriorityQueue<>();

pq.add(30);

pq.add(10);

pq.add(20);

System.out.println("PriorityQueue elements (polling):");

while (!pq.isEmpty()) {

System.out.println(pq.poll()); // smallest element first

}

}

}

**Output:**

10

20

30

**3.2 ArrayDeque**

* **Package:** java.util
* **Underlying Data Structure:** Resizable array
* **Properties:**
  + Implements **Deque** → can act as **stack (LIFO)** or **queue (FIFO)**.
  + Null elements not allowed.
  + Faster than LinkedList for deque operations.
  + Not synchronized.
* **Use Cases:** Implementing stack, queue, or double-ended queue efficiently.

**Example:**

import java.util.ArrayDeque;

import java.util.Deque;

public class ArrayDequeDemo {

public static void main(String[] args) {

Deque<String> deque = new ArrayDeque<>();

deque.add("A"); // addLast

deque.addFirst("B"); // add at front

deque.addLast("C"); // add at end

System.out.println(deque);

System.out.println("Removed first: " + deque.removeFirst());

System.out.println("Removed last: " + deque.removeLast());

}

}

**Output:**

[B, A, C]

Removed first: B

Removed last: C

**3.3 LinkedList as Queue**

* **Package:** java.util
* **Underlying Data Structure:** Doubly linked list
* **Properties:**
  + Implements **Queue and Deque interfaces**.
  + Allows **duplicates** and **null elements**.
  + Maintains **insertion order**.
  + Not synchronized.
* **Use Cases:** Queue or deque operations where **frequent insertions/deletions** occur.

**Example:**

import java.util.LinkedList;

import java.util.Queue;

public class LinkedListQueueDemo {

public static void main(String[] args) {

Queue<Integer> queue = new LinkedList<>();

queue.add(10);

queue.add(20);

queue.add(30);

System.out.println("Queue elements: " + queue);

System.out.println("Removed element: " + queue.poll()); // 10

System.out.println("Head element: " + queue.peek()); // 20

}

}

**Output:**

Queue elements: [10, 20, 30]

Removed element: 10

Head element: 20

**Summary Table of Queue Implementations**

| **Class** | **Underlying DS** | **Order** | **Duplicates** | **Nulls Allowed** | **Thread-Safe** | **Use Case** |
| --- | --- | --- | --- | --- | --- | --- |
| PriorityQueue | Min-heap | Priority (not FIFO) | Yes | No | No | Priority-based processing |
| ArrayDeque | Resizable array | FIFO or LIFO (Deque) | Yes | No | No | Stack/Queue/Deque operations |
| LinkedList | Doubly linked list | FIFO (insertion order) | Yes | Yes | No | Queue/Deque with frequent insert/delete |

💡 **Interview Tip:**

* Know **FIFO vs LIFO behavior**.
* Be ready to explain **PriorityQueue vs ArrayDeque vs LinkedList** differences.
* Mention **null handling** and **thread safety**.

If you want, I can **next cover the Map interface and its main implementing classes in the same detailed manner**, which completes the **core Java collections framework**.

Do you want me to do that?

Perfect! Let’s move on to the **Map interface and its main implementing classes in Java**, explained in **detail for interview purposes**.

**4. Map Interface in Java**

**Definition**

* Map is a **collection that stores key-value pairs**.
* Each key is **unique**, but values **can be duplicated**.
* Not part of Collection hierarchy but part of java.util package.

**Key Methods**

* put(K key, V value) – Adds or updates a key-value pair.
* get(Object key) – Returns the value associated with a key.
* remove(Object key) – Removes the key-value pair.
* containsKey(Object key) – Checks if a key exists.
* containsValue(Object value) – Checks if a value exists.
* size() – Returns number of key-value pairs.
* keySet() – Returns a Set of all keys.
* values() – Returns a Collection of all values.
* entrySet() – Returns a Set of Map.Entry objects (key-value pairs).

**Properties**

| **Property** | **Description** |
| --- | --- |
| Keys | Must be unique |
| Values | Can be duplicate |
| Null Key | Depends on implementation (HashMap/LinkedHashMap: 1 null key, TreeMap: no null key) |
| Null Value | Allowed in most implementations |
| Order | Depends on implementation (HashMap: unordered, LinkedHashMap: insertion order, TreeMap: sorted) |
| Thread Safety | Not synchronized by default; use Collections.synchronizedMap() or ConcurrentHashMap |
| Implementing Classes | HashMap, LinkedHashMap, TreeMap, Hashtable (legacy) |

**Main Implementing Classes of Map**

**4.1 HashMap**

* **Package:** java.util
* **Underlying Data Structure:** Hash table
* **Properties:**
  + **Keys unique**, values can be duplicated.
  + Allows **one null key** and multiple null values.
  + **Unordered** (no guaranteed order).
  + Not synchronized.
  + Fast access: O(1) on average.
* **Use Cases:** Lookup tables, caching, key-value storage.

**Example:**

import java.util.HashMap;

import java.util.Map;

public class HashMapDemo {

public static void main(String[] args) {

Map<Integer, String> map = new HashMap<>();

map.put(1, "Alice");

map.put(2, "Bob");

map.put(3, "Charlie");

map.put(1, "David"); // replaces previous value

System.out.println(map);

}

}

**Output:**

{1=David, 2=Bob, 3=Charlie} // order not guaranteed

**4.2 LinkedHashMap**

* **Package:** java.util
* **Underlying Data Structure:** Hash table + linked list
* **Properties:**
  + Maintains **insertion order** (or access order if specified).
  + Keys unique, values can be duplicate.
  + Allows **one null key** and multiple null values.
  + Not synchronized.
* **Use Cases:** Cache implementations (LRU), predictable iteration order.

**Example:**

import java.util.LinkedHashMap;

import java.util.Map;

public class LinkedHashMapDemo {

public static void main(String[] args) {

Map<Integer, String> map = new LinkedHashMap<>();

map.put(1, "Alice");

map.put(2, "Bob");

map.put(3, "Charlie");

System.out.println(map);

}

}

**Output:**

{1=Alice, 2=Bob, 3=Charlie} // insertion order maintained

**4.3 TreeMap**

* **Package:** java.util
* **Underlying Data Structure:** Red-Black tree (self-balancing BST)
* **Properties:**
  + Keys are **sorted in natural order** (or using a comparator).
  + Keys must be **comparable**; null keys not allowed.
  + Values can be duplicated.
  + Not synchronized.
  + Operations like get/put/remove: O(log n).
* **Use Cases:** Sorted key-value storage, range queries, ordered maps.

**Example:**

import java.util.Map;

import java.util.TreeMap;

public class TreeMapDemo {

public static void main(String[] args) {

Map<Integer, String> map = new TreeMap<>();

map.put(3, "Charlie");

map.put(1, "Alice");

map.put(2, "Bob");

System.out.println(map);

}

}

**Output:**

{1=Alice, 2=Bob, 3=Charlie} // sorted by keys

**4.4 Hashtable (Legacy Class)**

* **Package:** java.util
* **Underlying Data Structure:** Hash table
* **Properties:**
  + Keys unique, values can duplicate.
  + **No null keys or values allowed**.
  + Synchronized (thread-safe).
  + Legacy class, replaced by HashMap/ConcurrentHashMap in modern code.
* **Use Cases:** Thread-safe key-value storage in legacy applications.

**Example:**

import java.util.Hashtable;

import java.util.Map;

public class HashtableDemo {

public static void main(String[] args) {

Map<Integer, String> table = new Hashtable<>();

table.put(1, "Alice");

table.put(2, "Bob");

System.out.println(table);

}

}

**Output:**

{2=Bob, 1=Alice} // order not guaranteed

**Summary Table of Map Implementations**

| **Class** | **Underlying DS** | **Order** | **Null Key** | **Null Value** | **Thread-Safe** | **Use Case** |
| --- | --- | --- | --- | --- | --- | --- |
| HashMap | Hash table | Unordered | 1 | Multiple | No | Fast lookup, general purpose |
| LinkedHashMap | Hash table + linked list | Insertion order | 1 | Multiple | No | Predictable iteration, LRU cache |
| TreeMap | Red-Black tree | Sorted by keys | No | Multiple | No | Sorted maps, range queries |
| Hashtable | Hash table | Unordered | No | No | Yes | Thread-safe, legacy usage |

💡 **Interview Tip:**

* Be ready to **explain differences between HashMap, LinkedHashMap, TreeMap, and Hashtable**.
* Highlight **ordering, null handling, synchronization, and performance**.
* Always give a **small code snippet** for at least HashMap and TreeMap.

If you want, I can now **make a single big visual diagram of the entire Java Collections Framework hierarchy with List, Set, Queue, and Map classes**, which is **extremely handy for interviews**.

Do you want me to do that?