javapoint.com

Collections in Java

↳ a framework that helps us ~~to~~ store & manipulate gp of objects.

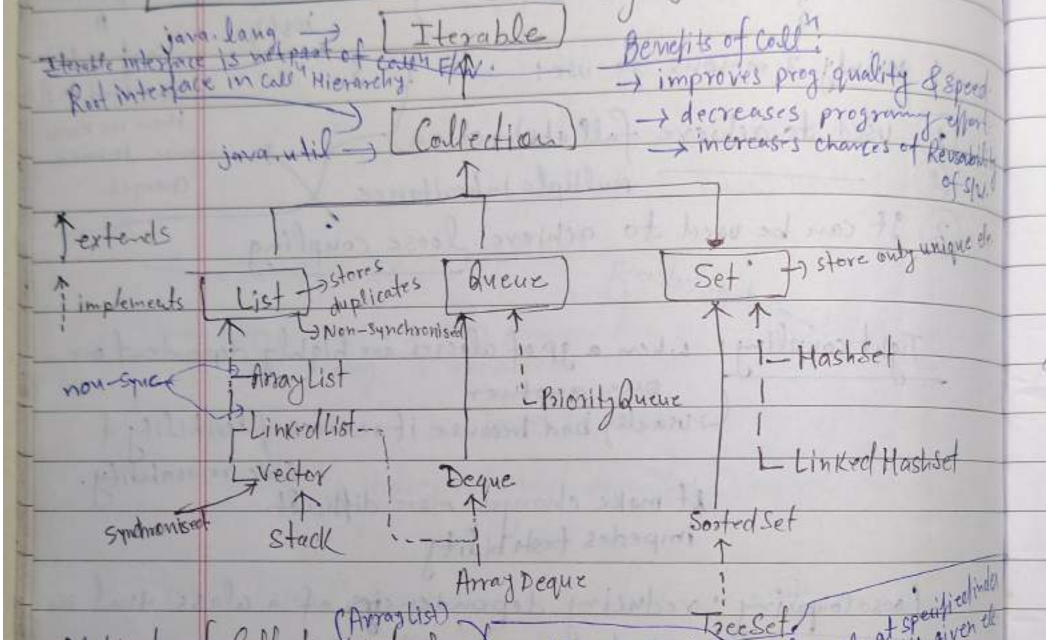
* java.util" pkg.

S1.equals(S2);

S1.equalsIgnoreCase(S2);

oper^{ns}: Searching, sorting, inserti manipul^{n}, del^{n} etc. can be perform by java coll^{n}!

Benefits of coll^{n}:
→ improves prog quality & speed
→ decreases programming effort
→ increases chances of Reusability of s/w.



Methods of Collection interface

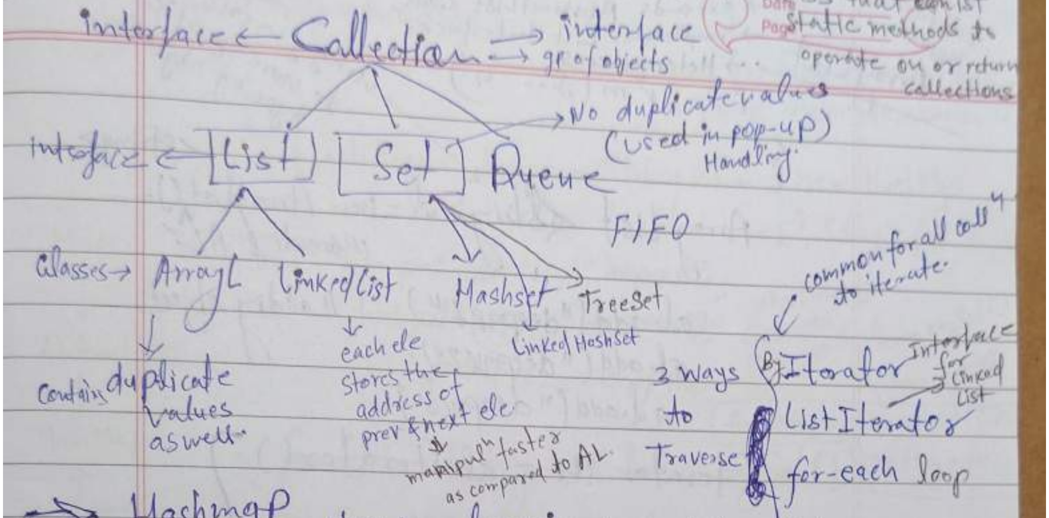
✓T/F	add() → obj	insert an ele → add(index, obj)
T/F	addAll() → Collection	insert the specified coll^{n} ele in the invoking coll^{n} → addAll(index, coll^{n}s)
"	remove() → obj	delete 1st occurrence of ele if present
	remove() → index	delete by specified posi^{n} → integer → prim D.T.
"	removeAll() → coll^{n}	To delete an int^{r} by obj Integer.valueOf() → pass it
"	retainAll() → coll^{n}	→ delete all ele of invoking coll^{n} except the specified coll^{n}
	removeRange(fromIndex, toInd) → Inclusive, exclusive	Delete ele b/w range.
int	size() → No. of ele.	
void	clear()	→ deletes all ele from coll^{n}
T/F	contains() → obj	→ search^{an} ele
T/F	containsAll() → coll^{n}	
Iter^{r}	iterator()	
	listIterator()	
obj[]	toArray()	coll^{n} → Array
T/F	isEmpty()	
T/F	equals() → obj.	matches two coll^{n}
int	hashCode()	returns hashcode no. of coll^{n}

(ArrayList)
set(index, obj) → Replace ele at specified index with given ele
get(index) → returns ele at sp pos
indexOf(obj) → index of 1st occurrence of ele
lastIndexOf(obj) → last occurrence, if not present ele → (-1) returns
subList(fromindex, to) → a part of list
Incl, Excl.

→ No method in Set which uses Index. List has.

Collection & Map ⇒ both interface

Collections → class that consist static methods to operate on or return collections



interface ← Collection → interface
→ gr of objects
interface ← List
Set
Queue
→ No duplicate values (used in pop-up) Handling
FIFO
Classes → ArrayL
LinkedList
Hashset
Treeset
LinkedHashSet
Contain duplicate values aswell.
each ele stores the address of prev & next ele
manipulⁿ faster as compared to AL
common for all collⁿ to iterate.
3 ways to Traverse
Iterator
Interface for Linked List
ListIterator
for-each loop

⇒ Hashmap stores key-value pair → called "entry"

Key	value
DeviceID	LG
OS	Android
soversion	5.0

collⁿ → interface, collⁿˢ → class
both are present in java.util
Both are part of collⁿ F/W

hasnext() → returns boolean value
next() → returns value of ele(objects) → object

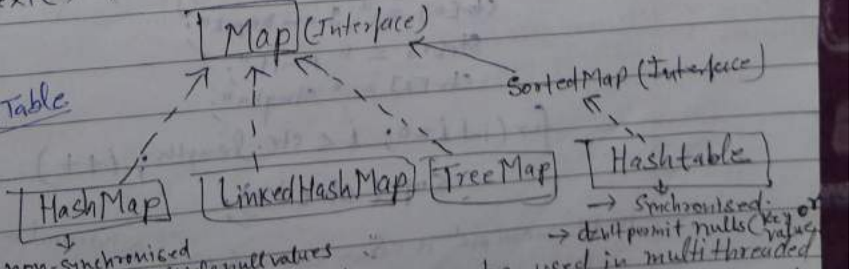
→ Collections do not support prim. Data type. → Auto boxing & unboxing happens here

⇒ Iterator interface provides facility of iterating the elements in forward direction only. has 2 methods

T/F hasNext() → returns True if iterator has more elements

Object next() → returns an ele & moves cursor pointer to the next ele.

HashMap vs HashTable



Hashtable → Synchronised
→ don't permit nulls (key or value)

HashMap
① → non-synchronised
② → one null key, multiple null values
③ faster

Synchronised ⇒ thread-Safe ⇒ can be used in multithreaded Environment.

Concurrent Hashmap
same 2pt of Hashtable. { Synchronized, don't allow null keys & null values.

AL → uses dynamic array for storing elements
→ can contain duplicate elements → Manipulation is slow
→ maintains insertion order. → Non-Synchronised
→ extends AbstractList class
→ implements List interface.
→ Heterogeneous

Diff from Hashset

~~ArrayList~~

using generic to store specific type of objects

```
p.s. v m(S... a)
{
    ArrayList <String> al = new ArrayList<String>().    // Creating AL

        al.add("deepak1").     // adding objects
        al.add("deepak2").
        al.add("deepak3").

    Iterator itr = al.iterator().
    while (itr.hasNext())
        {
            Sop( itr.next()).
        }
    }
}

for(String obj : al)
    {
        sop(obj).
    }
```

Arraylist	Linkedlist
① Implementation Resizable Array	Doubly linked list
② Preferred when more retrieve or Search oprn Needed	preferred where manipuln Needed
③ No memory overhead	Yes, LL needs to maintain the addresses of next & prev. node
④ Performance depend on type of oprn Search → fr Manip → sr	

Above using Array

```
String [] str = new String [3];
    Str[0] = " I am Deepak".
    Str[1] = " Amit".
    Str[2] = " Singha".
for(int i=0; i < str.length; i++)
    Sop(str[i]).
```

String s.length()
StringArray s.length
int
float

→ in AL manipuln is slow coz a lot of shifting occurs internally if any ele is removed from AL. → Internal mechanism is Array
→ AL internally uses dynamic Array to store ele

H.M ② → contains only unique ~~elements~~ Keys → Like HashSet
→ may have one null Key & multiple null values
→ It maintains no order.

Hashmap ① → Stores the data in key-value format. → used in mobile appⁿ testing to store device, OS, OSversion

```
p.s.vm(S... a)
{
    HashMap<Integer, String> hm = new HashMap<Iʳ, String>();
    hm.put(100, "deepak1");
    hm.put(101, "deepak2");
    hm.put(102, "deepak3");
    for(Map.Entry m: hm.entrySet())
    {
        Sop(m.getKey()+"  "+m.getValue());
    }
```

→ By looping
① entrySet
② key set. { hm.keySet() hm.get(k)

```
for(String k : hm.keySet())
sop(hm.get(k));
```

Map
↑
AbstractMap
↑
HashMap
↑
LinkedHashMap

Same as HashMap but it maintains insertⁿ order.

Hashset

add(), remove(), clear(), size()
contains(), iterator()
isEmpty()

- doesn't accept duplicate value, unique only
- Maintains No order (Unordered)
- Heterogeneous
- Max one null value can be added

```
p.s.v.m (S... a)
{
    HashSet<String> al = new HashSet<String>();
    al.add("deepak1");
    al.add("deepak2");
    al.add("deepak2");
    al.add("deepak3");
    Iterator<String> itr = al.iterator();
    while(itr.hasNext())
    {
        sop(itr.next());
    }
}
```

~~will be stored~~

→ This will not be added. add will return false as ele is already present

```
deepak2
deepak3
deepak4
deepak1
```

add(E e)
remove(Obj)
clear()
Size()
Search → contains(obj)
isEmpty()
iterator()

No Random Access of ele by Key or index

Iterable
↑
Collection
↑ extends
Set
↑ implements
AbstractSet
↑
HashSet
↑
LinkedHashSet
↓
Same as HashSet but maintains insertion order.

or sop(al);

o/p: [deepak2, deepak3, deepak4, deepak1]

LinkedHashSet<String> al = new LHS<String>();

same as hashset but maintains ascending ord? & dzn't allow any null value.

- ordered (sorts value of similar D.T.) → maintains ascending order.
- Homogeneous
- → only unique elements.
- → doesn't allow null values

TreeSet

→ Maintain ASC ord

→ NO null value

```
TreeSet ts= new TreeSet();
        ts.add("Deepak");
        ts.add("Bhaskar");
        ts.add("Haseena");
        ts.add("Surya");

    for(Object ob: ts)
      {  sop (ob);
      }
```

Ascending order of

TreeMap

→ Same as hashmap but maintains ascending ord
It can not have any null key but can hv multiple null values

→ contains only unique ele

TreeMap <S,I> hm = new TM

① performance:
(Insertn, deln, search)

HashTable < Colln. SynchronizedMap < ConcurrentHashMap < HashMap

TreeMap < HashMap

② but faster if you need to traverse key in sorted order

Non Generic vs Generic Colln

before JDK 1.5 colln F/W was non-generic since 1.5 its gen

→ Generic Colln allows you to hv only one type of object in coll

non-generic `Arraylist al = new AL();`

generic `Arraylist <String> al = new AL<String>();`

we specify the type in angular braces.

if you try to any other type of obj ⇒ compile time error.

index → 0   1

al1 → "ravi", "vijay", "Ajay" , al2 = "Ravi", "Hanumat"

al1.addAll(al2) → ravi, vijay, Ajay, ravi, Hanumat

al1.removeAll(al2) → Vijay, Ajay

al1.retainAll(al2) → Ravi

al2.add("Ali") → Ravi, Hanumat, Ali

al.add(123) / al.add('x') → all these primitive D.T. is automatically converted into obj of that type by JVM.
⇓
called autoboxing

al.add(2, "charan") → 2nd posn

al.remove(123) → index

collections.sort(al1);
  sop(al1); → ele sorted.

~~sop~~ al2.size() ⇒ 2

al2.clear().size() ⇒ 0

String a[] = al1.toArray()

al1.contains("ravi") → true

al2.isEmty() → false

al2.get(1) → Hanumat

al.add(230);
⇓
is converted internally to
al.add(Integer.valueOf(230))

internally uses red black tree to sort the ele in natural order.

Switch case can be used → switch-case is faster than if else when too many cond^n^s

:→ to handling diff^t type of browser — IE, FF, chrome, Safari

→ to handle diff^t type of locators ✓ return by.xpath()

```
p s v m (s...a)
{
        char grade = 'C'
        switch (grade)
        {
            case 'A':
                sop ("excellent");
                break;
            case 'B':

            case 'C':
                sop ("Well Done");
                break;
            case 'D':
                sop ("You Passed");
            case 'E':
                sop (" Better try Again");
                break;
            default:
                sop ("Invalid Grade");
        }
        sop ("your grade is:" +grade);
}
```

if D is true → both D & E will be printed

Hash-collision in Hashtable

How it is handled in Java.

if 2 diff Keys hv the same hash value then it lead to hash collision. A bucket of type linked list used to hold the diff^t keys of same hash value.

Internal working of HashSet

HashSet internally uses HashMap to maintain the uniqueness of elements.

↳ when we create obj of hs an obj of hm is created. and ~~value~~ item added in hs is added in hm as Key with a dummy value.

HashMap works on the principle of 'Hashing'.

we need to understand 3 terms to understand hashing.

→ Hash Function: hashCode() returns Integer value

→ Hash value: int value ret. by H.F. → points to 'Key'

→ Bucket: stores Key-value pair.

Remove duplicates from AL. without using coll^n (without using set)

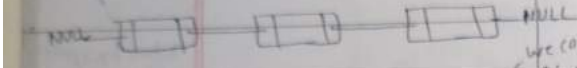
```
sop("Before:" +al);
for (int i =0; i<al.size(); i++)
{   for (int j=i+1; j<al.size(); j++)
    {   if (al.get(i).equals(al.get(j)))
            {al.remove(j);}
            j--;
}}
sop("After:" +al);
```

using LinkedHashSet → will remove duplicates → will maintain insert^n order ⇓ HashSet will not maintain insert^n order

```
LinkedHashSet lhs = new LHS();
lhs.addAll(al);
al.clear();
al.addAll(lhs);
Sop("After:" + al);
```

Java LinkedList class vs AL

Internally uses doubly linked list to store the ele.
→ duplicates allowed
→ maintains insert^n order
→ is non-synchronized
→ manipul^n fast as no shifting occurs
↳ can be used a list, stack or queue.



✓ LinkedList al = new LinkedList();

we can pass AL obj al here.

```
al.add("abc");
al.add("xyz");
```

AL RandomAccess of ele is allowed.
LL → Not allowed.

↳ ArrayList better for storing & accessing the data ↳ but Linked List is better for manipulating the data.
↳ LL consumes more memory than AL.

List Interface methods

can be used for ArrayList as well.

```
add(index, obj) → int
addAll(index, Coll^n)
get(index)
set(index, object)
remove(index)
```

listIterator()

ListIterator interface methods

✓ hasNext(), next()
hasPrevious(), previous()

```
ListIterator ltr=all.listIterator()
while (ltr.hasPrevious())
        sop(ltr.previous());
```

Sorting in Collection:

we can sort the ele of:

1) String objects
2) Wrapper Class objects (Integer etc)
3) user defined class obj.

if coll^n is of Set type to sort TreeSet
Map → TreeMap
but for List we can not sort elements
method for sorting list ele:-

Collections.sort(List l) :→ sorts elements of list.
→ List ele must be of comparable type.
Here Collections is a class

Map Interface Methods

→ useful if you try to search, delete, update elements on basis of Key.

Methods in Map Interface

	put(obj, obj) → insert / replace
	putAll(Map)
	remove(obj) → delete
	get(Obj Key) → returns value of specified key
T/F	containsKey(Key) → Search key
	✓ containsValue(value)
Set	KeySet() → return Set of keys
	values() → ——— values
Set	entrySet() → return Set of all keys & values

Entry Interface
↳ SubInterface of Map

Map.Entry Interface methods:
getKey() & getValue()

HashSet vs HashMap
HashSet ↓ contains only values
Random Access of ele Not Allowed.
HashMap → contains 2 things Keys & values
Allowed

LinkedHashMap<St..> lhm = new LHM..

```
Collections.sort(al);
```

for non-generic coll^n

```
while (itr.hasNext())
{ Object ob= itr.next()
        sop(ob);
          or
        sop(itr.next());
}
```

Comparable Interface:

- used to sort the objects of user defined class.
- contains only one method compareTo(obj)
- It provide single sorting sequence means you can sort the ele based on single data member only i.e. name, age or salary etc.

Sorting on the basis of Age

```
class Student implements Comparable<Student>
{
   int rollno;
   string name;
   int age;
 Student(int rollno, string name, int age)
  { this.rollno = rollno;
    this.name = name;
     this.age = age;
  }
 public int compareTo(Student st)
 { if(age = st.age)
      return 0;
   else if (age>st.age)
        return 1;
   else
        return -1;
 }
}
public class Sort3 {
 psvm(S... a)            AL will store user-defined class objects.
 { AL<Student> al = new AL<Stu>();
 al.add(new Student(101,"Deep", 23));
 al.add(new Student(102,"Ajay", 27));
 al.add(new Student(103,"Jai", 21));
     Collections.sort(al);
 for(Student st: al)
  sop(st.rollno+" "+st.name+" "+st.age);
```

To sort array of objects
Arrays.sort(a).

To sort coll^n of object
Coll^ns.sort(al).

List	Set
1) Allows duplicates	Not allowed
2) 3 classes, AL, LL, vector	3 classes HS, LHS, TS
3) ListIterator can traverse in reverse Dir^n	only forward.

To convert a given coll^n to synchronized coll^n

Synchronised: Coll^n classes:

Vector, Hashtable, Properties & Stack are synchronised classes

Non-Synchronised
ArrayList, Coll^ns.synchronizedColl^n(Coll^n obj);
" Map "m
" List "l
" Set "s

To make thread safe

Collections.synchronisedList (list) → makes class thread-safe
Set(set)
return type List, Set, ...

al = Coll^ns.sylist(al);

Array	Arraylist
① fixed in size or static	Dynamic in size
② store similar type of data	can store homogeneous & ~~Heterogeneous element~~
③ can contain Primitive data types	can not contain, automatic converted to Obj Ty...
④ defined	AL has lot of f^ns addAll, removeAll, iterator contain only objects

Properties class is subclass of Hashtable → used to stores Key-values in String form

String Array → Arraylist
List al = Arrays.asList(array);
↓
new String [] {"Hi", "Java"}
AL a = new AL(al);

List → Array
al.toArray()

↓

```
103 Jai 21
101 Deep 23
102 Ajay 27
```

Call^ns class : contains many useful static methods-

I. 1 Collections. sort (List l) → returns void → sorts ele in ascending order or acc. to Natural order.

2 synchronized Coll^n (c) → to get a thread-safe version of coll^n

3. unmodifiable Coll^n (c) → Read-only.

4 fill (call^n c, Obj o) replace all ele of c with o.

5. frequency (call^n c, Obj O) → ~~No. of~~ occurrance of O in c.

6. max (Call^n c) ↳ gives max ele acc. to natural ordering of ele. → returns int.

7. ~~min (call^n c)~~

8. replaceAll (Call^n c, Obj. old, Obj new) → replace all occurrence of old with new.

9. void reverse ( ) → List

10 shuffle ( ) → List ↳ randomly shuffle ele.

11 emptyList () returns empty LIST/M (Immutable)
→ " Set ()
" Map ()

wait
→ Dynamic
→ Static → Thread.sl

Dynamic → implicit W. / explicit W.

implicit W.
→ Applies on all ele
→ should be kept smaller as it makes script slower e.g. 3 sec or 5 sec
→ if implicit wait is 10 sec & ele is found immediatly when searched first time by driver driver will move on. if not found first time driver will ⇐ wait for ele complete 10 sec even if ele is found after 5 sec.

2nd check will be done only after 10 sec.

explicit W.
for part^r ele
It is a bit great than implict
if a part^r ele is t too much time to l
e.g. ~~10 sec~~
→ if explicit wait is 10 sec & ele is found after 5 s driver will mo on after 5 sec

Copy all ele of T.S. to Array.

```
Object [ ] a = ts. to Array ( )     TreeSet
for (Object ar : a)
        sop (ar);

Integer [ ] a = ts. toArray (new Integer [ ts.size() ]);
```

Call^ns. unmodifiableColl^n (Coll^n c) → returns Coll^n

How to make Coll^ns read only

Call^ns. unmodifiableMap (Map m
________ List (List l
________ Set (Set s

"UnsupportedOper^n Exception" is thrown if we try to perform add/remove op^n on read only

LinkedList
ll.getFirst ()
ll.getLast ()
" .addFirst ()
" .addLast ()
" .removeFirst ()
" .removeLast ()

```
List<Integer> al = new AL<>();
        al. add (3); al.add (6);
    sop ( Coll^ns. max (al)); ⇒ 6
al = Coll^ns. unmodifiableList (al); → returns List
        al. add (7); ⇒ UnsupportedOp^n E at compilation
```

ts. first () → lowest value
ts. last () → highest value

List Sorting

sort ( ) → Applicable for List only
Not for Set

Coll^ns. sort (al)
sop (al);
↳ sorted

```
Collection. sort (al);
sop ("ascending order:" + al)

Comparator com = Collections. reverseOrder ();
    Collections. sort (al, com);
sop ("descending order: " + al);
```