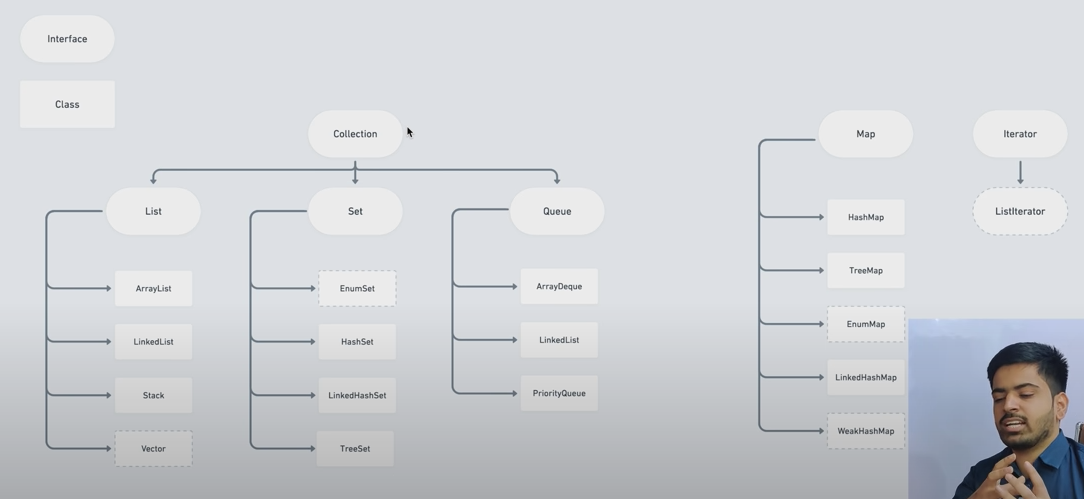
**Collection**: Framework consisting of interfaces and classes.

**Collection Interfaces:** List, set, Queue

**Map Interface:** Map

**Iterator interface:** ListIterator



**ArrayList and LinkedList.**

**Advantage of ArrayList over normal Array**: Array is static while ArrayList is dynamic. Array size is fixed while ArrayList size is dynamic. (n + n/2 + 1)

**Functions:**

**List<Integer> list = new ArrayList<>();**

**list.add(1); // Adding values in the list at the end.**

**list.add(2, 30); // Adding values at specific index.**

**list.addAll(newList); // Adding list into list.**

**list.addAll(1, newList); // Adding list into list at a specific position.**

**list.clone(); // Shallow copy of an arrayList.**

**list.get(1); // Getting element of specific index.**

**list.remove(1); // removing element of specific index.**

**list.remove(Integer.ValueOf(30)); // removing value 30 from list;**

**list.clear(); // clear the list.**

**list.set(2, 100); // updating value at second index with a new value.**

**list.contains(50); // return true/false if contains the value or not.**

**list.size(); // return the size of list.**

**list.indexOf(3); // returns index the element if present else returns -1.**

**List.isEmpty(); // returns true/false.**

**Ways to add multiple values together in arrayList.**

**List<Integer> list = new ArrayList<>();**

**list = Arrays.asList(1, 2, 3, 4);**

**Collections.addAll(list, 1, 2, 3, 4);**

**Stream in java 8**

**List<Integer> source = new ArrayList<>();**

**source = Arrays.asList(1,3,4);**

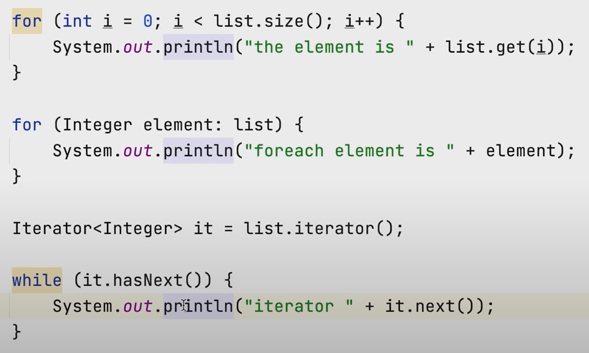
**List<Integer> target = new ArrayList<>();**

**source.stream()**

**.filter(i -> i > 1)**

**.forEachOrdered(target::add);**

**Iterating the ArrayList :**



**Stack: LIFO(Last in first out)**

**Stack<String> stack = new Stack<>(); // creating stack**

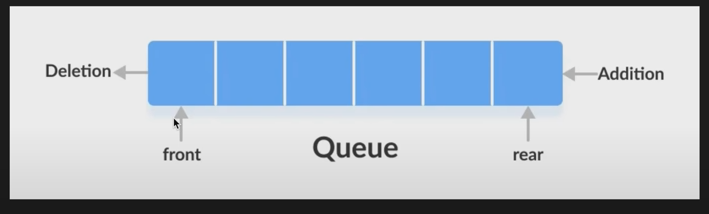
**stack.push(“saurav”); // pushing element.**

**stack.peek(); // return top element.**

**stack.pop(); // removes top element.**

**Stack.isEmpty(); // return true/false.**

**Queue: (FIFO)(First in First out)**



**Queue<Integer> queue = new LinkedList<>();/ ArrayList<>();**

**queue.offer(12); // adding element from rear, if unsuccessful, returns false.**

**queue.add(12); // adding element from rear, if unsuccessful, throws exception.**

**queue.poll(); // removes and returns top element. Returns null if empty.**

**queue.remove(); // removes and returns top element. Throws exception if queue is empty.**

**Queue.peek(); // front element. Returns null if empty.**

**Queue.element(); // front element. Throws exception if queue is empty.**

**Priority Queue :**

**Elements with specific priority will get removed first.**

**PriorityQueue<Integer> pq = new PriorityQueue<>(); // min heap**

**PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.reverseOrder()); // Collection.reverseOrder() // Max Heap**

**pq.offer(1); // Adding elements, makes sure that the front element is the smallest.**

**pq.poll(); // removes and returns priority element**

**pq.peek(); // front element**

**ArrayDeque : (sliding window)**

**Elements can be entered/removed from rear and front both.**

**ArrayDeque<Integer> adq = new ArrayDeque<>();**

**Adq.offer(); // add element at the last**

**Adq.offerLast(); // add element at the last**

**Adq.offerFirst(); // add element at the front**

**Adq.peek(); // front element**

**Adq.peekFirst(); // front element**

**Adq.peekLast(); // Last element**

**Adq.poll(); // removes and returns first element.**

**Adq.pollFirst(); // removes and returns first element.**

**Adq.pollLast(); // removes and returns Last element.**

**SET : Doesn’t add duplicate elements.**

**HashSet:**

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

**Set.add(3); // add elements in random order.**

**Set.remove(3); // remove mentioned element.**

**Set.contains(3); // return true/false.**

**Set.isEmpty(); // return true/false.**

**LinkedHashSet:**

**Set<Integer> set = new LinkedHashSet<>(); // same as above but order is maintained here .**

**TreeSet : O(log n)**

**Set<Integer> set = new TreeSet <>(); // same as set but elements are in sorted order with the help of binary search tree .**

**We can make a Set of custom class and can use equals and hashcode for checking that no two same objects are added.**

**MAP : Key value pair with unique keys.**

**HashMap:**

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

**hashMap.put(“one”, 1); // add key value pair. Over rides the key’s value if done.**

**hashMap.remove(“one”); // remove entry with one key.**

**hashMap.get(“one”); // get corresponding value of the key.**

**hashMap.containsKey(“one”); // returns true/false.**

**hashMap.putIfAbsent(“one”, 1); // put key value if key is absent**

**hashMap.containsValue(1); // returns true/false.**

**hashMap.isEmpty(); // return true/false;**

**hasMap.clear();**

**Iterating map:**

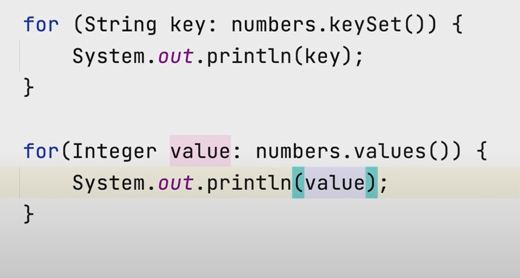
**for(Map.Entry<String, Integer> e = hashMap.entrySet()){**

**System.out.println(e); // print the entry set**

**System.out.println(e.getKey()); // print the key**

**System.out.println(e.getValue()); // print the value**

**}**



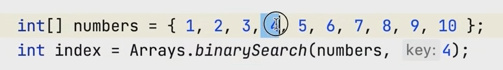
**LinkedHashMap are same as above but they are not random, they are ordered in which the keys are inserted in the Map.**

**TreeMap :**

**Sort the entry as per key. // same as above.**

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

**ArraysClass : inbuilt Array methods.**



**Returns the index of searched element.**

**Arrays.sort(numbers); // for sorting element. (Nlogn with quick sort)**

**Arrays.fill(numbers, 12); // fill whole array with 12**

**CollectionClass :**

**Collections.min(arrayList); // prints min element from the list**

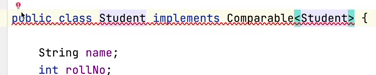
**Collections.max(arrayList); // prints max element from the list**

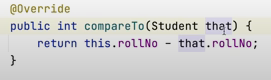
**Collections.frequency(arrayList, 9); // prints frequency of element from the list**

**Collections.sort(arrayList);**

**Collections.sort(arrayList, Comparator.reverseOrder());**

**If we have to sort an object as per one value inside it. We can implement comparable<className> and then make a comapreTo function.**





**For more complex comparator.**

**Collections.sort(arrayList, new Comparator<arrayList>(){**

**Public int compare(Student s1, Student s2){**

**return s1.name.compareTo(s2.name);**

**return s1.no – s2.no;**

**}**

**});**

**Lambda function.**

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



**hashcode and equals can be used to differentiate between multiple objects of same class as per their parameter values inside them. (It will prevent from adding object with same parameter values inside the set)**