**Collection Framework**

**Collection🡪 A group of elements or objects as a single entity called Collection.**

**Collection(I)🡪 Root interface of collection framework for all other collection related I & C.**

**Collections(C)🡪 Provides methods for all common collection object operation.**

From java.util package ar(1,3,6,4,2) Collections.sort(ar);

**Collection(I)🡪 List(I), Set(I), Queue(I)**

**Collection🡪List🡪ArrayList(c), LinkedList(c), Vector(c)🡨Stack (Vector and Stack are legacy classes)**

1. List is a child interface of Collection interface
2. Insertion order should be preserved, Duplicates allowed

**Collection🡪Set 🡪HashSet(c), LinkedHashSet(c)**

1. Insertion order should not be preserved
2. Duplicates are not allowed

**Collection🡪Queue 🡪PriorityQueue(c)**

Whenever we want to represent a group of objects, which are prior to processing **(FIFO system)**

**Map(I)🡪 HashMap(c), LinkedHashMap(c), HashTable(c)**

Map is not a child interface of Collection interface. Map also represents group of objects but in the form of key-value pairs.

Keys🡪 Cannot be duplicate

Values🡪 Can be duplicate

**Methods available in Collection Interface :**

1. add(Object o)🡪 To add an object to the collection
2. addAll(Collection c)🡪 To add multiple object to the collection
3. remove(Object o)🡪 To remove an object from the collection
4. removeAll(Collection c)🡪 To remove all objects from the collection specified in Collection c
5. retainAll(Collection c)🡪 Except these (c) objects remove all others from the collection
6. clear()🡪 clear all the objects from the collection
7. isEmpty()🡪 to check if collection is empty or not
8. size()🡪 to check how many objects are available in the collection
9. contains(Object o)🡪 will check that particular object is present in the collection or not
10. containsAll(Collection c)🡪 All group of element present or not
11. toArray(Collection c)🡪 Convert entire collection into array and return Object[].

**Methods available in List Interface : Has its own methods (Index play important role)**

1. add(index, Object o)🡪 To add element at specific index or position
2. addAll(index, Collection c)
3. remove(index)
4. get(index)🡪 Return the object at specified index
5. set(index, Object o)🡪Replace existing object with new object

**ArrayList(c) : Allowed duplicates, insertion order is preserved**

Present in java.util pkg

ArrayList al = new ArrayList(); (here al is a reference variable)🡨 **For Heterogeneous data**

By default 10 locations are allocated when we creates new ArrayList object.

ArrayList <String> al = new ArrayList<String>(); 🡨 **For Homogeneous data**

**Methods available in ArrayList class :**

All Collection and List Interface methods are available in the ArrayList class.

To search🡪 contains() method

To sort🡪 Make use Collections class

e.g Collections.sort(al); // But for sorting elements should be same type

To shuffle🡪 Collections.shuffle(al);

To convert Array to ArrayList🡪

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class Tester {

   public static void main(String args[]) {

      String[] array = {"a", "b", "c", "d", "e"};

      //Method 1

      List<String> list = Arrays.asList(array);

      System.out.println(list);

      //Method 2

      List<String> list1 = new ArrayList<String>();

      Collections.addAll(list1, array);

      System.out.println(list1);

      //Method 3

      List<String> list2 = new ArrayList<String>();

      for(String text:array) {

         list2.add(text);

      }

      System.out.println(list2);

   }

}

**LinkedList(c) : Allowed duplicates, insertion order is preserved, used when frequent insert / delete.**

**ArrayList :** Whenever we need to perform **retrieving** **operation frequently** then go for ArrayList, because in ArrayList list of elements are stored in the form of index. E.g get(index)

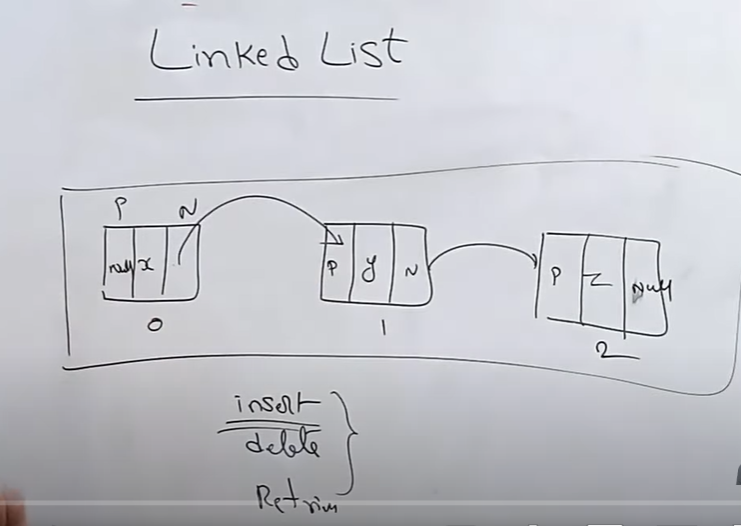
We should not prefer ArrayList if we need to perform **Insertion / deletion operation frequently**.

Why not ArrayList, because in insertion / deletion shifting of element takes place and that causes high performance.

In ArrayList elements are stored in the consecutive memory locations. In case of LinkedList elements are stored in random free space in the memory locations and they are linked together.

A white paper with black writing

Description automatically generated with low confidence

**NODE** : Represent a single element with 3 parts 🡪 prev\_address, actual\_value, next\_address****

INSERT :

A picture containing text, whiteboard, outdoor

Description automatically generated

By default LinkedList follow the doubly Linked list data structure.

LinkedList class we can use to implement **stack(FILO) and queue(FIFO)**

**There is no default size for LinkedList as ArrayList has 10 default size.**

**Extra methods of LinkeList : List(I), Dequeue(I) methods are by default available to LinkedList(c) 🡪These methods are mostly used to implement STACK and QUEUE**

1. addFirst(Object o)🡪 adds element as a first node in LinkedList
2. addLast(Object o) 🡪 adds element as a last node in LinkedList
3. removefirst()
4. removeLast()
5. getFirst()
6. getLast()

**Set(I) :** Duplicates are not allowed and insertion order should not be preserved.

**HashSet(c) :** Duplicates are not allowed and insertion order should not be preserved. HashSet will insert the elements by using the concept **HashCode**. Because of this searching an element will be much faster. There is **no index** concept. Heterogeneous data and null are allowed. Underlaying DS 🡪 HashTable

**When to HashSet?**

If you have more number of searching then go for HashSet.

**Creating HashSet :**

HashSet hs = new HashSet();🡪 by this it provides 16 initial locations, load factor=0.75

HashSet hs = new HashSet(100);🡪 by this it provides 16 initial locations, load factor=0.75

HashSet hs = new HashSet(100, 0.90);🡪 by this it provides 16 initial locations, load factor=0.75

HashSet<Integer> hs = new HashSet<Integer>();🡪 for Homogeneous data

**Load factor always in decimal format.**

Load factor / Fill ratio 🡪 0.75

**Load factor / Fill ratio** 🡪In case of HashSet as soon as it fills 75% locations of 16, I will create the new object with big size, copies all elements into it and then adds new elements to it.

**ArrayList🡪** In case of ArrayList by default size is 10 as soon as we try to insert 11th element it will creates new object with big size, copies all elements into it and then adds new elements to it.

HashSet don’t have their own specific method, they only have Set(I) methods.

We can not directly sort and shuffle HashSet, for that we need to first convert it to the ArrayList.

**Methods Available in Set(I):**

1. add(Object o)
2. addAll(Collection c)
3. remove(ObjectName / value)
4. removeAll(Collection c)
5. contains(Object o)
6. containsAll(Collection c)
7. isEmpty()

**LinkedHashSet(c):** Duplicates not allowed but insertion order is preserved.

Underlaying DS 🡪 HashTable + LinkedList, Initial\_size=16 and Load\_factor=0.75

**Queue(I) : 🡪 Dequeue(I), BlockingQueue(I), BlockingDequeue(I), PriorityQueue(C), LinkedList(C).**

If we want to represent a group of element which are prior to processing then we should go for Queue.

e.g You want to send SMS to 100 mobile number, then SMS is sent to one by one number in Queue.

**PriorityQueue(C)🡪** Insertion order is preserved, duplicates allowed, heterogeneous data is not allowed.

**LinkedList(C)🡪** Insertion order is preserved, duplicates allowed, heterogeneous data is allowed.

**Methods of Queue Interface :**

1. add()🡪 to add element in queue at the tail, returns true if successful, else returns exception
2. offer()🡪 to add element in queue at the tail, returns true if successful, else returns false
3. element()🡪 returns head/first element for processing, if queue is empty it will return exception
4. peek()🡪 returns head/first element for processing, if queue is empty it will return null
5. remove()🡪 it will return and then remove header element, if queue is empty it will return exception
6. poll()🡪 it will return and then remove header element, if queue is empty it will return null

**Map(I)🡪 HashMap(C), HashTable(C)**

Map is not a child interface of Collection(I), it is a totally independent Interface in java collection.

If you want to represent group of element in the key-value pair, then go for Map. E.g EmpId and Ename

A combination of key-value is called a pair or a dataset or a enrty. Every key is an object and every value is an object. So map is a collection of entries.

In Map duplicate keys are not allowed but duplicate values are allowed.

When we try to add duplicate key with new value in that case it will not throw any error or exception, instead it will replace old value with new value for the mentioned key.

**HashMap(C)🡪**

Underlaying data structure is HashTable. Insertion order is not preserved. Duplicate keys are not allowed but duplicate values are allowed. Null key is allowed only once but multiple null values are allowed.

When to use🡪 for search operation.

**Methods available in HashMap(c) 🡪**

1. m.put(key, value)🡪 to add an entry
2. m.putAll(Map m)🡪 to add multiple entries
3. m.get(key)🡪 to get the value of associated key
4. m.remove(key)🡪 to remove an entry from the map
5. containsValue(value)🡪 to search value (true/false)
6. containsKey(key)🡪 to search key (true/false)
7. isEmpty()
8. m.size()🡪 It will return the how many entries HashMap have.
9. m.clear()🡪 clear the entire HashMap
10. **Methods of key-value pair**
11. m.keySet()🡪 it will return all keys, and return type is Set
12. m.valuse()🡪 it will return all the values from HashMap, and return type is Collection
13. m.entrySet🡪 will return all the entries from HashMap as return type Set

**Entry(I)🡪** Entry interface represents a single entry from HashMap/Map.

**Methods of Entry Interface :**

1. e.getkey()
2. e.getvalue()
3. e.setValue(Object o)🡪 change the value of a particular key

**Hashtable(C)🡪 Represents group of element as a key-value pair 🡪 LEGACY class**

**Hashtable🡪** All methods Synchronized (means at a time only one thread can access Hashtable methods, thread safe, performance is faster, nulls cannot be accepted either as key/value

**HashMap🡪** Non synchronized (multiple threads allowed), not thread safe, performance is poor/low, nulls can be accepted either as key/value (1 null-in keys, multiple nulls-in values allowed)

**Note :** All HashMap methods are applicable for Hashtable (same methods for both)