Collections Framework Overview in Java

What is a Collection?

- A **Collection** means a group of objects.
- Instead of handling each object separately (like arrays), Java Collections provide a way to **store, retrieve, and manipulate** a group of objects efficiently.

Java Collections Framework (JCF)

What is JCF?

- The **Java Collections Framework (JCF)** is a **unified architecture** that provides a set of classes and interfaces to handle a group of objects.
- It defines **standard ways** to store, retrieve, and manipulate data.
- Instead of writing your own data structures like linked lists or hash tables, JCF gives you **ready-made, efficient, and tested implementations**.

Why Collections Framework?

- **Arrays** are fixed in size → once created, can’t grow/shrink.
- Collections are **dynamic** → can grow or shrink as needed.
- Collections give **built-in methods** for searching, sorting, insertion, deletion, etc.
- Provide **ready-made data structures** like List, Set, Queue, and Map.

How Java Collections Work Internally (Dynamic Nature)

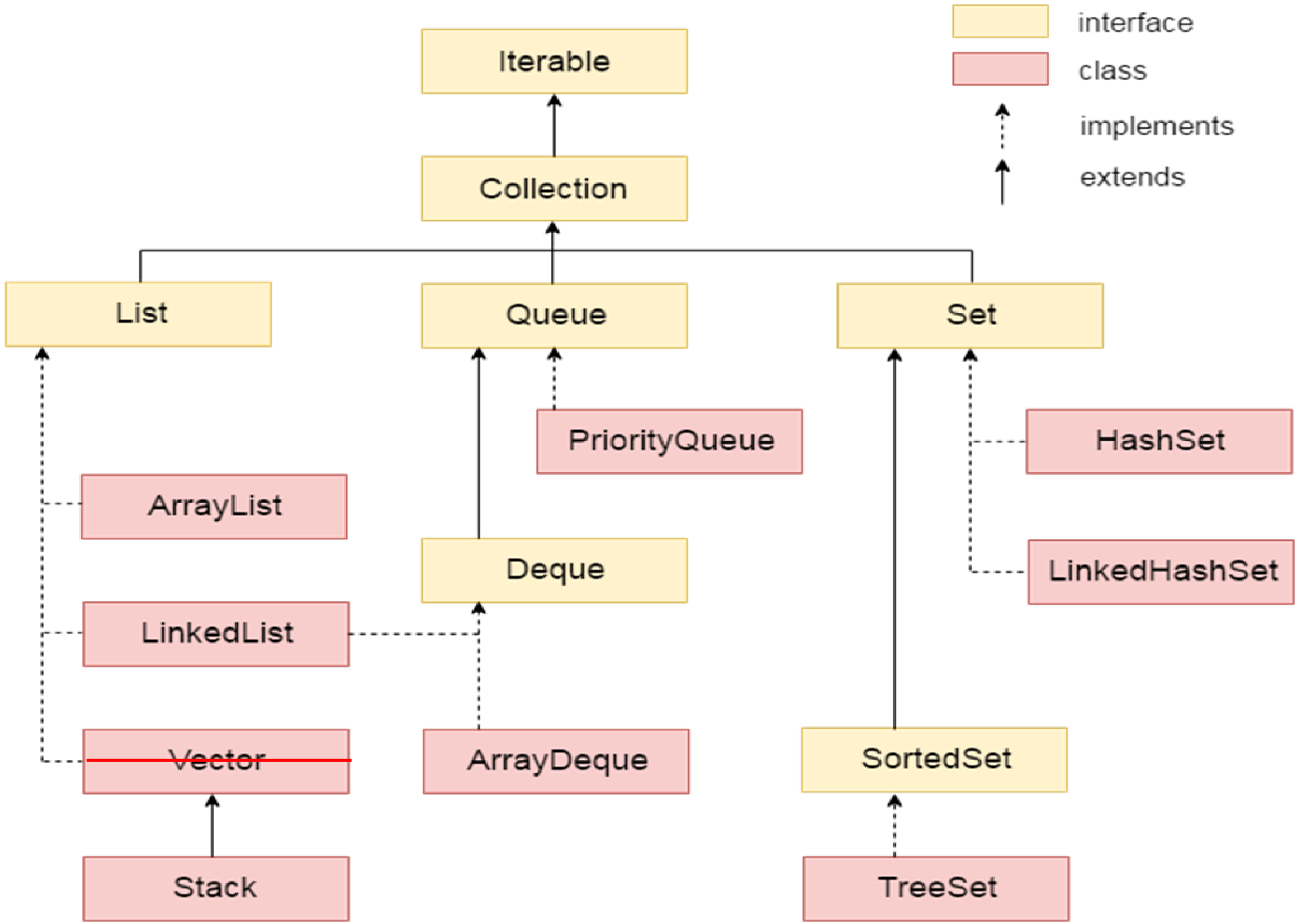
- **Array** → fixed length. Once declared as int arr[10], you can’t add an 11th element.
- **Collections (like ArrayList, HashSet, etc.)** → internally they use **resizable data structures**.
  - When storage is full, they **create a bigger space inside**, copy old elements there, and then continue adding.
  - This “behind the scenes resizing” makes them *look dynamic* to us.

Difference Between Collection and Collections

Collection	**Collections**
Interface	**Class**
Root interface of the framework	Utility/helper class
Defines standard methods (`add`, `remove`, `size`)	Provides algorithms (`sort`, `reverse`, `shuffle`)
Implemented by List, Set, Queue	Works on Collection objects

Hierarchy of Collections Framework

- **Collection Interface** (root for List, Set, Queue)
- **Map Interface** (separate branch for key–value pairs)



interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
~~Vector~~
ArrayDeque
SortedSet
Stack
TreeSet

Main Interfaces:

- **List** → Ordered, allows duplicates.
  - Implementations: ArrayList, LinkedList, Vector, Stack
- **Set** → No duplicates, unordered.
  - Implementations: HashSet, LinkedHashSet, TreeSet
- **Queue** → Used for holding elements in a queue (FIFO).
  - Implementations: PriorityQueue, ArrayDeque
- **Map** → Key–value pairs, no duplicate keys.
  - Implementations: HashMap, LinkedHashMap, TreeMap, Hashtable

Collection interface has methods which are mentioned below:

public boolean add(Object element): It is used to insert an element in this collection.

- **public boolean addAll(Collection c):** It is used to insert the specified collection elements in the invoking collection.
- **public boolean remove(Object element):** It is used to delete an element from this collection.
- **public boolean removeAll(Collection c):** It is used to delete all the elements of specified collection from the invoking collection.
- **public boolean retainAll(Collection c):** It is used to delete all the elements of invoking collection except the specified collection.
- **public int size():** It return the total number of elements in the collection.
- **public void clear():** It removes the total no. of elements from the collection.
- **public boolean contains(Object element):** It is used to search an element.
- **public boolean containsAll(Collection c):** It is used to search the specified collection in this collection.
- **public Iterator iterator():** It returns an iterator.
- **public Object[] toArray():** It converts collection into array.
- **public boolean isEmpty():** It checks if collection is empty.
- **public boolean equals(Object element):** It matches two collections.
- **public int hashCode():** returns the hash code number of the collection.

Example: ArrayList (Most Common)

- Internally, ArrayList uses a **dynamic array** (like a normal array but managed smartly).

- **Default capacity** = 10.
- When it becomes full →
  - A new array is created with **1.5x size** (in Java 8+).
  - Old elements are copied into the new array.
- That's why ArrayList can grow as needed, but resizing is an **expensive operation** (copying takes time).

```java
import java.util.*;

class Example {
    public static void main(String[] args) {
        ArrayList<String> students = new ArrayList<>();

        // Add elements
        students.add("Amit");
        students.add("Priya");
        students.add("Rahul");

        // Access by index
        System.out.println(students.get(1)); // Priya

        // Remove element
        students.remove(0); // removes Amit
        System.out.println(students); // [Priya, Rahul]
    }
}
```

ArrayList Methods -

Some of the methods in array list are listed below:

- **boolean add(Collection c):** Appends the specified element to the end of a list.
- **void add(int index, Object element):** Inserts the specified element at the specified position.
- **void clear():** Removes all the elements from this list.
- **int lastIndexOf(Object o):** Return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
- **Object clone():** Return a shallow copy of an ArrayList.
- **Object[] toArray():** Returns an array containing all the elements in the list.
- **void trimToSize():** Trims the capacity of this ArrayList instance to be the list’s current size.

Key Benefits

- **Reusable data structures** (no need to write from scratch).\
- **Polymorphic behavior** → code written using interfaces works with any implementation.
- **Algorithms ready-made** → Sorting, Searching, etc.
- **Consistent API** → Same method names like add(), remove(), size().

Iterator Interface -

The `Iterator` interface in Java provides a way to iterate over elements in a collection sequentially. It allows traversal of elements one by one and supports operations to retrieve, remove, and check for the existence of elements.

```
public interface Iterator<E> {
    E next();           // Returns the next element in the iteration
    void remove();      // Removes the last element returned by the iterator from the collection
    boolean hasNext();  // Returns true if there are more elements in the collection
}
```

Iterator interface has three methods which are mentioned below:
public boolean hasNext() – This method returns true if iterator has more elements.
public object next() – It returns the element and moves the cursor pointer to the next element.
public void remove() – This method removes the last elements returned by the iterator.
There are three components that extend the collection interface i.e List, Queue and Sets.

```java
import java.util.*;

public class IteratorExample {
    public static void main(String[] args) {
        // Get iterator from the collection
        Iterator<String> it = names.iterator();

        // Traverse the collection
        while (it.hasNext()) {
            String name = it.next();  // get next element
            System.out.println(name);
         }
         }
}
```

Methods:

1. boolean add(Object element)

- **What it is**: Adds a single element into the collection.
- **Why**: To insert new data into a list, set, queue, etc.
- **Where used**: Whenever you want to grow the collection.
- **How**: Returns true if the collection was modified successfully. Some collections (like Set) may return false if the element is already present (because sets don't allow duplicates).
- **Example**:

```
List<String> list = new ArrayList<>();
list.add("Apple");   // true → element added
list.add("Apple");   // true in List, but false in Set
```

2. boolean addAll(Collection c)

- **What it is**: Adds all elements from another collection into the current one.
- **Why**: To merge or copy collections.
- **Where used**: Copying data from one list to another, merging sets, etc.
- **How**: Returns true if the collection changed.
- **Example**:

```
List<String> list1 = new ArrayList<>();
list1.add("A"); list1.add("B");
List<String> list2 = new ArrayList<>();
list2.add("C"); list2.add("D");
list1.addAll(list2);  // list1 = [A, B, C, D]
```

- **3. boolean remove(Object element)**
- **What it is**: Removes one instance of the given element from the collection.
- **Why**: To delete unwanted data.
- **Where used**: Removing an item from cart, dropping a student from list, etc.
- **How**: Returns true if removal happened, false if element not found.
- **Example**:

```
List<String> list = new ArrayList<>();
list.add("A"); list.add("B");
list.remove("A");   // true → "A" removed
list.remove("X");   // false → not found
```

- **4. boolean removeAll(Collection c)**
- **What it is**: Removes all elements that are also in the given collection.
- **Why**: To bulk-remove matching elements.
- **Where used**: Filtering, cleanup tasks.
- **How**: Returns true if any elements were removed.
- **Example**:

```
List<String> list = new ArrayList<>(List.of("A", "B", "C", "D"));
list.removeAll(List.of("B", "D")); // list = [A, C]
```

5. boolean retainAll(Collection c)

- **What it is**: Keeps only the elements present in the given collection. Removes others.
- **Why**: To find **intersection** between collections.
- **Where used**: Filtering students common in two classes, etc.
- **How**: Returns true if collection was modified.
- **Example**:

```
List<String> list = new ArrayList<>(List.of("A", "B", "C", "D"));
list.retainAll(List.of("B", "D", "X"));  // list = [B, D]
```

- **6. int size()**
- **What it is**: Returns number of elements in the collection.
- **Why**: To know the current count.
- **Where used**: Looping, validations.
- **How**: Simple integer return.
- **Example**:

```
List<String> list = new ArrayList<>(List.of("A", "B", "C"));
System.out.println(list.size()); // 3
```

7. void clear()

- **What it is**: Removes all elements.
- **Why**: To empty the collection quickly.
- **Where used**: Resetting cache, reusing same collection.
- **How**: No return value.
- **Example**:

```
list.clear(); // list becomes []
```

- **8. boolean contains(Object element)**
- **What it is**: Checks if element exists in the collection.
- **Why**: Searching.
- **Where used**: To check membership.
- **How**: Returns true or false.
- **Example**:

```
list.contains("A");  // true if "A" exists
```

- **9. boolean containsAll(Collection c)**
- **What it is**: Checks if this collection contains all elements of another.
- **Why**: To check **subset** relationship.
- **Where used**: Permission checks, validations.
- **How**: Returns true or false.
- **Example**:

```
List<String> list = new ArrayList<>(List.of("A", "B", "C"));
list.containsAll(List.of("A", "B")); // true
list.containsAll(List.of("A", "X")); // false
```

- **10. Iterator iterator()**
- **What it is**: Returns an Iterator to loop through collection.
- **Why**: Provides uniform way of traversal across all collection types.
- **Where used**: Instead of using for-loops, especially in Set or Queue.
- **How**: Used with hasNext() and next().
- **Example**:

```
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

- **11. Object[] toArray()**
- **What it is**: Converts collection into an array.
- **Why**: For array-based operations or APIs requiring arrays.
- **Where used**: Interfacing with legacy code.
- **How**: Returns a new array.
- **Example**:

```
Object[] arr = list.toArray();
```

- **12. boolean isEmpty()**
- **What it is**: Checks if collection has zero elements.
- **Why**: To prevent errors (like looping over empty).
- **Where used**: Validations before processing.
- **How**: Returns true if empty.
- **Example**:

```
list.isEmpty(); // true if list.size() == 0
```

- **13. boolean equals(Object element)**
- **What it is**: Compares two collections for equality.
- **Why**: To check if they contain same elements in the same order (for List) or same set of elements (for Set).
- **Where used**: Testing, validation, comparison.
- **How**: Returns true if logically equal.
- **Example**:

```
List<String> l1 = new ArrayList<>(List.of("A", "B"));
List<String> l2 = new ArrayList<>(List.of("A", "B"));
l1.equals(l2); // true
```

- **14. int hashCode()**
- **What it is**: Returns hash code of collection.
- **Why**: Used internally in hashing (like HashSet, HashMap).
- **Where used**: Storing collections in hash-based data structures.
- **How**: Depends on elements inside.
- **Example**:

```
System.out.println(list.hashCode());
```

What is LinkedList?

- LinkedList is a **class in Java** that implements the **List** and **Deque** interfaces.
- It stores **elements in a sequence of nodes**, where each node contains:
  - **Data** (the actual value)
  - **Pointer/Reference** to the **next node**
  - **Pointer/Reference** to the **previous node** (in case of doubly linked list, which Java's LinkedList is)
- Unlike **ArrayList**, LinkedList does **not store elements in contiguous memory**.

Key Features

Feature	**Description**
Dynamic size	Can grow or shrink at runtime.
Insertion/Deletion	Fast at start, end, or middle (if node reference is known).
Access time	Slow for random access (`O(n)`), fast for sequential access.
Duplicates allowed	Yes
Null elements	Yes, can store multiple `null`s.
Thread safety	Not synchronized (not thread-safe by default).

Constructors

1. LinkedList<String> list1 = new LinkedList<>(); // empty linked list

2. LinkedList<String> list2 = new LinkedList<>(existingCollection);
   // copy from another collection

Common Methods

From List interface:

- **add(E e)** → adds element at end
- **add(int index, E e)** → adds element at specified position
- **get(int index)** → returns element at position
- **set(int index, E e)** → replaces element
- **remove(int index)** → removes element at index
- **size()** → returns number of elements

E stands for "Element"
*E is a **generic type parameter**.*
*It represents the **type of elements the collection will hold**.*

From Deque / Queue interface:

- **addFirst(E e)** → add at beginning
- **addLast(E e)** → add at end
- **removeFirst()** → remove first element
- **removeLast()** → remove last element
- **peek()** → retrieve head without removing
- **poll()** → retrieve and remove head

When to Use LinkedList

- **Use it when:**
  - You need **frequent insertions/deletions** at the **beginning or middle**.
  - You are using it as a **Queue** or **Deque** (FIFO/LIFO operations).
- **Avoid it when:**
  - You need **fast random access**, because get(i) is O(n).

Example

```
LinkedList<String> list = new LinkedList<>();

// Adding elements
list.add("A"); // [A]
list.addLast("B"); // [A, B]
list.addFirst("C"); // [C, A, B]
list.add(1, "D"); // [C, D, A, B]

// Accessing elements
System.out.println(list.get(2)); // A

// Removing elements
list.removeFirst(); // removes C -> [D, A, B]
list.remove(1); // removes A -> [D, B]

// Iterating
for(String s : list) {
   System.out.print(s + " ");
}
// Output: D B
```

Comparison with ArrayList

Feature	LinkedList	ArrayList
Underlying structure	Doubly Linked List	Dynamic Array
Access by index	Slow `O(n)`	Fast `O(1)`
Insert/Delete at start/middle	Fast `O(1)` if node known	Slow `O(n)`
Memory usage	Higher (extra pointers)	Less
Best use case	Frequent insertions/deletions	Frequent reads/random access

What is Stack?

- **Stack** is a **class in java.util** that represents a **last-in-first-out (LIFO) stack of objects**.
- The **last element pushed onto the stack is the first one to be popped**.
- Conceptually, it works like a **real-life stack of plates**: you put plates on top and remove from the top.

Key Features

Feature	**Description**
LIFO	Last-in-first-out order.
Dynamic size	Can grow as needed.
Thread-safe	Because it extends `Vector` (synchronized methods).
Null allowed	Yes, `null` can be pushed.
Methods	Provides **push, pop, peek, search, empty**.

Important Methods

Method	**Description**
`push(E item)`	Adds element to the top of the stack
`pop()`	Removes and returns the top element
`peek()`	Returns the top element **without removing**
`empty()`	Returns `true` if stack is empty
`search(Object o)`	Returns **1-based position** from top, -1 if not found

Example

```
import java.util.Stack;

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

        // Push elements
        stack.push("A"); // Stack: [A]
        stack.push("B"); // Stack: [A, B]
        stack.push("C"); // Stack: [A, B, C]

        System.out.println("Top element: " + stack.peek()); // C

        // Pop element
        System.out.println("Popped: " + stack.pop()); // C
        System.out.println("Stack after pop: " + stack); // [A, B]

        // Search
        System.out.println("Position of A: " + stack.search("A")); // 2
        System.out.println("Is stack empty? " + stack.empty()); // false
    }
}
```

Stack vs ArrayList vs LinkedList

Feature	Stack	ArrayList	LinkedList
Order	LIFO	Random	Sequential
Access	Top only	Any index	Any index
Thread-safe	Yes (because Vector)	No	No
Insert/Delete	Push/Pop	Add/Remove anywhere	Add/Remove anywhere
Underlying structure	Vector (dynamic array)	Dynamic array	Doubly linked list

Important Notes

- Stack is **synchronized**, so it's **thread-safe**, but slower than Deque implementations.
- Modern Java recommends using **ArrayDeque** as a stack instead of Stack because:
- Stack is **legacy**.
- ArrayDeque is **faster** and cleaner.

```
ArrayDeque<Integer> stack = new ArrayDeque<>();
stack.push(10);
stack.pop();
```

Queue Interface (java.util.Queue)

- **What it is**:
  A **Queue** is a collection designed to hold elements **before processing**, usually in **FIFO (First In First Out)** order.
- **Where it is used**:
- Task scheduling (CPU scheduling, printer jobs).
- Messaging (like chat queues, request queues).
- Buffers in data streaming.

Key Methods in Queue:

Method	Description
`boolean add(E e)`	Inserts element at end. Throws exception if capacity full.
`boolean offer(E e)`	Inserts element at end. Returns `false` if capacity full.
`E remove()`	Removes and returns head (first element). Throws exception if empty.
`E poll()`	Removes and returns head, returns `null` if empty.
`E element()`	Returns head without removing. Throws exception if empty.
`E peek()`	Returns head without removing, returns `null` if empty.

Example Use Case:

```
Queue<String> q = new LinkedList<>();
q.offer("Task1");
q.offer("Task2");
System.out.println(q.poll());   // Task1 (FIFO)
System.out.println(q.peek());   // Task2 (next element)
```

Deque Interface (java.util.Deque)

- **What it is**:
  Deque = Double Ended Queue (pronounced "deck").
  You can insert/remove elements **from both ends** (head & tail).
  It can act as **Queue (FIFO)** or **Stack (LIFO)**.
- **Where it is used**:
- Implementing **stacks** and **queues**.
- Browser history (back & forward navigation).
- Palindrome checking.
- Sliding window problems in algorithms.

Key Methods in Deque:

Method	**Description**
`void addFirst(E e)`	Inserts at head.
`void addLast(E e)`	Inserts at tail.
`E removeFirst()`	Removes and returns head.
`E removeLast()`	Removes and returns tail.
`E getFirst()`	Returns head without removing.
`E getLast()`	Returns tail without removing.
`boolean offerFirst(E e)`	Inserts at head, returns `false` if fails.
`boolean offerLast(E e)`	Inserts at tail, returns `false` if fails.
`E pollFirst()`	Removes head, returns `null` if empty.
`E pollLast()`	Removes tail, returns `null` if empty.

Example Use Case:

```
Deque<String> dq = new ArrayDeque<>();
dq.addFirst("A");
dq.addLast("B");
dq.addFirst("Start");
dq.addLast("End");
System.out.println(dq); // [Start, A, B, End]

dq.removeLast(); // removes "End"
dq.removeFirst(); // removes "Start"
System.out.println(dq); // [A, B]
```

PriorityQueue (class under Queue Interface)

- A **PriorityQueue** is a special kind of **Queue** (part of the Java Collections Framework) where the elements are **ordered by priority**, not just FIFO (First-In-First-Out).
- By default, it arranges elements in **natural ordering** (for numbers: ascending, for strings: alphabetical).
- But you can also define your own **custom ordering** using a Comparator.

Key Points

- **Implements**:
  - Queue<E> interface
  - Serializable and Iterable
- **Ordering**:
  - By default → Natural order (Comparable).
  - Custom → Comparator passed in constructor.
- **Null elements** → **Not allowed**.
- **Duplicates** → Allowed.
- **Underlying data structure** → **Heap (binary heap)**.
- **Not thread-safe** → Use PriorityBlockingQueue for multithreading.

Common Methods

- boolean add(E e) → Inserts element based on priority.
- boolean offer(E e) → Same as add, but won’t throw exception if capacity is restricted.
- E peek() → Retrieves head (highest priority element) **without removing**.
- E poll() → Retrieves and **removes** head.
- boolean remove(Object o) → Removes specific element.
- int size() → Returns number of elements.

Example 1 – Natural Ordering (Numbers)

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

pq.add(30);
pq.add(10);
pq.add(20);

System.out.println(pq);      // [10, 30, 20] (internal heap order, not sorted fully)
System.out.println(pq.poll()); // 10 (smallest comes out first)
System.out.println(pq.poll()); // 20
System.out.println(pq.poll()); // 30
```

Example 2 – Custom Ordering (Max Heap)

```
PriorityQueue<Integer> pq = new
PriorityQueue<>(Comparator.reverseOrder());

pq.add(30);
pq.add(10);
pq.add(20);

while(!pq.isEmpty()) {
    System.out.println(pq.poll());
}
```

ArrayDeque in Java (class of Deque Interface)

- **Definition**
- ArrayDeque (short for **Array Double Ended Queue**) is a **resizable array-based implementation** of the Deque interface.
- It allows you to **insert and remove elements from both ends** (front and rear).
- Unlike a normal Queue (FIFO) or Stack (LIFO), an ArrayDeque can act as **both queue and stack**.

Key Points

- **Implements**: Deque<E> → so supports both queue & stack operations.
- **Null elements** → **Not allowed**.
- **Resizable** → No capacity restrictions (unlike ArrayBlockingQueue).
- **Faster than Stack & LinkedList** for stack/queue operations (no synchronization overhead).
- **Not thread-safe** → Use ConcurrentLinkedDeque if multiple threads access it.
- **Underlying Data Structure** → **Resizable circular array**.

Example 1 – Double-ended Operations

```
ArrayDeque<String> deque = new ArrayDeque<>();
deque.addFirst("Front");
deque.addLast("Rear");

System.out.println(deque.peekFirst()); // Front
System.out.println(deque.peekLast());  // Rear
```

What is HashSet?

- **HashSet** is a **class in java.util** that implements the **Set** interface.
- It **stores unique elements — no duplicates allowed**.
- Elements are stored in a **hash table** internally (uses hashCode() and equals() for uniqueness).
- **No guaranteed order** — elements are **not stored in insertion order**.

Key Features

Feature	**Description**
Unique elements	Duplicate values are ignored.
Null allowed	Only **one null** element allowed.
No order	Does not preserve insertion order (use `LinkedHashSet` for that).
Underlying structure	Hash table (uses hash codes)
Fast operations	`add`, `remove`, `contains` are **O(1)** on average.
Not synchronized	Not thread-safe (use `Collections.synchronizedSet()` if needed).

Constructors

1. HashSet<String> set1 = new HashSet<>(); // Empty HashSet
2. HashSet<String> set2 = new HashSet<>(existingCollection);

   // Copy from another collection
3. HashSet<String> set3 = new HashSet<>(16, 0.75f);

   // Initial capacity & load factor

- **Load factor**: Threshold to increase capacity (default 0.75)
- **Initial capacity**: Number of buckets (default 16)

Common Methods

Method	Description
`add(E e)`	Adds element (ignored if duplicate)
`remove(Object o)`	Removes the element
`contains(Object o)`	Checks if element exists
`size()`	Returns number of elements
`isEmpty()`	Checks if set is empty
`clear()`	Removes all elements
`iterator()`	Returns iterator to traverse the set

Example

```java
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();

        // Adding elements
        set.add("A");
        set.add("B");
        set.add("C");
        set.add("A"); // Duplicate, ignored

        System.out.println("HashSet: " + set);
            // Order not guaranteed

        // Checking element
        System.out.println("Contains B? " +
            set.contains("B")); // true

        // Removing element
        set.remove("B");
        System.out.println("After removal: " +
            set);

        // Iterating
        for(String s : set) {
            System.out.print(s + " ");
        }
    }
}
```

LinkedHashSet

- **LinkedHashSet** is a **class in java.util** that implements the **`Set`` interface**.
- It **stores unique elements** like HashSet.
- **Maintains insertion order** — elements are returned in the **order they were added**.
- Internally, it uses a **hash table + linked list**.
- So basically, it’s a **combination of HashSet (for uniqueness) and LinkedList (for order)**.

Key Features

Feature	Description
Unique elements	Duplicate elements are not allowed.
Maintains order	Preserves **insertion order**.
Null allowed	Only one `null` allowed.
Underlying structure	Hash table + doubly linked list
Performance	Slightly slower than `HashSet` for add/remove due to linked list overhead.
Not synchronized	Not thread-safe (use `Collections.synchronizedSet()` if needed).

Constructors

```
LinkedHashSet<String> set1 = new LinkedHashSet<>(); // empty set

LinkedHashSet<String> set2 = new LinkedHashSet<>(existingCollection);
// copy from another collection

LinkedHashSet<String> set3 = new LinkedHashSet<>(16, 0.75f);
// initial capacity & load factor
```

Common Methods

Method	**Description**
`add(E e)`	Adds element (ignored if duplicate)
`remove(Object o)`	Removes element
`contains(Object o)`	Checks if element exists
`size()`	Returns number of elements
`isEmpty()`	Checks if set is empty
`clear()`	Removes all elements
`iterator()`	Returns iterator to traverse in **insertion order**

Example

```
import java.util.LinkedHashSet;

public class LinkedHashSetExample {
    public static void main(String[] args) {
        LinkedHashSet<String> set = new LinkedHashSet<>();

        // Adding elements
        set.add("A");
        set.add("C");
        set.add("B");
        set.add("A"); // Duplicate ignored

        System.out.println("LinkedHashSet: " + set); // [A, C, B] - preserves insertion order

        // Checking element
        System.out.println("Contains B? " + set.contains("B")); // true

        // Removing element
        set.remove("C");
        System.out.println("After removal: " + set); // [A, B]

        // Iterating
        for(String s : set) {
            System.out.print(s + " ");
        }
    }
}
```

TreeSet

- **TreeSet** is a **class in java.util** that implements the **SortedSet** interface.
- It **stores unique elements** (like HashSet) but also **keeps them sorted** in **ascending order by default**.
- Internally, it uses a **Red-Black tree** (self-balancing binary search tree).

- So basically: **TreeSet = Unique elements + Sorted order**

Key Features

Feature	**Description**
Unique elements	No duplicates allowed
Sorted order	Elements automatically sorted (ascending by default)
Null allowed	Only **one null** allowed in Java 7+, null not allowed in Java 8+ for comparison
Underlying structure	Red-Black tree (self-balancing BST)
Performance	`add`, `remove`, `contains` → **O(log n)**
Not synchronized	Not thread-safe (use `Collections.synchronizedSortedSet()` if needed)

Constructors

```
TreeSet<String> set1 = new TreeSet<>(); // empty set

TreeSet<String> set2 = new TreeSet<>(existingCollection);
// copy from another collection

TreeSet<Integer> set3 = new TreeSet<>(Comparator.reverseOrder());
// custom comparator (descending order)
```

Common Methods

Method	**Description**
`add(E e)`	Adds element (ignored if duplicate)
`remove(Object o)`	Removes element
`contains(Object o)`	Checks if element exists
`first()`	Returns the **smallest** element
`last()`	Returns the **largest** element
`headSet(E toElement)`	Returns elements **less than** `toElement`
`tailSet(E fromElement)`	Returns elements **greater than or equal to** `fromElement`
`subSet(E from, E to)`	Returns elements in **range** `[from, to)`
`size()`	Returns number of elements
`iterator()`	Returns iterator in **sorted order**

Example

```
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();

        // Adding elements
        set.add(50);
        set.add(20);
        set.add(40);
        set.add(10);
        set.add(20); // Duplicate ignored

        System.out.println("TreeSet: " + set); // [10, 20, 40, 50] - sorted automatically

        // Access first and last
        System.out.println("First: " + set.first()); // 10
        System.out.println("Last: " + set.last());   // 50

        // Range example
        System.out.println("HeadSet(40): " + set.headSet(40)); // [10, 20]
        System.out.println("TailSet(20): " + set.tailSet(20)); // [20, 40, 50]

        // Iterating
        for(int num : set) {
            System.out.print(num + " ");
        }
    }
}
```

When to Use TreeSet

- **You need unique elements**.
- **You need them sorted automatically**.
- **You want range operations** (subSet, headSet, tailSet) easily.
- **You don't care about insertion order**, only sorted order.