**Collection Framework**

**Collection :** It is a single entity that represents group of objects.

Collection is an interface.

**Collection Framework :** Define different interfaces and classes that represents the group of objects into single entity.

**Array**  **Collection Framework**

1. Stores only Homogeneous data 1) Stores both Homogeneous and Heterogeneous
2. Fixed Size (not grow able in nature). 2) Grow able in nature.
3. No underlaying data structure 3) Implemented by using (ready made methods) data structure

Collection framework classes and interface :

ArrayList, List, HashMap, HashTable, Tree, stack…

**Collection :**

* **Collection** is a **interface** which is the root of all other interfaces and classes.
* Collection interface has methods are commonly used by all other collection classes.

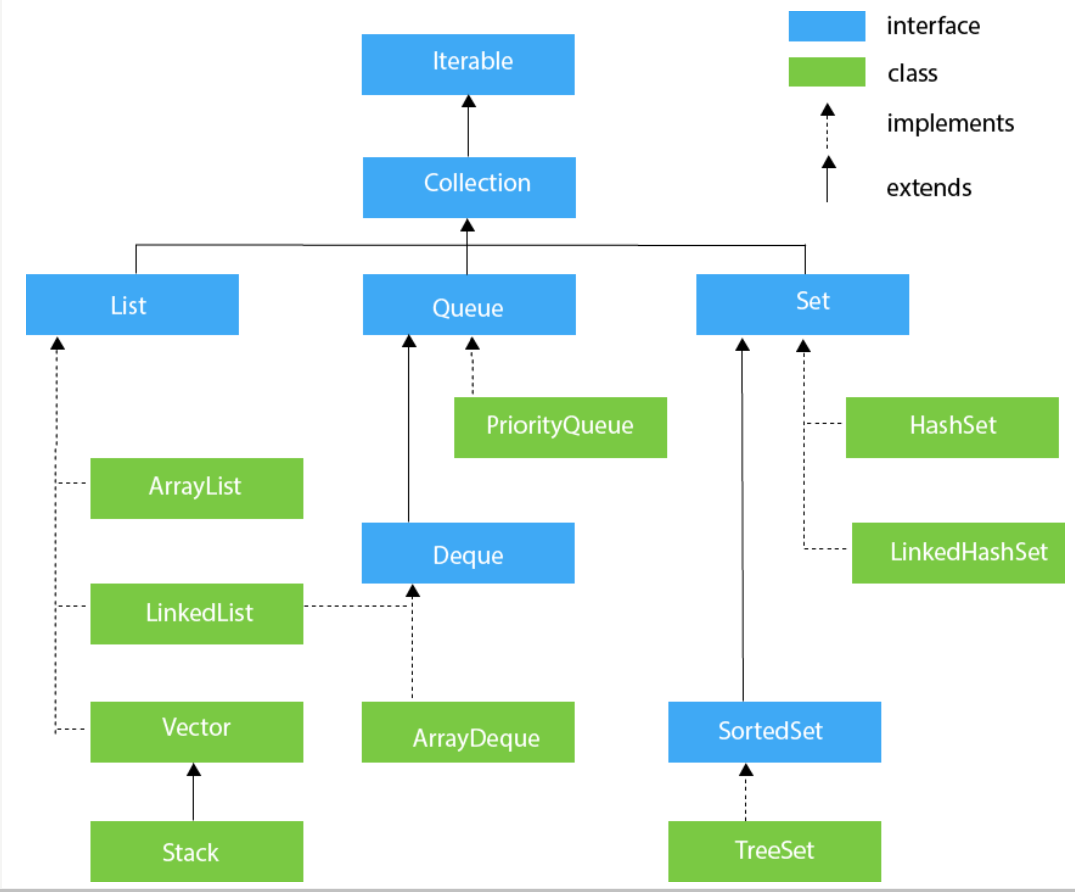
**Collections :**

* **Collections** is a pre-defined **class** from java.util package.
* Collections class contain certain methods that are used to perform some operations on collection interface.
* Example that explain above statement

Let an array ARR contains some objects. Now sort the objects using

Collections.sort(ARR)

Here Collections is a class in the util package that have sort method which performs on ARR that is collection.



1. **List :**

* List is an interface which is a child of collection interface that means List extends collection
* It is used when Insertion order is preserved and duplicates are allowed.
* **Classes :**
  + ArrayList
  + LinkedList
  + Vector (extends stack) (legacy classes)

1. **Set :**

* Set is an interface that do not preserve the insertion order and do not allow duplicates.
* **Classes:**
  + HashSet
  + LinkedHashSet

1. **Queue**

* Queue is an interface used when ever the objects are to be prioritize.
* **Classes :**
  + PriorityQueue

**Note :** The methods that are available in root interface collection are also available in List,set and Queue.

**Map**

* Map is an interface and it is **not** a **child** of **collection** interface.
* Whenever the objects are stored in the form of key-value pair Map interface is used.
* Where both key and value are objects. Key is unique but values can be duplicate.
* **Classes :**
  + HashMap
  + LinkedHashMap

**Methods in collection interface :**

1. **add(Object o) : //append**

* Add a new object into a collection.

1. **addAll(Collection c) :**

* Add multiple objects into a collection

1. **remove(Object o):**

* Remove an object from the collection.

1. **removeAll(Collection c) :**

* Remove group of objects(collection) that are specified from the collection.

1. **retainAll(Collection c)**

* Remove group of object from the collection except that are specified.
* For example if there is a collection contain (1,2,3,4) objects then a collection given is (1,3). Then the method retainAll() will remove the other objects (2,4) from the collection.

1. **clear() :**

Remove all objects from the collection.

1. **isEmpty() :**

Check whether collection empty or not

1. **size() :**

Return number of objects in a collection

1. **contains(Object o) :**

Check whether object is present in collection or not

1. **containsAll(Collection c) :**

Check whether group of objects are in collection

1. **toArray(collection c)**

Convert collection into array. Returns an object array.

**Methods in List interface :**

1. add(index,Object o) //insert
2. addAll(index,Collection c)
3. remove(index)
4. get(index) : return the object at index
5. set(index,object o) : replace the object with new at index

**Classes in List**

1. **ArrayList :**

* ArrayList is a class under java.util package.
* **Declaration :** ArrayList al = new ArrayList();
* Provides **10** default memory locations.
* Accepts Heterogeneous data.
* To store any type of data(Heterogeneous):
  + ArrayList al = new ArrayList();
* To store specific type of data(Homogeneous):
  + ArrayList<String> al =new ArrayList<String>;

**Methods in ArrayList :**

add(object o) **// append**,add(index,object o) **// insertion**,size()

remove(index),get(index),set(index,object),contains(object),isEmpty() , removeAll(),addAll()

Sort an array using Collections class.

* To sort an array the array should be of same data type.
* Collections.sort(arr) // ascending order
* Collections.sort(arr,Collections.reverseOrder())//descending
* Collections.shuffle(arr) // arrange objects in random order
* Convert array to ArrayList:
  + ArrayList al=new ArrayList(Arrays.asList(arr));

**Read the data in three ways:**

1. **For loop**

for(int i=0;i<al.size();i++)

{

System.out.println(al.get(i));

}

1. **For..each**

for(Object e:al)

{

System.out.println(e);

}

1. **Iterator**

Iterator is a parent interface of collection interface

Iterator it=al.iterator();

while(it.hasNext())

{

System.out.println(it.next());

}

**Note :** (i) **ArrayList** is used when there are more number of **retrieving operations.**

(ii) The performance is more for insertion or deletion operations.

1. **LinkedList :** (By default it is doubly linked list)

* LinkedList is a class that implements List and Deque interface.
* The elements are not stored in contiguous order.
* Contains node
* Each node contains three parts : previous node address, object and next node address.
* Used to develop stacks and queues.
* No default memory locations.
* **Methods in LinkedList :**
* General methods from List interface :
* add(x) //append
* add(index,x) //insertion
* addAll(Collection c)
* remove(index)
* removeAll(Collection)
* get(index)
* set(index,object)
* Sorting and shuffling using Collections class (same as in ArrayList)
* Methods from deque
* addFirst(object) //add new node at the starting
* addLast(object) // add new node at the last
* removeFirst()
* removeLast()
* getFirst()
* getLast()
* Declaration:

Heterogeneous: LinkedList l=new LinkedList();

Homogeneous:LinkedList<String>l=new LinkedList <String>();

**Note :** **LinkedList** is used when there are more number of **Insertion or deletion operations.**

1. **Set**

* Classes :
  + HashSet
  + LinkedHashSet
* **HashSet:**
* Insertion order not preserved and no duplicates.
* Insert elements using hash code that helps for fast searching.
* 16 memory allocation by default after declaration.
* Declaration : HashSet hs=new HashSet();

By default **ArrayList** has 10 memory locations, if 11th object is added then a new ArrayList object is created with more memory locations . The 10 elements are added to the new and then 11th element is added.

Similarly in **HashSet** has 16 default memory locations are created. HashSet has a term known as **fill ratio or load factor.** The **default** load factor is **0.75.** When ever the elements filled up to 0.75 in an HashSet then a new HashSet is created with more memory locations. These elements are copied to the new hash set and remaining elements are added to new hashset.

The fill ratio or load factor can be changed

HashSet hs=new HashSet(); //set of default size 16

HashSet hs=new HashSet(100); // set of size 100

HashSet hs=new HashSet(100,0.65); //set with size 100 and load factor 0.65

* The HashSet does not have any specific methods. Have the methods that are available in Set.
* add(value),addAll(Collection),remove(Value),removeAll(Collection),contains(value), containsAll(Collection), isEmpty()
* No sort method. If we need to sort convert it into ArrayList.

Union : set1.addAll(set2)

Intersection : set1.retainAll(set2)

Difference : set1.removeAll(set2)

Subset : set1.containsAll(set2)

**Note** : **HashSet** is used when there are **searching** operations.

* **LinkedHashSet**
* Duplicates are not allowed and insertion is preserved
* LinkedHashSet = HashTable + LinkedList
* Intial size 16 and load factor 0.75

1. **Queue :**

* Child Interfaces
  + Deque
  + BlockingQueue
  + Blockingdeque
* Classes
  + PriorityQueue
  + LinkedList (implemented two interfaces List and Queue)
* When group of elements which are prior to processing then queue is used.
* Principle - First In First Out
* Similarities : Insertion order is preserved and Duplicates are allowed in both LinkedList and Priority Queue classes.
* Difference : In LinkedList Heterogeneous data where as in PriorityQueue only Homogeneous data.
* 6 methods :
  + **add()** - Add an element to the queue. Return True if it is successful, Otherwise throws **Exception**.
  + **offer()** - Add an element to the queue. Return True if it is successful otherwise return **Null.**
  + **element()** - Return head element from the Queue. If the queue is empty then throws Exception.
  + **peek()** - Return head element from the Queue. If the queue is empty then return Null.
  + **remove()** - Return header element and then remove the head element. If the queue is empty then throws Exception.
  + **poll()** - Return header element and then remove the head element. If the queue is empty then return Null.

Exception - NoSuchElementException

**Map**

* Map is a collection of **Entry.**
* Entry is a **key-value pair**.
* Classes:
  + HashMap
  + HashTable

**HashMap :**

* Under laying data structure is HashTable.
* Insertion order is not preserved and duplicate keys are not allowed but duplicate values are allowed.
* Null key is allowed once and Multiple null Values are allowed.
* Declaration : HashMap hm=new HashMap();

HashMap<Integer,String> hm=new HashMap<Integer,String>();

* **Methods :**
  + put(key,value) - add new entry
  + putAll(map c) - add a map to the map
  + get(key) - return the value based on the key
  + remove(key) - remove the key-value pair
  + containsKey(key) - return true if key is available in the map, Otherwise return false.
  + containsValue(value) - return true if value is available in the map, Otherwise return false.
  + isEmpty()
  + size()
  + clear()
  + **keySet()** - return set that contains all keys.
  + **values()** - return collection that contains all values.
  + **entryset()** - return set that contains all entries.
* Entry -(key-value pair) is represented by interface called as **Entry** that is subset of HashMap class.
* Methods available in Entry interface:
  + getKey()
  + getValue()
  + setValue(object) // update value
* Displaying the enteries using entryset() method:

for(Map.Entry entry:hm.entrySet())

{

System.out.println(entry.getKey() + “ “+entry.getValue();

}

**Using iterator:**

Set s=hm.entrySet();

Iterator it=s.iterator();

while(it.hasNext())

{

Map.Entry entry=(Entry)it.next();

System.out.println(entry.getKey() + “ “+entry.getValue();

}

Using keySet() method

for(Object k:hm.keySet())

{

System.out.print(k+” “+hm.get(k));

}

**Note** : HashMap is used when there are more number of **search operations.**

**HashTable:**

* Default size is 11 when declared and load factor is 0.75.
* Declaration : Hashtable ht=new Hashtable();

Hashtable<Integer,String> ht=new Hashtable<Integer,String>();

**HashMap vs HashTable:**

**Similarities :**

* Data is in the form of key-value pair.
* Underlaying Data structure is hashtable (internally it uses hashcode).
* Insertion order not preserved.

**Differences:**

**HashMap** **HashTable**

1. Non-Synchronized 1. Synchronized
2. Not Thread safe 2. Thread safe
3. Performance is faster 3. Performance is slow
4. Null’s accepted 4. Null’s not accepted

**Synchronized :** Only one thread can access a method at a time

**Non-Synchronized :** Multiple threads can access a method at a time.