Date : / /

Collections.

→ Group of objects to be treated as a Single unit

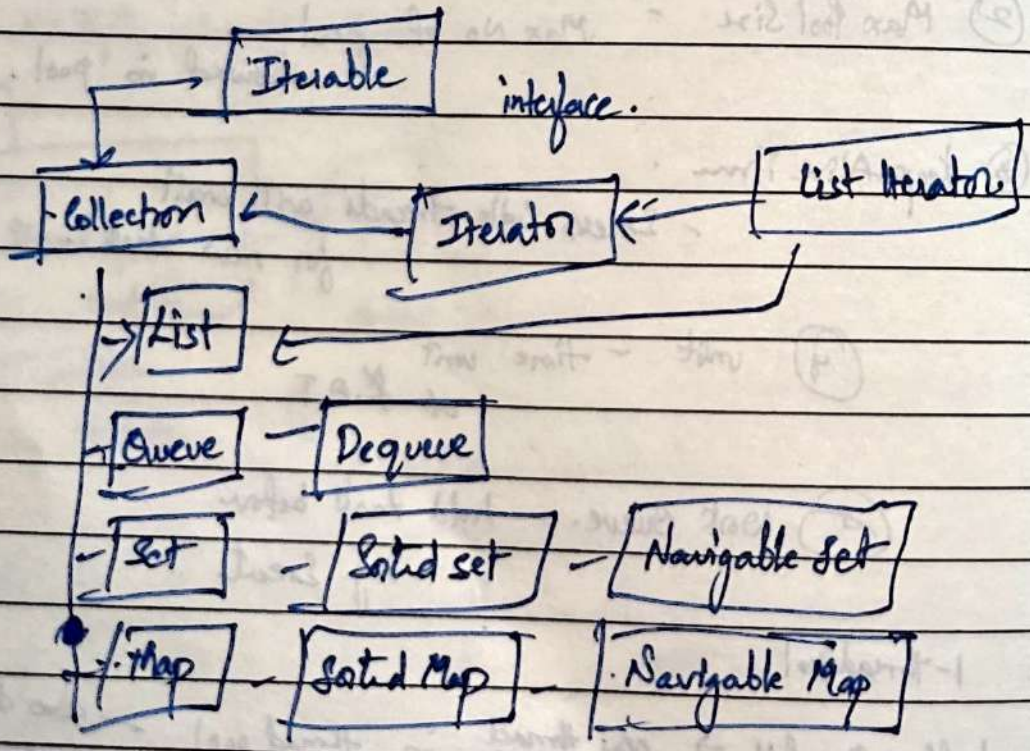
→ set of Standard utility classes for managing various kinds of Collections. provide

→ Core interfaces

→ implementations of these interfaces

→ methods found in Collections & Arrays classes that can be used to perform various operations

Eg Searching, Sorting (or) creating customised collections.



Date : / /

Iterators

implements Iterable<T>. — interface

public Iterator<T> iterator()
{
} return —;

iterator
(2 methods.
hasNext() — boolean value

next()
value at particular index,
increment index.

So for Each loop applicable
on Collection which implement
iterable interface

Date : / /

```
public class OurGenericList<T> implements Iterable<T>
{
    private T[] items;
    private int size;

    public OurGenericList()
    {
        size = 0;
        items = (T[]) new object[100];
    }

    Public void add(T item)
    {
        items[size++] = item;
    }
    Public T getItemAtIndex(int index){
        return items[index];
    }
    @Override
    Public Iterator<T> iterator() {

        return new OurGenericListIterator
                (this);

    }
```

internal implementation

Date: / /

```
private class OurGenericListIterator implements Iterator<T>
{
    private OurGenericList<T> list;
    private int index = 0;

    @Override
    public boolean hasNext() {
        return index < items.size();
    }

    @Override
    public T next() {
        return list.items[index++];
    }

    public OurGenericListIterator(OurGenericList<T> list)
    {
        this.list = list;
    }
}
}
```

for each – iterator

Date : / /

main()

{ OurGenericList<Integer> list
= new OurGenericList<>(); — object

list.add(1);
(2);
(3);

Iterator<Integer> iterator = list.iterator();

while(iterator.hasNext()) — boolean

{ sout(iterator.Next()); — value at the index.

ArrayList
Local
Extends
Iterator → method call. → return .iterator()

Date : / /

Collection

- boolean containsAll (Collection<?> c)
  - add (Collection<? extends E> c)
  - remove (Coll<?> c)
  - retain (Coll<?> c)
- void clear()

that

List - maintain Collection elements in order & can contain duplicates.

- ⇒ Elements are ordered.
- → position based.

Methods

- E get (int index)
- any type. E set (int index, E element)
- void add (int index, E element)
- addall (int index, Collection<? extends E> c)
- E remove (int index)

Date : / /

Implementations

- ArrayList
- Vector
- LinkedList.

A.L

- Dynamic Array
- Size of Array is not known.
- internally uses normal array, set to some default Capacity.
- When Capacity is reached, it will create a new array of bigger size & Copies all Elements from old to new array.

→ new array's reference is used.

Garbage Collected, previous reference.

Date : / /

Vector
- thread Safe.
- Concurrent calls to vector will not compromise its integrity
- Synchronization.

Linked List
- uses doubly linked list.
- implements deque.
- Linear time

Array list - Constant time performance.
Position based access.
frequent insertion & deletions.

best choice

, to Str() method.
to print obj

```
List<Integer> obj = new AL<>();
obj.add();
;
```

'[1,2,3]'.

Sout(obj)

Date : / /

new arraylist<>(Collection);

↓

Copy will be added to new arraylist

- addAll(Collection)

↓

added to list.

list interface extends Collection.

List Iterator

↓

Extends iterator.

hasNext();
next();

E next(); after cursor

E previous(); before cursor.

⇒ LI<E> listIterator()

LI<E> listIterator(int index)

- list iterator

List traversal

LL

↓

bidirectional

↓

doubly LL.

Date : / /

next :
return curr value & move to next value.

previous
move to previous index & return its values.

```
List<I> lk = new LL<>();
lk.add(1);
(2);
(3);

LI<Integer> iterator = lk.listIterator();
```

list.toArray(new Integer[0])
= Integer[] arr.

- creates array of Integer Contains elements of list

Date : / /

Queue → Collection.

dequeue → queue.

LinkedList → dequeue.

Array deque → dequeue.

priority queue → Queue.

boolean add (E element)

boolean offer (E element)

E poll()

E remove()

E peek()

E element()

dequeue - double ended queue.

Stack - dequeue.

LIFO - 田

FIFO. 川

boolean offerFirst (E elem)

boolean offerLast (E elem)

void push (E element)

void addFirst (E element)

Void addLast (E elem)

Date : / /

removeFirst()
removeLast()
pollFirst()
lastE()
pop()

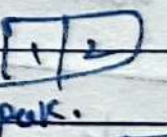
peekFirst()
peekLast().

L
FIFO
↳ Array deque (Stack)

L.L - Queue.

Queue<Integer> q = new LinkedList<>();

q.offer(1);
q.offer(2); } add.



q.peek(); → 1st element in queue = // 1

q.poll(); → remove 1st elent in queue. // 1

q.peek(); // 2.

first in first out queue

↳ queue interface - implemented by LinkedList.

Date : / /

Stack<Integer> stack = new Stack<>();

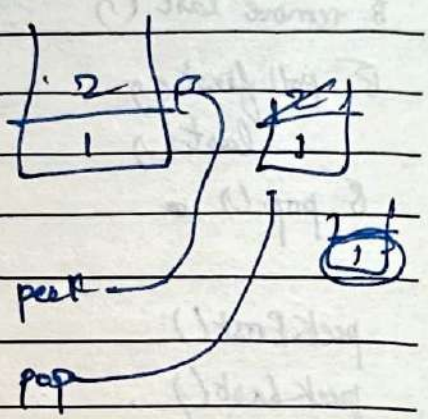
.push(1);
.push(2);

while (!stack.isEmpty())

(stack.pop())

peek

stack.pop()



Stack - Stack class

queue - LinkedList class

double ended queue - array deque

Stack & FIFO queue

deque interface - A.d implementation

Date : / /

priority Queue :-

- works on priority.
- implementation is based on priority heap.
- multiple ~~[illegible]~~ ~~priority~~ Elements having same ~~[illegible]~~ priority - arbitrarily chosen.

⇒ not Sorted

PriorityQueue<Integer> Pq = new PriorityQueue<>();

Pq.offer(); 1, 2, 0, 100.

Good idea

traverse

not use iterator.

List<Integer> top2 = new AL<>();

int index = 0

while(!Pq.isEmpty)

if(index == 2)

~~return~~

break;

top2.add(Pq.poll());

index++;

Sout(top2)

Date : / /

java Collection works on wrapper classes

We can pass Collection to priority queue to initialize.

So in order to deal with Custom class.

Collections work on object

it will Convert primitive to object (Wrapper)

based on this object we use Collection data structure to Store (or) Collect objects.

Date : / /

```
public class StudentMarks implements Comparable<StudentMarks>
{
    private int maths ;
    private int physics;

  @override
    public String toString()
        {
            return " StudentMarks[maths =" + marks +
                                    "physics =" + physics + "]";
        }

    public int getMath()                    public int getPhysics()
    {                                           {
            return maths ;                          return physics ;
    }                                           }

// Constructor.
    public StudentMarks (int maths, int physics)
{
    this.maths = maths
    this.physics = physics.
}

    public  int  CompareTo ( StudentMarks o)
                                                            other
                                        current object < Student obj
                                                              -1
                                                              >  1
    return
        O.maths - this.maths ;                                == 0
    }
                        // descending order,
```

Date : / /

```
list<StudentMarks> stMarks = new ArrayList<>();
stMarks.add(new StudentMarks(70,80));
```

- (38,10)
- (100,38)
- (40,88)
- (97,19)

```
priorityqueue<StudentMarks> spq = new PQ<>(stMarks);
```
collection

priority queue with StudentMarks initialised

```
// creating top3 student list
List<StudentMarks> top3 = new
                          AL<>();
```
// default arraylist.

```
index = 0
while(!spq.isEmpty())   // loops if element exists.
{
    top3.add(spq.poll());   if(index==3)
    index++;                    break;
}
```

// fills 3 top student from PQ, terminates after 2nd list is full.

index is built

Date : / /

sort(top2); // internal sorting is called.

// $o_1 - o_2$

$o_1 = 5$
$o_2 = 10$ $\quad 5-10 = -5 < 0$ so 5 should come before 10.

② 10 5
$10-5=5$ so 10 should come after 5.

5 5
$5-5=0$ Equal so no procedure

Single object pass

Sorting in Comparator

so 1st object StMark, o_1
2nd object StMark o_2

o_2 - current object o_1 is previous.

$o_1 - o_2$.

if $o_1 - o_2 = 0$ any one is placed
> 0 o_2 is placed first.
< 0 o_1 is placed first.

object
o_1. math
o_2. math.

$o_1 - o_2$ — Ascending order

$o_2 - o_1$ — descending order

Date : / /

iterable

iterator implement

to iterate or traverse

Comparable

CompareTo implement.

internally CompareTo is implemented by wrapper class in ascending order

Exception arise if wrapper for customclass object because CompareTo not implemented.

Custom class obj Compare.

To implement Comparator <>

we need to implement Compare method.

Compare.To(pass object)

Compare method pass 2 objects.

Comparator : Compare 2 object

Comparable : CompareTo natural ordering either ascending (or) descending.

- sorting by using Custom logic.
- item by name etc.

Date : / /

marks based sorting.

class Comparable implement.

Flexibility & Comparing strategy.

Comparator implement

Comparator interface

Compare To

Comparator will be passed to Queue to total ordering.

Comparator strategy if provided over natural order if

class itself defines natural order.

Comparator Pass ?

Yes

Compare() Strategy

No

class - Comparable implement ?

Yes

Compare To Call.

No

ClassCast Exception

Date : / /

Ex PQ<Integer> = new PQ<>();

offer // 0, 1, 2, 100 order

PQ<Integer> = new PQ<>(new Comparator());

descend order logic.

poll

[0, 1]

// 100, 2, 1, 0

poll. [100, 2].

top 2 element

Custom logic

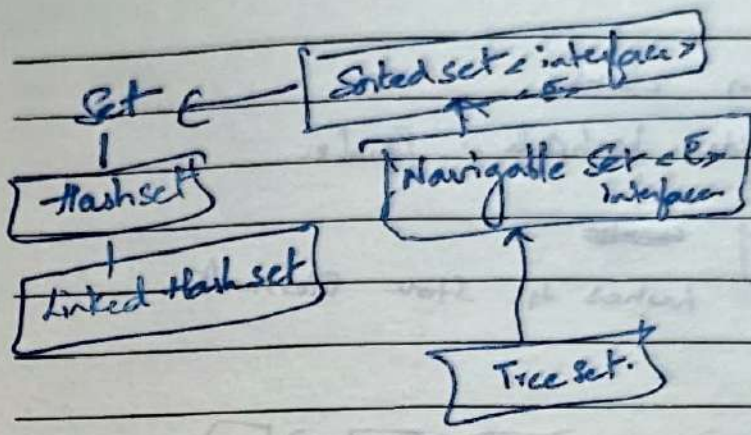
Collections.sort(Collection, Class)

Class implement Comparator interface

pass - Total ordering

precedence over natural ordering..

Date : / /



containsAll (Subset)
addAll (union)
removeAll (difference)
retainAll (intersection)
clear() (Empty Set).

Set - unique Elements.
add Elements - duplicate values removed.

Set can take Collection to initialize.

1 2 3 4 - set 1
2 3 4 set 2
set1.retain(set2).
set1 - 2 3 4.
set2 - 2 3 4.

Constant time insertion.
- no ordering.
- insertion order not maintained.

Linked - HashSet
| internally LinkedList.
Extends HashSet.
| insertion order maintain.

Date : / /

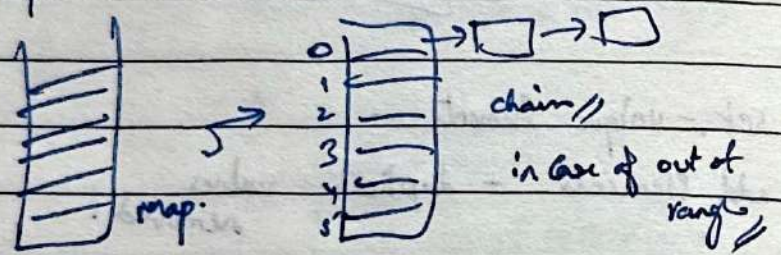
HashSet, Custom class

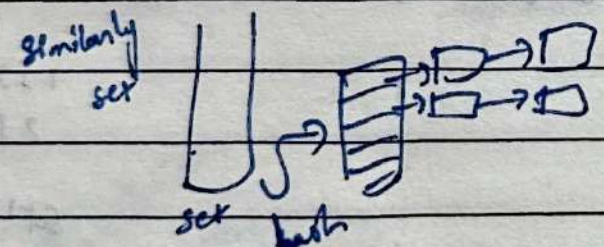
Override hashCode, Equals.

At hashset internally

hashes & store Elements

Ex hashmap.





Contains method, Go to that hash region — Compare object — if true return true.

if anonymous objects are passed.

obj — Object Class.

So typecast into Custom class type & use that object use member variable.

Date : / /

Sorted Set (Inteface)

- provide functionality for handling sorted sets.
- sorts the set — Elements sorted.
  Accessing according to ordering used by set.
  - E first()
  - E last().

Navigable Set Inteface. :

Extends Sorted Set Inteface by Implementing navigation methods.
to find matches for Specific Search targets.

- Navigation
  - operation that require searching for Elements in Navigable Set.
- absence of Element - null.

addition to Sorted Set Inteface → Navigable Set inteface add additional methods.

Date : / /

Either Custom class - Comparable interface.

Compare To method

(or)

Comparator interface

Compare by 2 objects.

(Use)

Eg: Set<StudentMarks> treeset = new TreeSet<>

treeset.add(new StudentMarks(70,80));

anonymous object

↓

Object Class.

((S1, S2) ->

S2.getPhysics() - S1.getPhysics())

desired order.

Comparator.

↓

Lambda

↓

functional interface

for(StudentMarks X : treeset)

Sout(" " + X);

↓

desired order based on physics marks

Comparator(method)

Custom Compare fn.

Date : / /

natural order - defined in class
+
insertion
Already.

Total order - defined in the interface
+
insertion
& passed into object as parameter.

>> High precedence -

object - Constructor.

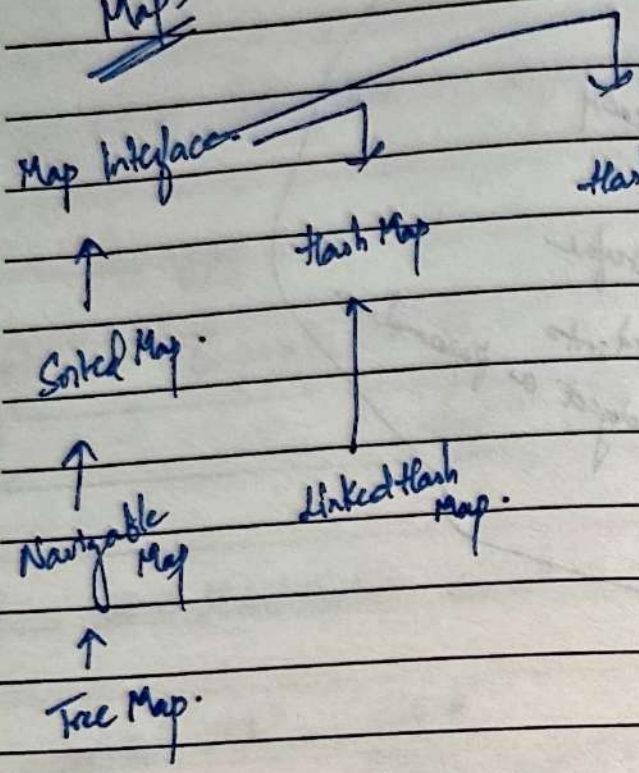
floor - less or equal to.

higher - higher than current

ceil - round off.
to next higher.

Date : / /

Maps



Map defines mappings from keys to values.

<Key, Value> — Entry.

- no duplicate key.

Key, value - object.

Map - not Collection.

Map Interface doesnot Extend Collection.

Key Set,

Value Collection,

Entry Set -

Date : / /

Object put (K key, V Value)

Object get (Object key)

Object remove (Object key)

boolean Contains key (Object key)

Contains Value (Object Value)

int size ()

boolean isEmpty () .

Set - Collection of elements .

Map - Entries .
- Keys
- Values .

void PutAll (Map<? extends K, ? extends V> map) .

void clear()

Set<K> keySet()

Collection<V> values()

Set<Map.Entry<K, V>> entrySet() .

Date : / /

Values - may be duplicate.

Entry in Entry Set can be manipulated.

Each <key, Value> in entry set view represented by object implemented in Nested Map.

Maps Implementation

HashMap
HashTable } - Unordered Maps.

LinkedHashMap - Ordered Maps.

TreeMap - Sorted maps - implement.

HashMap class - not thread safe.
It permits one null key.

HashTable - thread safe.
permits not null K, V only.
- (performance Penalty) - thread safe.

Date : / /

LinkedHashMap - Insertion order.
maintain access order.
least recently accessed.
most recently

ordering mode
4 (LRU Cache)

Hashtable - linear time.

SortedMap - key - sorted.

Search based on key.

TreeMap - Implements NavigableMap Interface
↳ SortedMap Interface

Operations - natural ordering

total order can be passed by using customized Comparator & pass it to Constructor.

Date : / /

Tree Map - uses balanced trees.

- Excellent performance

Hashmap Search fast than Tree Map.

2 balanced trees (red).