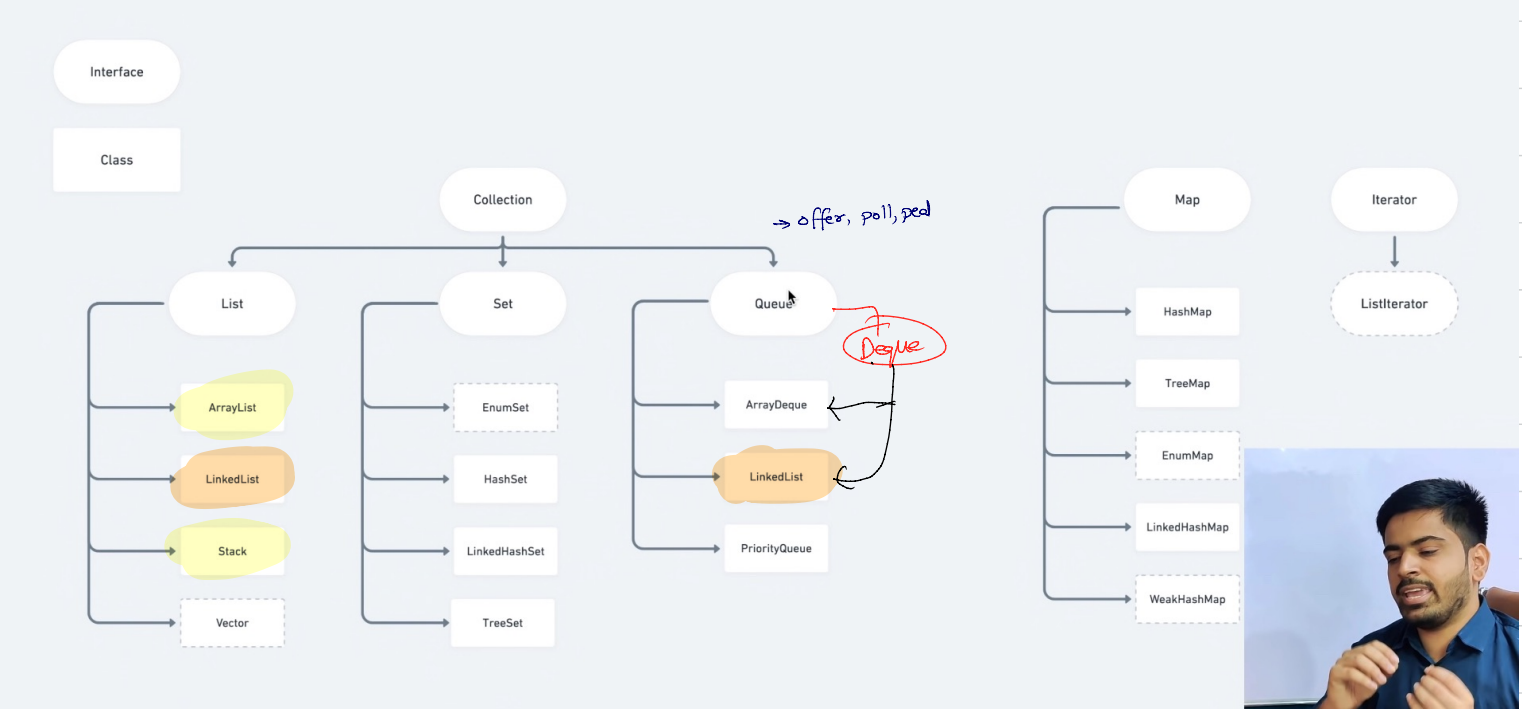
Collections



→ isEmpty()
→ size()
→ contains()
→ Iterator()
→ toArray()

Common

→ Insertion

add() → ArrayList, Queue, Set, LinkedList, → offer()
push() → Stack
put() → Map

→ Deletion

pop() → Stack
remove() → ArrayList, Queue, Set, LinkedList, Map
poll() → queue

peek() → stack, queue → Top element

Collections Class

→ import java.util.Collections;
→ useful for collection framework DS;

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
list.add(4);
```

Min Element :- Collections.min(list);
Max Element :- Collections.max(list);
frequency :- Collections.frequency(list, 2);

Sorting :- Collections.sort(list);
→ Collections.sort(list, Comparator.reverseOrder());

Reverse — Collections.reverse(list);

frequency — to find the occurences of particular element.

Collections.frequency(list, 'a');

Binary Search — Collections.binarySearch(list, 3);

fill — Collections.fill(list, 1);

copy — Collections.copy(dest, src);

swap — Collections.swap(list, 0, 2); (index)

replaceAll — Collections.replaceAll(list, "Ram", "Sankara"); (from → to)

unmodifiableList — List<String> unmodifiableList = Collections.unmodifiableList(list);

emptyList — Collections.emptyList();

disjoint — Collections.disjoint(list1, list2);

https://www.linkedin.com/in/kapilyadav22

List

.length() → for strin

Java.util.List;

→ ArrayList :- dynamic size array

Syntax :- ArrayList<String> names = new ArrayList<>();

→ names.add("Kapil"); names.add(index, value);

In Java, an ArrayList is a dynamic array implementation that allows you to add, remove, and access elements easily. You can use the ArrayList class from the java.util package. Here's a simple example:

→ Default initial size is 10 and it grows double when it get completely filled.

→ names.get(index);

O(N) → names.remove("Kapil"); | → names.remove(Integer.valueOf(value)); ← for integer values in Arraylist

O(N) → names.remove(index); | → names.remove(index);

→ ArrayList<String> List2 = new ArrayList<>();
List2.add("Ray");
List2.add("Abhishek");

→ names.addAll(List2);
↳ to add all elements of List2 in names.

→ names.size(); → No of elements

→ names.clear(); //empty the names list

→ names.set(index, value); // update the value at particular index. O(1)

→ names.contains(value); // checks if value is present or not

→ for(int i=0; i<names.size(); i++)
{ Sy.out.println(names.get(i));
}

→ for(int element : names)
{ s.o.println(element);
}

```
Iterator<Integer> it = list.Iterator();
while(it.hasNext()) {
    s.o.println("iterator" + it.next());
}
```

C++ Stl

```
vector<int> data;
data.push_back(value);
data.pop_back();
data.size();
data.clear(); //empty the vector
data.insert(index, value);

vector<int> numbers(size, initial value);
find(numbers.begin(), numbers.end(), valueToFind);
numbers.erase(numbers..begin() + 2);
```

```
boolean contains = list.contains(20);
int index = list.indexOf(20);
int lastIndex = list.lastIndexOf(20);
```

→ first index of 20

→ last index containing 20;

Key Difference:

- **Arrays**: Use `.length` → `arr.length`
- **ArrayList**: Use `.size()` → `list.size()`

```
Object[] arr = list.toArray(); // Converts to Object array
Integer[] arr2 = list.toArray(new Integer[0]); // Converts to
Integer array
```

```
list.retainAll(Arrays.asList(10, 20, 30)); // Keeps only these elements
list.removeAll(Arrays.asList(10, 20)); // Removes 10 and 20
```

Stack

```
import java.util.Stack;

Stack<String> st = new Stack<>();
```

→ `st.push("kapil");` } push in LIFO order
→ `st.push("Vimal");`

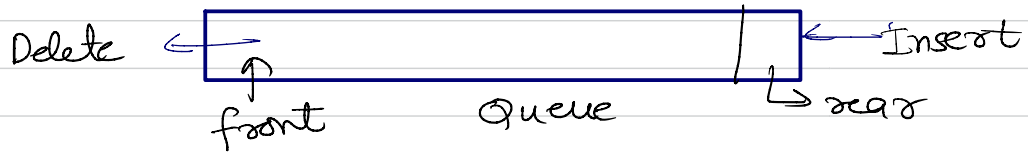
→ `st.peak();` // top element

→ `st.pop();` // pop top element

In C++ Stl

→ `st.push();` // insert in LIFO manner
→ `st.pop();` // pop the top element
→ `st.top();` // get the top element
→ `st.size();` // get the size of stack

Queue



→ Queue is an Interface.
→ import java.util.Queue;

Queue can be create using ArrayDeque, LinkedList and PriorityQueue

1. Create Queue using LinkedList

use LinkedList or arrayDeque

```
Queue<Integer> que = new LinkedList<>();
```

→ Insert in queue :- There are two ways add() and offer()

```
que.add(10);
que.offer(10);
```

```
// C++ stl
que.push(10);
```

→ Both are same for unbounded queue

Key Difference:

- The primary difference between `add()` and `offer()` is how they handle cases where the queue has a capacity restriction, and adding an element is not possible.
- `add()` throws an exception (`IllegalStateException`) if the element cannot be added due to capacity restrictions.
- `offer()` returns a boolean (`true` if added successfully, `false` if not) without throwing an exception.

When working with unbounded queues or queues with sufficient capacity, the distinction between `add()` and `offer()` might not be critical. However, in scenarios where capacity constraints matter, using `offer()` allows you to handle the failure gracefully without throwing an exception.

c++ stl

Remove elements from Queue :-

que.poll(); //removes front element

que.pop();

que.peak(); } returns null if queue is empty

que.element(); } front element — If the queue is empty, it throws a **NoSuchElementException**.

que.front();

Summary of Queue Methods

Method	Description
`add(E e)`	Inserts an element (throws exception if full)
`offer(E e)`	Inserts an element (returns false if full)
`poll()`	Retrieves and removes head (returns `null` if empty)
`remove()`	Retrieves and removes head (throws exception if empty)
`peek()`	Retrieves head without removing (returns `null` if empty)
`element()`	Retrieves head without removing (throws exception if empty)
`size()`	Returns the number of elements
`isEmpty()`	Checks if the queue is empty

java.util.LinkedList;

→ LinkedList using List;

```
List<String> names = new LinkedList<>();
```

→ Rest operations will be same as List (ArrayList which we used above)
→ add, addAll, get, contains, clear, set.

NOTE:- We can also use LinkedList directly from java.util.LinkedList.
It contains all the methods of list and queue.
LinkedList implements both Queue and List Interface.

NOTE:- It's a good idea to store LinkedList in type Queue or List, so that we can get more Code flexibility.

Create a Linked List

1. Add Elements

- add(element) → appends element at end of the list.
- add(index, element) → inserts the element at particular index.
- addFirst(element) → Insert at beginning.
- addLast(element) → append at end.

Removing elements:-

- remove() :- Remove first element.
- remove(int index) : Remove the element at particular position
- remove(element): Remove first occurence of element.
- removeFirst() : Removes and returns the first element.
- removeLast() : Removes and returns the last element.

Accessing Elements:-

- get(index) :- returns element at particular position
- getFirst() :- returns the first element in the list.
- getLast() :- returns the last element in the list.

Priority Queue

→ import java.util.PriorityQueue;

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

OR

```
Queue<Integer> pq = new PriorityQueue<>();
//By default it uses minheap.
```

→ uses same methods as Queue – offer, poll, peak, size

→ Using pq as Maxheap

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

---

1. Using Pair (Entry) from AbstractMap.SimpleEntry

java Copy

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Map.Entry<Integer, String>> pq = new PriorityQueue<>(Compar

        pq.offer(new AbstractMap.SimpleEntry<>(3, "C"));
        pq.offer(new AbstractMap.SimpleEntry<>(1, "A"));
        pq.offer(new AbstractMap.SimpleEntry<>(2, "B"));

        while (!pq.isEmpty()) {
            Map.Entry<Integer, String> pair = pq.poll();
            System.out.println(pair.getKey() + " -> " + pair.getValue());
        }
    }
}
```

Output:

Copy

```
1 -> A
2 -> B
3 -> C
```

2. Using Custom Pair Class

java Copy

```java
import java.util.*;

class Pair {
    int key;
    String value;

    public Pair(int key, String value) {
        this.key = key;
        this.value = value;
    }
}

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Pair> pq = new PriorityQueue<>(Comparator.comparingInt(a ->

        pq.offer(new Pair(3, "C"));
        pq.offer(new Pair(1, "A"));
        pq.offer(new Pair(2, "B"));

        while (!pq.isEmpty()) {
            Pair pair = pq.poll();
            System.out.println(pair.key + " -> " + pair.value);
        }
    }
}
```

3. Using int[] for Pair Storage

java Copy

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -

        pq.offer(new int[]{3, 100});
        pq.offer(new int[]{1, 200});
        pq.offer(new int[]{2, 300});

        while (!pq.isEmpty()) {
            int[] pair = pq.poll();
            System.out.println(pair[0] + " -> " + pair[1]);
        }
    }
}
```

Best Approach?

- **If readability matters:** Use `SimpleEntry` or a **custom class**.
- **If performance matters:** Use an **int[] pair**, as it avoids object overhead.
- **If the pair is immutable and built-in support is needed:** Use `Map.Entry<Integer, String>`.

Max heap

```
import java.util.*;

class Pair {
    int key, value;
    public Pair(int key, int value) {
        this.key = key;
        this.value = value;
    }
}

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Pair> maxHeap = new PriorityQueue<>((a, b) -> Integer.compare(b.key, a.key));

        maxHeap.offer(new Pair(3, 100));
        maxHeap.offer(new Pair(1, 200));
        maxHeap.offer(new Pair(2, 300));

        while (!maxHeap.isEmpty()) {
            Pair pair = maxHeap.poll();
            System.out.println(pair.key + " -> " + pair.value);
        }
    }
}
```

```
PriorityQueue<Pair> minHeap = new PriorityQueue<>((a, b) -> Integer.compare(a.value, b.value));
PriorityQueue<Pair> maxHeap = new PriorityQueue<>((a, b) -> Integer.compare(b.value, a.value));
```

Custom Comparator in PQ.

2. Which One is More Efficient?

1. **For single-use and concise code → Use a lambda expression**

lambda expression for comparator

java Copy

```
PriorityQueue<Pair> pq = new PriorityQueue<>((a, b) -> {
    if (a.first != b.first) return Integer.compare(a.first, b.first);
    return Integer.compare(a.second, b.second);
});
```

✅ **Best when used once in a function.**

2. **For multiple comparisons in different parts of the code → Use a separate Comparator class**

java Copy

```
class PairComparator implements Comparator<Pair> {
    @Override
    public int compare(Pair a, Pair b) {
        if (a.first != b.first) return Integer.compare(a.first, b.first);
        return Integer.compare(a.second, b.second);
    }
}
PriorityQueue<Pair> pq = new PriorityQueue<>(new PairComparator());
```

✅ **Best if the comparator is used in multiple places.**

3. **For cleaner and more readable code → Use Comparator.comparingInt()**

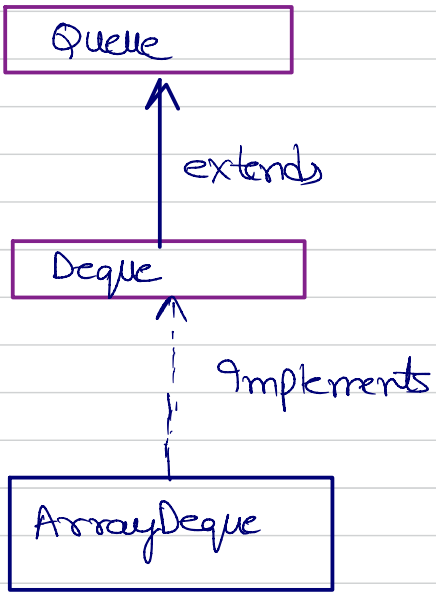
java Copy

```
PriorityQueue<Pair> pq = new PriorityQueue<>(
    Comparator.comparingInt((Pair p) -> p.first)
              .thenComparingInt(p -> p.second)
);
```

✅ **Most optimized and modern approach.**

ArrayDeque

→ Implements Deque Interface which extends Queue Interface.
→ java.util.ArrayDeque
→ Double ended Queue



→ `ArrayDeque<Integer> adq = new ArrayDeque<>();`

```
adq.offer(value);
adq.offerFirst(value);
adq.offerLast(value); // same as offer
```

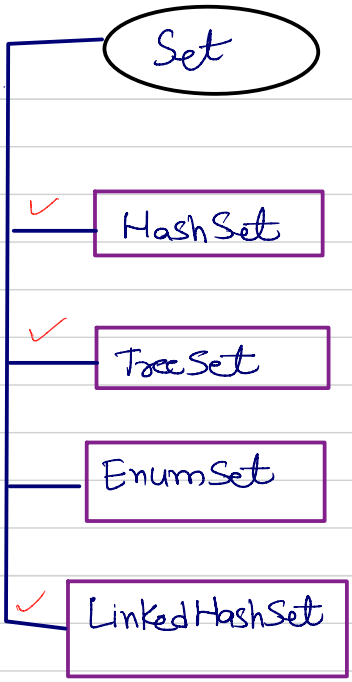
```
adq.peek()      // front
adq.peekFirst() // same as peek
adq.peekLast()  // last
```

```
adq.poll()      // front
adq.pollFirst() // same as poll
adq.pollLast()  // last
```

NOTE :- We can implement queue & stack using ArrayDeque.

java.util.HashSet
LinkedHashSet;
TreeSet;

Set



HashSet — unordered set

→ stores unique elements only.

```
Set<Integer> set = new HashSet<>();
```

```
set.add(1);
set.add(2);
set.add(3);
set.add(4);
```
} O(1)

→ stores elements in random order.

- → set.remove(value);
- → set.contains(value);
- → set.isEmpty();
- → set.size();
- → set.clear();

LinkedHashSet — uses LinkedList

→ Maintains the insertion order of elements

TreeSet → uses BST

→ Maintains the sorted order of elements. — O(logN)

Set of Custom classes (hashCode and equals)

```java
package equalsandHashcode;

import java.util.Objects;

public class Student {

    String name;
    int rollNo;

    public Student(String nameString, int rollNo) {
        this.name = nameString;
        this.rollNo = rollNo;
    }

    @Override
    public String toString() {
        return "Student [name=" + name + ", rollNo=" + rollNo + "]";
    }

    @Override
    public int hashCode() {
        return Objects.hash(rollNo);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Student other = (Student) obj;
        return rollNo == other.rollNo;
    }


}

```

```java
package equalsandHashcode;

import java.util.HashSet;
import java.util.Set;

public class setExample {

    public static void main(String[] args) {
        Set<Student> studentSet = new HashSet<>();

        studentSet.add(new Student("Kapil", 26));
        studentSet.add(new Student("Rahul", 22));
        studentSet.add(new Student("Ajay", 25));
        studentSet.add(new Student("Yogesh", 21));
        studentSet.add(new Student("Vineet", 20));
        studentSet.add(new Student("Ajay", 26));

        System.out.println(studentSet);
    }


}

```

→ using hashcode and equals to, we can differentiate custom class based on any field (Roll no in our case).

java.util.Map;
HashMap;
LinkedHashMap;
TreeMap;

MAP

HashMap

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

numbers.put("One", 1);
numbers.put("Two", 2);
numbers.put("Three", 3);

numbers.putIfAbsent("Two", 22);

for(Map.Entry<String, Integer> e : numbers.entrySet())
{
    Sout(e);          // whole entry
    e.getKey()        // key
    e.getValue();     // value
}

for(String key : numbers.keySet()) {
    sout(key);
}

for(Integer value: numbers.values()){
    sout(value);
}
```

O(1)

map.getOrDefault(key, 0);
↳ if key is present on map, then get value, otherwise get 0.

→ numbers.get(key); → to get the value of Particular key
→ numbers.containKey(key); → Boolean
→ numbers.containsValue(value); → Boolean
→ isEmpty();
→ clear();
→ numbers.remove(key);

TREEMAP → O(logN)

→ Maintains the Sorted Ordering of elements.

ArraysClass

→ Use to perform operations in Arrays (Default Arrays)

1. Binary Search: `int index = Arrays.binarySearch(arrayName, valueToSearch);`
2. Sorting :- `Arrays.sort(arrayName);`

   parallel sort → (Min 8192 elements)
3. Set values in every Index :- `Arrays.fill(arrayName, value);`
4. Convert Array in List :- `Arrays.asList(arr);`

Custom Comparator

1. Using Comparable class :-

Inside it, we have compareTo(Student that);

```
public int compareTo(Student that) {
    return this.rollNo - that.rollNo;
}
```

2. Using Custom Comparator

```
Collections.sort(list, new Comparator<Student>(){
    @Override
    public int compare(Student o1, Student o2) {
        return o1.name.compareTo(o2.name);
    }
});
```

Can be written as Using Lambda expressions.

```
Collections.sort(list, (o1,o2) -> o1.name.compareTo(o2.name));
```

- Comparator is a functional Interface, which has only one abstract method Compare(,).
- Since it is func.. Interface, we can write it in Lambda expression.

3. **For cleaner and more readable code → Use Comparator.comparingInt()**

java Copy

```java
PriorityQueue<Pair> pq = new PriorityQueue<>(
    Comparator.comparingInt((Pair p) -> p.first)
              .thenComparingInt(p -> p.second)
);
```

✅ **Most optimized and modern approach**.