Collection Framework

Before we explore ArrayList, HashSet, HashMap, and other data structures in more detail, it's important to understand that all of these are part of something bigger - the Java Collections Framework.

The Java Collections Framework provides a set of interfaces (like List, Set, and Map) and a set of classes (ArrayList, HashSet, HashMap, etc.) that implement those interfaces.

All of these are part of the java.util package.

They are used to store, search, sort, and organize data more easily - all using standardized methods and patterns

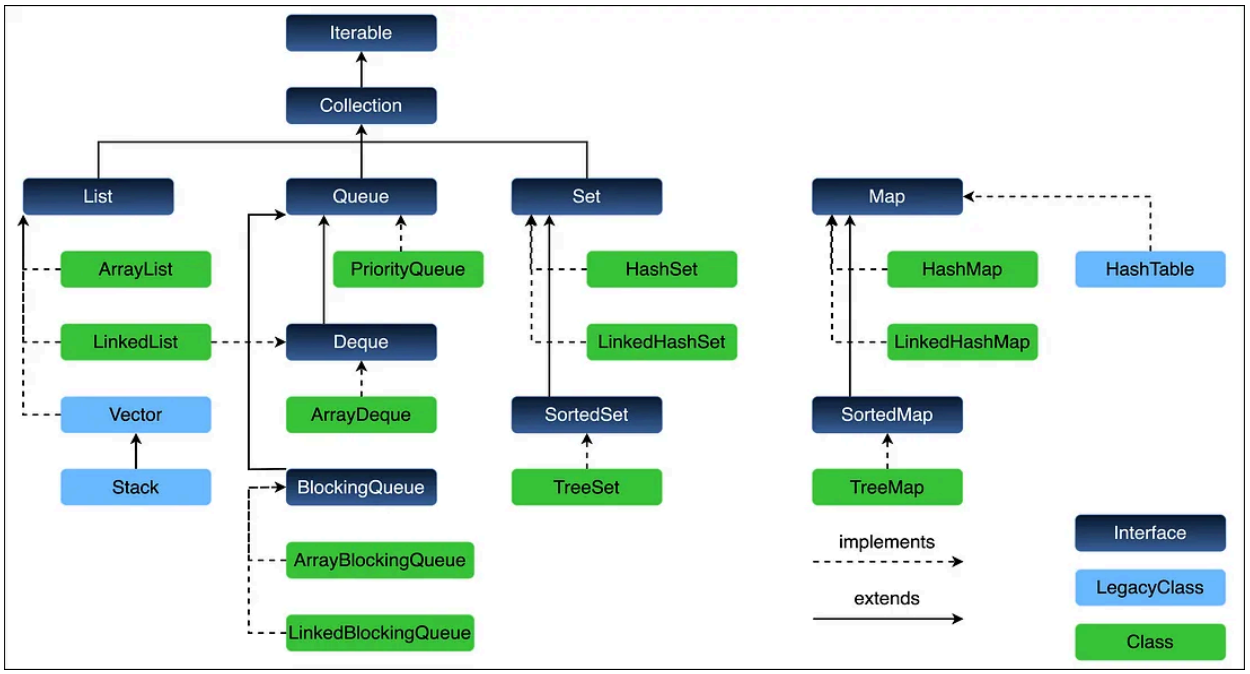
Core Interfaces in the Collections

Here are some common interfaces, along with their classes:

Interface	Common Classes	Description
`List`	`ArrayList`, `LinkedList`	Ordered collection that allows duplicates
`Set`	`HashSet`, `TreeSet`, `LinkedHashSet`	Collection of unique elements
`Map`	`HashMap`, `TreeMap`, `LinkedHashMap`	Stores key-value pairs with unique keys

Collection Interface

The **Collection interface** is the root interface of the **Java Collection Hierarchy**. It is located in the java.util package. The **Collection interface** is not directly implemented by any class. Instead, it is implemented indirectly through its sub-interfaces like List, Queue, and Set.



List:

List is a **subinterface** of `Collection` and used when we want to **store elements in a specific order** and allow **duplicates**.

Key Features:

- ✅ Maintains the order of insertion
- ✅ Allows duplicate elements
- ✅ Supports null values (implementation-dependent)
- ✅ Provides index-based access (get, add, set, remove)
- ✅ Includes ListIterator for forward & backward traversal
- ✅ Implementation Classes:

  - ArrayList - LinkedList - Stack - Vector

📘 ArrayList

ArrayList is an implementing class of the List interface. It stores the element dynamically. It maintains insertion order, allows duplicates and null values.

✅ Key Features:

- Maintains **insertion order**
- Allows **duplicate** elements
- Allows **null** values
- **Dynamic resizing** (no fixed size)
- **Index-based access** (get/set)
- **Not thread-safe** (non synchronized)

✅ Performance:

Operation	Time Complexity	Notes
get(index)	**O(1)**	Fast access
add(element)	**O(1) (amortized)**	Append at end
add(index, e)	**O(n)**	Elements shift right
remove(index)	**O(n)**	Elements shift left

✅ When to Use:

- When fast **random access** is needed.
- When mostly **reading** data.
- When adding/removing at the end**.**

✅ Internal Working:

- Uses a **dynamic array** internally.
- Initial capacity 10 and it grows by **50%** when capacity is full.
- Resizing involves creating a new array and copying elements.

✅ **Example Code:**

```
List<String> names = new ArrayList<>();
names.add("Sachin");
names.add("Virat");
names.add(null);
System.out.println(names.get(1)); // Output: Virat
```

✅ **Important Points to Remember:**

- Not suitable for **multi-threading** (use Collections.synchronizedList() or CopyOnWriteArrayList if needed)
- **Faster** than LinkedList in access
- **Slower** than LinkedList in insertion/deletion in the middle

✅ **One-Line Summary:**
ArrayList = Fast access + Dynamic Array + Allows duplicates & nulls + Not thread-safe

---

📘 LinkedList

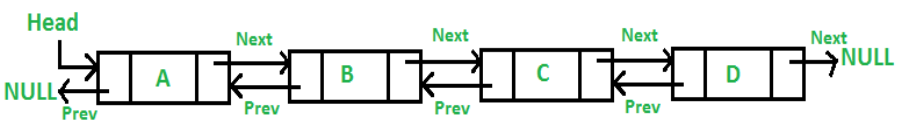
LinkedList is an implementing class of the List and Deque interfaces. It uses a doubly **linked list** to store the element.
It allows **insertion, deletion, and access** of elements efficiently at both ends.

✅ **Key Features:**

- Implements both **List** and **Deque**
- **Not synchronized** (not thread-safe)
- Best for **frequent insertion/deletion**

✅ **Internal Working:**

- Uses a **doubly linked list**: Each node stores:
    - **Data**
    - Reference to the **previous** and **next** nodes



✅ Performance:

Operation	Time Complexity	Notes
get(index)	**O(n)**	Traverses list from start or end
add/remove at end	**O(1)**	Very efficient
add/remove at start	**O(1)**	Very efficient
add/remove at middle	**O(n)**	Must traverse to the position

✅ When to Use:

- When frequent **insertions/deletions** are needed.
- When you need to add/remove at both **beginning and end.**
- Not suitable for **frequent random access.**

✅ Example Code:

```
List<String> names = new LinkedList<>();
names.add("Amit");
names.addFirst("Sachin"); // From Deque interface
names.addLast("Virat");
System.out.println(names); // [Sachin, Amit,Virat]
```

✅ Important Methods:

Method	Description
addFirst()	Add element at beginning
addLast()	Add element at end
removeFirst()	Remove from beginning
removeLast()	Remove from end
get(index)	Access by index (slow)

✅ Drawbacks:

- Slower for **searching/accessing** elements by index.
- **More memory** required (because of node pointers).

✅ One-Line Summary:

LinkedList = Fast insertion/deletion (O(1)) + Doubly linked + Not ideal for search

Difference B/W Array List and Linked List.

Array List	Linked List
Internally it uses a dynamic array.	Internally uses a doubly linked list.
Implements only List interface	Implements List, Deque, and Queue
When we remove/add elements start and middle all elements are shifted.	When we remove/add elements start and middle no need for shifting. uses a doubly linked list.
Best for search-heavy operations	Best for frequent add/remove operations
Fast random access with get(index) → O(1)	Slow access → O(n), must traverse
Adding/removing at end is fast → O(1)*	Adding/removing at end is fast → O(1)

Difference B/W Array List and Vector

ArrayList	Vector
It's not thread-safe	It's Thread-safe (synchronized)
Faster in single-threaded apps	Slower due to synchronization
Grows 50% when full	Grows 100% (doubles size) when full
Introduced in Java 1.2	Older – from Java 1.0 (legacy class)
Uses only Iterator to loop	Can use Iterator and Enumeration
Used in modern apps	Rarely used now, old-style thread safety

Difference B/W Array and Array List

Array	ArrayList
Fixed size (length set when created)	Dynamic size (grows automatically)
Can hold primitive types (int, char, etc.)	Can hold only objects (no primitives directly)
Faster for fixed-size data	More flexible, easier to use when size changes
No built-in methods like add/remove	Provides many useful methods (add, remove, contains,)
Can be multi-dimensional	No direct support for multi-dimensional lists (but can store lists of lists)
Better performance for simple use cases	Slightly slower due to overhead and dynamic resizing

📘 Stack

- A **Stack** is a **Last-In-First-Out (LIFO)** data structure.
- Imagine a stack of the **last plate placed** on top is the **first one you take out**.

✅ **In Java:**

- `Stack` is a **class** that extends `Vector`.
- It provides methods to **push**, **pop**, and **peek** elements.

✅ **Main Operations:**

Operation	Description
`push()`	Add an element to the top
`pop()`	Remove and return the top element
`peek()`	Return the top element without removing it
`isEmpty()`	Check if the stack is empty

✅ **When to Use Stack:**

- Undo operations in editors
- Backtracking algorithms (like maze solving)
- Expression evaluation (parsing)

✅ **Important Note:**

- `Stack` is **synchronized** because it extends `Vector` (thread-safe but slower).
- For better performance in single-threaded apps, use `Deque` implementations like `ArrayDeque` as a stack.

🎯 Quick Revision Note:

> **Stack = LIFO + push/pop/peek + extends Vector + synchronized**

📘 Set Interface

✅ What is Set?

- A **Set** is a **Collection** that **doesn't allow duplicate elements**.
- It models the **mathematical "set"** — where each element is **unique**.

✅ Key Points to Remember:

- **No duplicates allowed**
- **At most one null** (depends on implementation)
- **No indexing** → You **can't access elements by position**
- Used when you want to **store unique elements only**

🔁 Set vs List – Key Difference:

List	Set
Allows duplicates	Doesn't allow duplicates
Maintains order (in List)	May or may not maintain order
Can access by index	No index access

🧠 Easy Use Cases for Set:

- Removing duplicate values from a list
- Maintaining unique usernames, emails, IDs
- Finding distinct words/characters in a string

📘 HashSet

✅ What is HashSet?

- `HashSet` is a class that **implements the Set interface**.
- It uses a **HashTable** (internally a `HashMap`) to store **unique elements**.
- Not thread-safe. For thread-safe version: use
  `Set s= Collections.synchronizedSet(new HashSet( ))`

✅ Key Features of HashSet:

Feature	Description
✅ No duplicates	Stores only unique elements
❌ No order	Does **not maintain insertion order**
✅ One null allowed	Can store **only one null** element
💡 Uses Hashing	For fast access, insertion, and deletion
🔄 Not synchronized	Not thread-safe by default.
🚀 Performance (Time Comple)	Very fast: `add()`, `remove()`, `contains()` → **O(1)**

🔍 Internal Working of HashSet

✅ **When you call `add(element)`:** Uses a **HashMap** internally and **Each element becomes a key and value** is a constant dummy object (`PRESENT`)

1. HashSet calls `map.put(element,PRESENT);`
2. `HashMap`:
    - Calculates **hashCode** of the element.
    - Finds the **bucket (index)** in the array using hash function.
    - If no element exists in that bucket → adds it.
    - If a **collision** occurs (i.e., same hash) → uses **equals()** to check:
        - If equal → duplicate → not added.
        - If not equal → stored in the **same bucket (as linked list or tree)**.

✅ Why Only Unique Elements and Unordered?

- Because keys in `HashMap` must be unique.
- Before adding, `HashMap` checks if the **key already exists** using `equals()` + `hashCode()`.
- Order is based on the **hash code**, not insertion order.
- So, output can be in **any order**.

📘 LinkedHashSet

- `LinkedHashSet` is a subclass of `HashSet`.
- It **maintains insertion order** (unlike `HashSet`).
- Internally it uses a **combination of HashTable + LinkedList**.

✅ Internal Working:

- Inherits from `HashSet` → uses **HashMap** internally but instead of a regular `HashMap`, it uses a **LinkedHashMap**.
- This `LinkedHashMap` maintains a **doubly linked list** to preserve order of insertion.

✅ When to Use LinkedHashSet?

- Want to store unique items
- Need to **preserve insertion order**

Difference Between HashSet and LinkedHashSet

HashSet	LinkedHashSet
❌ Does not maintain insertion order	✅ Maintains insertion order
✅ Faster in performance	⏳ Slightly slower due to linked list
🧠 Internally uses HashMap	🧠 Internally uses LinkedHashMap
🚀 Best for fast, unordered sets	🚀 Best when order matters
🎯 Ideal for general uniqueness check	🎯 Ideal when you need order + uniqueness

---

🌳 TreeSet

- `TreeSet` implements **Sorted Set** .
- It **stores elements in sorted (ascending) order** by default.
- **No duplicates allowed**, like all Sets.
- It is **based on a Red-Black Tree** (a type of self-balancing binary search tree).

✅ Internal Working of TreeSet:

- `TreeSet` uses a `TreeMap` internally.
- Each element becomes a **key** in `TreeMap`, with a dummy value (`PRESENT`).
- **Elements are compared** using:
  - Natural ordering (`Comparable`) or
  - Custom comparator (`Comparator`)

✅ Difference Between HashSet and TreeSet

HashSet	TreeSet
❌ **Does not maintain order**	✅ **Maintains ascending sorted order**
✅ **Faster** (O(1) for add/remove/search)	🐢 **Slower** (O(log n) for add/remove/search)
✅ **Allows 1 null** element	❌ **Does not allow null** (throws exception)
🧠 Uses **HashMap** internally	🌳 Uses **TreeMap (Red-Black Tree)** internally
✅ Good for checking unique element	✅ Good for sorted + unique elements
❌ Cannot use custom sorting	✅ Can sort using `Comparator`
✅	✅

🧾 Queue

- A **Queue** is a **linear data structure** that follows the **FIFO** principle:
  → The element added **first** is removed **first**. **Eg: Ticket counter Line**
- It's part of **`java.util`** package and is an interface.
- Queue also allows storing duplicate values.

✅ Important Methods of Queue Interface:

Method	Description
`add(e)`	Inserts element, throws exception if fails
`offer(e)`	Inserts element, returns false if fails

`remove()`	Removes and returns head, throws exception if empty
`poll()`	Removes and returns head, returns null if empty
`peek()`	Returns head without removing, returns null if empty
`element()`	Returns head without removing, throws exception if empty

✅ Example: Basic Queue with LinkedList

```
Queue<String> queue = new LinkedList<>();
queue.add("A");
queue.add("B");
queue.add("C");
System.out.println(queue);      // [A, B, C]
System.out.println(queue.poll()); // A (removed)
System.out.println(queue);      // [B, C]
```

🎯 PriorityQueue

✅ What is PriorityQueue?

- `PriorityQueue` is a special type of **Queue** where **elements are ordered based on their priority**.
- **Not FIFO** like a normal queue — instead, **lowest or highest priority comes out first**.

✅ Key Features:

Feature	Description
✅ No nulls allowed	Throws `NullPointerException`
🎯 Default is **Min Heap**	Smallest element is removed first (natural order) | **poll( )**
🛠️ Can customize order	Use `Comparator` to create Max Heap or custom sorting
❌ Not thread-safe	Use `PriorityBlockingQueue` for multi-threading
📦 Part of `java.util`	Available from Java 1.5 onward

✅ Internal Working:

- **Internally implemented as a Min Heap (Binary Heap using array)**
- Maintains the **heap property**:
  Parent node is always smaller than child nodes (for min heap).

✅ Examples:

🌱 Default: Min Heap (Natural order)

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(30);
pq.add(10);
pq.add(20);
System.out.println(pq);      // Output might be: [10, 30, 20] (Heap structure)
System.out.println(pq.poll()); // 10 (Smallest removed)
```

🔁 Custom: Max Heap using Comparator

```
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
maxHeap.add(10);
maxHeap.add(30);
maxHeap.add(20);
System.out.println(maxHeap.poll()); // 30 (Highest removed first)
```

✅ When to Use PriorityQueue?

- You need to **process tasks by priority**
- You want a **sorted output** but don't need full control over insertion order
- Ideal for **algorithms** like: Dijkstra's

**** If we want simple FIFO behavior with fast add/remove, use normal Queue (O(1)). But if you need to remove elements based on priority, use PriorityQueue (O(log n)) even though it's slightly slower. ****

✅ ArrayDeque

> `ArrayDeque` is a special kind of **resizable array** by which add or remove elements from **both ends** — front and back.

We can use it like a **Queue (FIFO)** or a **Stack (LIFO)** — just more efficiently!

🧠 Key Points:

- It is **faster than Stack and LinkedList** for **add/remove** operations.
- **No capacity limit** — grows dynamically like ArrayList.
- Does **not allow `null` elements**.
- **Not thread-safe**.

🚀 Common Methods:

Method	What it does
`addLast(e)`	Add at rear
`removeFirst()`	Remove from front
`removeLast()`	Remove from rear
`peekFirst()`	View first element
`peekLast()`	View last element
`offerFirst(e)`	Add at front (no exception)
`offerLast(e)`	Add at end (no exception)
`pollFirst()`	Remove front (no exception)
`pollLast()`	Remove rear (no exception)

📦 Complexity:

- **Time Complexity**: `O(1)` for add/remove at front or rear, `O(n)` for search;
- **Space Complexity**: `O(n)` to store `n` elements in a dynamic array.

🔁 **Use Cases:** Implementing both **Queue** and **Stack** || Browser history (back and forward) || Undo-redo functionality.

ArrayDeque 🆚 LinkedList

ArrayDeque 🧺	LinkedList 🔗
📦 Uses **resizable array** internally	🔗 Uses **doubly linked list** internally
⚡ **Faster** for stack/queue operations	🐢 **Slower** than ArrayDeque for those ops
🚫 Doesn't allow null elements	✅ Allows null elements
💾 **Less memory usage**	🧠 More memory due to node links
🚫 Doesn't support random access	✅ Can access elements via get(index)
🔄 Only Deque interface	🔁 Implements List, Deque, and Queue
🕒 Add/remove at ends in **O(1)** time	🕒 Also O(1) at ends, but slower in practice
💡 Best when using as **Queue or Stack**	💡 Best when frequent **middle insert/remove**
🚫 Not thread-safe	🚫 Not thread-safe

Map interface

- Map<K, V> is a part of **Java Collections Framework** that **maps keys to values**.
- Each key maps to **at most one value** (no duplicate keys).
- It **does not extend Collection** but is part of the collections framework.
- Common implementations:
    - HashMap
    - LinkedHashMap
    - TreeMap
    - ConcurrentHashMap
    - Hashtable

Key Points

✅ Stores data in **key-value pairs**.
✅ **Unique keys only**, but values can be duplicate.
✅ Allows **null values** and one **null keys** (except in some implementations like Hashtable).
✅ Provides efficient **retrieval based on keys**.

Methods in Map interface

Here are **commonly used methods**:

Method	Description
V put(K key, V value)	Inserts or updates a key-value pair.
V get(Object key)	Returns value of the key or null if not present.
V remove(Object key)	Removes the mapping for the key if present.
boolean containsKey(Object key)	Checks if the key exists.
boolean containsValue(Object value)	Checks if the value exists.
Set<K> keySet()	Returns a set view of all keys.
Collection<V> values()	Returns a collection view of all values.
Set<Map.Entry<K, V>> entrySet()	Returns a set view of key-value pairs (Map.Entry).
int size()	Returns the number of key-value mappings.
boolean isEmpty()	Checks if the map is empty.
void clear()	Removes all mappings.

Common Implementations

Implementation	Characteristics
HashMap	Unordered, allows one null key and multiple null values, not thread-safe, fast for most operations.
LinkedHashMap	Maintains insertion order, otherwise same as HashMap.
TreeMap	Sorted by keys (natural ordering or Comparator), no null keys allowed.
Hashtable	Synchronized (thread-safe), no null keys/values, legacy class.
ConcurrentHashMap	Thread-safe, allows concurrent read/write, does not allow null keys/values.

Why use Map?

✅ **Fast lookups** using keys.
✅ Helps in **indexing, caching, frequency counting, building adjacency lists for graphs**.
✅ Suitable for **search-heavy data structures**.

Map.Entry Interface

- Nested interface inside Map.
- Represents a **key-value pair**.
- Commonly used with entrySet() to iterate:

```
public static void findElementThatAppearsOnlyOnce(int[] arr) {
   Map<Integer, Integer> map = new HashMap<>();
   for(Integer i:arr){
       if(!map.containsKey(i)){
           map.put(i,1);
       }
       else {
           map.put(i,map.get(i)+1);
       }
   }
   int num=0;
   Set<Map.Entry<Integer,Integer>>entrySet=map.entrySet();
   for(Map.Entry<Integer,Integer>entry:entrySet){
      if(entry.getValue()==1){
          System.out.print(entry.getKey()+", ");
      }
   }
}

public static void main(String[] args) {
   int[] arr = {3, 4, 3, 1, 2, 4, 5, 5, 7};
   findElementThatAppearsOnlyOnce(arr);
}

OUTPUT:
1, 2, 7
```