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)

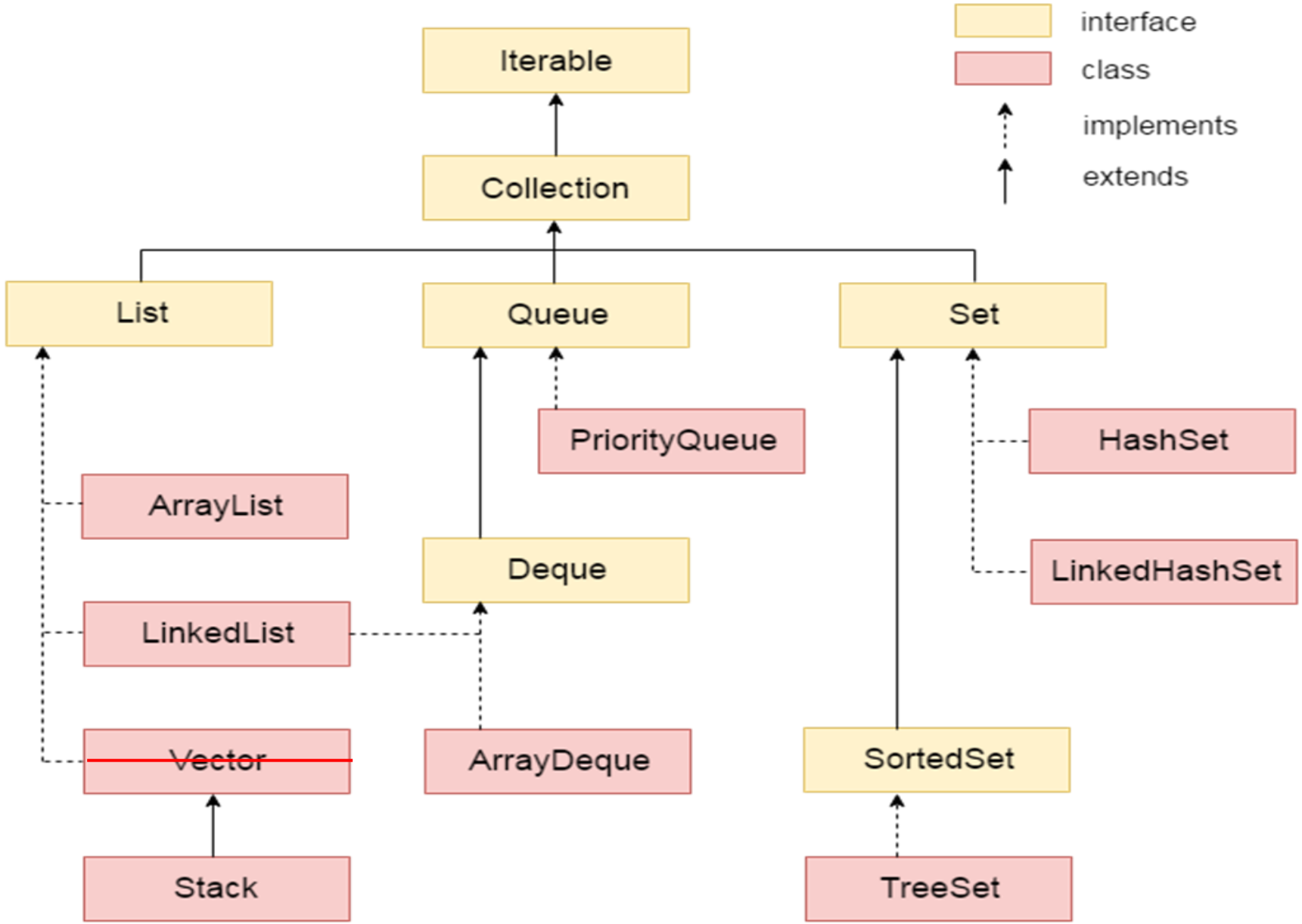
- **Array** → fixed length. Once declared as int arr[10], you can't add an 11th element.
- **Collections (like ArrayList, HashSet, etc.)** → internally they use **resizable data structures**.
  - When storage is full, they **create a bigger space inside**, copy old elements there, and then continue adding.
  - 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).

```
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.

```
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());
```

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
```