**Collection Framework in Java**

* **It has 3 main interfaces:**

1. **Collection**
2. **Map**
3. **Iterator**

**Following image illustrates the same.**

|  |
| --- |
|  |

**Now, we will see all the important classes in detail**

1. **ArrayList**

Here, in starting when we create array-list at that time it’s length internall is zero ( means empty array is created). When we insert the first element then it’s internal array size will be 10.

After that if we insert the 11th element then it’s size will be

N = N + N/2 +1

And it will be continue

* **Syntax to create ArrayList**

ArrayList<data-type> variable-name = ArrayList<>()

* **Add function**

v.add(value:1)

v.add(index:1, value:2)

//for second one time complexity O(n) as internally it’s for loop

* **Remove**

v.remove(index:1)

v.remove(Integer.valueof(31))

//remove also has O(n) as it’s also internally implemented as for loop

* **Get function**

v.get(index:1)

* **Set function**

v.set(index:3, value:31)

* **Contain function**

v.contains(value:50) //O(n)

It is boolen type so it returns true or false

1. **Iterator**

It has many functionality inbuild. Basically it’s used to iterate any type of collection

* **Syntax to create**

Iterator<data-type> variable-name = variable2.iterator()

//variable2 is the variable for which we are creating it.

1. **Stack**

It has Last in first out (LIFO) functionality

* **Syntax to create a stack**

Stack<data-type> variable-name = new Stack<>()

//variable2 is the variable for which we are creating it.

* **Push function**

s.push(value: “Lion”)

* **POP function**

s.pop()

* **Peek function**

s.peek() // returns top of stack

1. **Queue**

It has First in First Out (FIFO) mechanism

We can implement it, using array or Linked-List.

In collection framework simple queue is implemented as Linked-list.

* **Syntax to create normal Queue**

Queue<data-type> variable-name = new LinkedList<>()

//here LinkedList class has implemented all the functions of queue.

* **Offer function (alternative add)**

q.offer(value:2) //to add data

it returns true if successful otherwise false

* **Poll function (alternative remove)**

q.poll() // to remove

It returns value that have been removed by it. Also “null” if queue is empty

* **Peek function (alternative element)**

q.peek() //to get value which is at most front. Null for empty.

Here, all the alternatives performs the same task as it’s neighbour. But, it throws exception if task unsuccessful and in that case your program will crash.

1. **Linkedlist**

We know how the linked-list is implemented in DSA. Here, also it has the same meaning.

It has all the functions that Arraylist has.

1. **Priority Queue**

Same meaning as we studied in DSA. Has all the function that queue has. Just added concept of priority

* **Syntax**

PriorityQueue<data-type> varible-name

= new PriorityQueue<>()

//by default it uses ascending order. But to use as descending order write following in () while creating it.

Comparator.reverseOrder()

1. **ArrayDeque**

It is double ended queue. Internally it is implemented as

Array.

* **Syntax**

ArrayDeque<data-type> varible-name

= new ArrayDeque<>()

It has all the function as Queue

* **Function for addition**

ad.offer(value:3) //normal addition at last

ad.offerFirst(value:3) //front addition

ad.offerLast(value:3) //addition at last

* **Function to remove**

ad.poll() // normal removes from last

ad.pollLast() //removes from last

ad.PollFirst() //removes from front

* **peek functions**

ad.peek(), ad.peekLast(), ad.peekFirst()

**Now, related to Set interface:**

1. **HashSet**

* As it is implements set interface it not allowed duplicate and elements are in random order
* Here all the operations have the time-complexity **O(n)**
* It internally uses hashing, so set generate Hash for every element. Means each element of the Set has it’s corresponding Hash. And that Hash must be unique.

//this is the reason for why duplicate elements are not allowed and elements are not in order?

* **Syntax to create Hash-set**

HashSet<data-type> variable-name = new HashSet<>()

* **Add operation**

set.add(value:3) // to add the element

* **Remove operation**

set.remove(value:3)

//remember to provide value as it stores elements in random order

* **Contains function**

set.contains(value:50)

//It is boolean type so it returns true or false

* **isEmpty function**

set.isEmpty()

//It is boolean type so it returns true or false

* **size function**

set.size() //will return the size of set

* **clear function**

set.clear() // to clear whole set

* additional for Hash-set
* It will not work properly for user-defined classes, as we have to dynamically create the elements in that case. And Hash-set just checks that whether the 2 elements are different or not, remember it not check the value.
* To overcome it we have to override the equals and Hash-code methods so that we can instruct to HashSet that check the value of particular member of class instead to check whether they are seprate or not.

1. **LinkedHashSet**

* It is same as HashSet but, as it implements set + Linked-list properties so, in addition to HashSet here order of element will be maintain

Means, here duplicates are not allowed but order of element is matter of subject.

* Operations are same as Hash-set.

1. **TreeSet**

* It also implements the set interface but behind the scene it also uses properties of binary search tree, means here element will be unique and in sorted order. (order of element insertion is not the matter of subject)
* Here all the operations have the time-complexity **O(log n),** as they implements the binary search tree too.
* Operations are same as HashSet.

**So, all the 10 classes that we discussed, they implements the collection interface. Now we will see Map interface which is seprate from collection interface**

**Reasons:**

* Map used to store key value pair, whereas all the classes of collection interface are used to store just values

1. **HashMap**

* Here, keys are unique. And if we renter the data with same key then it will override old data.
* Same as HashSet here also Hash is created for every pair. So keys are unique but order not matter.
* It has entry class which keeps entry of all the value pair entered.
* Also, time-complexity for all the operations is **O(1)**
* **Syntax to create Hash-map**

HashMap<data-type1, data-type2> variable-name

= new HashMap<>()

* **Put operation**

map.put(value1: “us”, vlaue2: “United States”)

//to add the data.

* **putIfAbsent function**

map.putIfAbsent(value1: “us”, vlaue2: “United States”)

// add it only if provided key is absent

* **contains-key function**

map.containsKey(value1:”us”) // to check, is key present?

//it’s Boolean so will return true or false

* **contains-value function**

map.containsValue(value2:”United States”)

// to check, is provided value present?

//it’s Boolean so will return true or false

* **is-Empty function**

map.isEmpty()

//it’s Boolean so will return true or false

* **How to iterate in it**

for (Map.Entry<data-type1, data-type2> e:

Variable-name.entrySet())

{

System.out.println(e); //will print whole entry

System.our.println(e.getKey()) //print only key

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

}

1. **TreeMap**

* Same as TreeSet it internally implements Binar Search Tree so, basically it puts all the keys int binary search tree.
* So, here we have unique keys in sorted order.
* Functions are same as HashMap but time-complexity is

**O(log n)**

**So, with it all the important classes of collection and map interface is completed. Few of the it’s remaining classes are rerly used in real world problems and applications**

**Now, let’s see 2 other classes which is part of collection framework but not contains in collection and map interface**

1. **Arrays**

* It’s normal class available in java.util package.
* It provides functionality to manipulate the array

(not Array-list !!!)

* **binarySearch operation**

Arrays.binarySearch(array-name, value)

//it will return the index of given value in array.

* **Sort operation**

Arrays.sort(array-name)

* **Fill operation**

Arrays.fill(array-name, value)

// all the elements of the array will be replaced by value

1. **Collection class**

* It’s provided on top of collection interface
* It has some handy functions like -find min, -find max, etc…
* **Operations**

Collections.min(variable-name) //return min

Collections.max(variable-name) //return max

Collections.frequency(variable-name, value)

//return frequency of given value

Collections.sort(variable-name) //sort in ascending

Collections.sort(variable-name, Comparator.reverseOrder()) //sort in descending