Collection

- Array is an indexed collection of fixed no. of homogeneous data elements.
- The main advantage of arrays is we can represent multiple values by using single variable so that readability of code will be improved.

* Limitation of Array:→

1) Arrays are fixed in size i.e. once we create an array then no change of increasing or decreasing of size based on our requirements due to this to used arrays concept compulsory we should know size in advance which may not be possible always.

2) Array can hold only homogeneous datatype elements.

Ex: Student [] S = new Student [10000];
S[0] = new Student(); ✓
S[1] = new Customer(); ✗
↳ CE → req. student.

- we can solve this problem by using obj type of arrays.

object [] a = new object [10000];
a [0] = new Student [];
a[1] = new Customer();

3) Arrays concept is not implemented based on some std. data structure & Hence readymade method support is not available for every requirement we have to write code explicitly which increases complexity of programming.

- To overcome above problems of Arrays we should go for 'Collections' concept.

1) Collections are growable in nature i.e. based on our requirement we can increase or decrease the size.

2) Collections can hold both homogeneous & heterogeneous element objects.

3) Every collections class is implemented based on some data structure hence for every requirement readymade method is available being a programmer we are responsible to use those method & we not responsible to implements those method.

Scanned by CamScanner

* Difference :→ Arrays & Collections.

Arrays	Collections
1) Fixed in size	1) Growable in nature
2) Homogeneous only	2) Both homo & hetro obj.
3) with respect to memory, arrays are not recommended to use	3) with respect to memory collections are recommended to use.
4) with respect to performance, arrays are recommended to use.	4) with respect to performance coll^n are not recommended to use.
5) There is no underline as for arrays & hence readymade method is not available.	5) every coll^n class is implemented based on some data structure & hence for every req readymade.
6) Arrays can hold both primitive & objects.	6) Coll^n can hold only object types but not primitives.

* Collection :-

- If we want to represent group of individual objects as a single entity then we should go for coll^n.

* Collection Framework :→

- It contains several classes & interfaces which can be used to represent a group of individual object as single entity

Java	C++
Collection →	Container
Coll^n framework →	STL (Std. Template Lib)

* 9 Key Interfaces of Collection Framework :-

1. Collection
2. List
3. Set
4. SortedSet
5. NavigableSet
6. Queue
7. Map
8. SortedMap
9. Navigable Map

1. Collection (I) :→

- If we want to represent group of individual obj. as a single entity then we should go for coll^n.

Scanned by CamScanner

- Collection Interface defines the most Comm methods which are applicable for any Coll obj.

- In general coll^n interface is consider as root interface of coll^n framework.

- There is no concrete class which implemen Collection Interface directly.

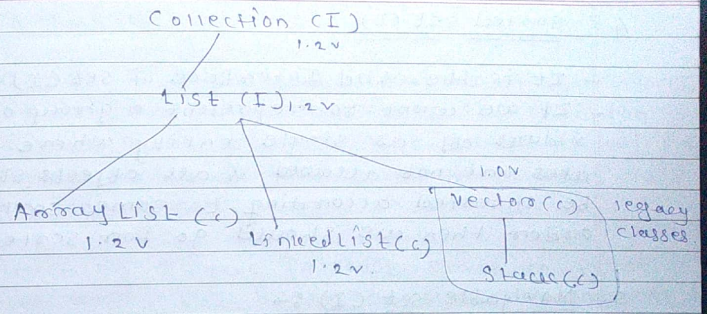
Difference between Collection & Collections:

* **Collection** :→ is an interface. If we want to represent group of individual objects as a single entity then we should go for Collection.

Collections :→ is an utility class, present in java.util package to define several utility methods for coll^n objects (like sorting, searching).

2) List (I)

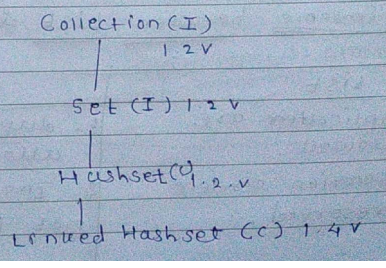
- It is the child interface of collection (I)
- If we want to represent a group of individual objects as a single entity where duplicate are allowed & insertions order must be preserved then we should go for list (I)



Note :→
In 1.2v vector & stack classes are re-engineered (modified) to implement List Interface.

3. Set (I)

- It is the child interface of coll^n (I)
- If we want to represent a group of individual objects as a single entity where duplicates are not allowed & insert in a single entity where duplicates are not allowed & insertion not required then we should go for set.



Scanned by CamScanner

4. Sorted set (I)

- It is the child Interface of set (I)
- If we want to represent a group of Individual obj. as single entity where duplicates are not allowed & all objects should be inserted according to some sorting order then we should go for sorted set.

5. Navigable set (I) :→

- It is the Child Interface of sortedset (I)
- It Contains several methods for navigation purposes.

Collection (I) 1.2v
|
Set (I) 1.2v
|
SortedSet (I) 1.2v
|
Navigableset (I) 1.6 v
|
Treeset (C)
1.2v

* Difference between List & Set

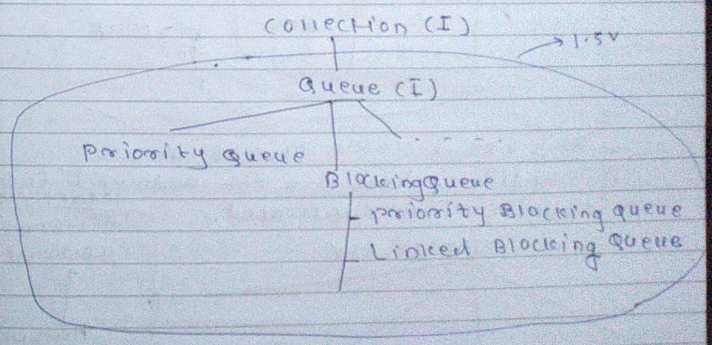
List	Set
1. duplicates are allowed	1. duplicates are not allowed
2. Insertion order preserved.	2. Insertion order not preserved.

6. Queue (I)

- Queue is the child interface of collection(I)
- It is the child interface of collection
- If we want to represent group of individual object prior to processing then we should go for Queue.

- Usually Queue follows first in first out order but based on our requirements, we can implement our own priority order also.

ex: before sending a mail all mail id's we have to store in some data structure.
- In which order we ordered mail id's in the same order only, mail should be delivered for this requirement queue is best choice.



Scanned by CamScanner

Note :→ All the above interfaces (Collection, List, set, sortedset, navigable set & Queue) meant for representing a group of individual objects.

- If we want to represent a group of object as key value pairs then we should go for Map.

7. **Map (I)**

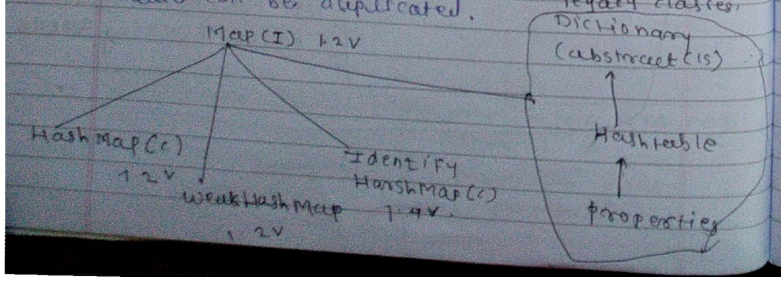
- It is not child interface of collection.
- If we want to represent group of objects as key value pairs then we should go for map.

Ex:-

Key	Value
Sr. No	Name
101	durga
102	Ravi
103	Shiva

→ Map

- Both key & value objects only
- Duplicate keys are not allowed but value can be duplicated.



8. **Sorted Map (I)**

- It is a child interface of Map (I)
- If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap

9. **NavigableMap (I)**

- It is the child interface of SortedMap (I)
- It defines several methods of navigation purposes.

Map (I) 1.2V
↓
Sorted Map (I) 1.2V
↓
NavigableMap (I) 1.6V
↓
TreeMap (c)
1.2V

Note :→ The following are legacy character present in collection framework

1. Enumeration (I)
2. Dictionary (AC)
3. Vector (c)
4. Stack (c)
5. Hashtable (c)
6. properties (c)

Scanned by CamScanner

* If we want to represent a group of Individual object as a single entity then we should go for collection.

- Collection interfaces defines the most common methods which are applicable for any collection objects.

1) boolean add (object o)
2) boolean add (collection c)
3) boolean remove (object o)
4) boolean removeAll (collection c)
5) boolean retainAll (collection c) → to remove all obj. except those present in c.
6) void clear()
7) boolean contains (object o)
8) boolean containsAll (collection c)
9) boolean isEmpty()
10) int size();
11) object [] toArray();
12) Iterator iterator();

Note :→ There is no concrete class which implements collection interfaces directly.

1) List (I) :→

- List is the child interface of collection. If we want to represent a group of individual objects as a single entity where duplicates are allowed & insertion order is preserved.
Then we should go for List.

- we can preserve insertion order via index & we can diff^n duplicate obj by using index hence index will play very important role in list.

- List interface defines the following specific methods

1. void add (int index, object o)
2. boolean addAll (int index, collection c)
3) object get (int index)
4. object remove (int index)
5. object set (int index, object new) - to replace the element present at specified index with provided object & return the object.
6. int indexOf (object o) - returns index of first occurance of 'o'
7. int Last Index of (object o)
8. ListIterator ListIterator();

Scanned by CamScanner

Collection

List (I)

Array List (C) LinkedList (C) Vector (C)

Stack (C)

1. Arraylist (C)

- The underlined data structure is resizeble array or growable array.
- Duplicates are allowed
- Insertion order is preserved
- Heterogeneous object are allowed (except treeset & treemap everywhere heterogeneous objects allowed).

* Constructors:

1. ArrayList l = new ArrayList();

- It creates an empty arraylist obj with default initial capacity 10.
- once arrylist reaches its max capacity then now arraylist object will be created with

$$\boxed{\text{new capacity} = \left(\text{current capacity} * \frac{3}{2}\right) + 1}$$

2. ArrayList l = new Arraylist (int initial capacity);

- It create, an empty arraylist obj with specified initial capacity.

3. Arraylist l = new Arraylist (Collection c)

- It creates an equivalent arraylist obj for given collection.

* usually we can use coll^n to hold & transfer objects from one location to another location (container) to provide support for this requirement every coll^n class bydefault implements serilizable & clonable interface.

- Arraylist & vector classes implements random access interface so that any random element we can access with the same speed.

* Random Access :-

- Random Access Interfaces present in Java util package & it does't contain any method
- It is a marker interface, where required ability will be provided automatically by JVM
- ArrayList is best choice if our frequent option is retrival operation because arrayList implements Random access interface.

Scanned by CamScanner

- ArrayList is worst choice if our frequent opn is insertion or deletion in middle.

- By default arrayList is non-synchronized but we can get synchronized version of arraylist object by using synchronized List method of collection class.

Syntax :→

```
public static ~~void~~ List synchronized
                        List (List l)

ArrayList l = new ArrayList();
List l1 = Collections.synchronizedList(l);
```

synchronized non synchronized

- similarly we can get synchronized version of set & map obj by using the fonn methods of collnn class.

1) public static set synchronized set (set s)
2) public static Map synchronizedMap (Map m)

* Diffn betn ArrayList & vector

ArrayList	Vector
1. Every method in array List is non-synchronized	1. Every method in vector is synchronized
2. At a time multiple threads allowed to operate an arraylist obj & hence it is not thread safe.	2. At a time only one thread is allowed to operate on vector object & hence it is a thread safe.
3. Relatively performance is high because threads are not required to wait to operate all arraylist object.	3. Relatively performance is low because threads are required to wait to operate all vector object.
4. Introduced in 1.2V & it is non legacy.	4) Introduced in 1.0V & it is legacy

\# Linked List :→

- The underlined data structure is doubly linked list
- Insertion order is preserved
- Duplicates are allowed.
- Heterogeous objects are allowed.
- Null insertion is possible.

Scanned by CamScanner

- LinkedList implements serializable & Clonable interfaces but not random access.

- linked List is the best choice if our frequent opⁿ is insertion or deletion in the middle.
- Linked List is the worst choice if our frequent opⁿ is retrival opⁿ.

* Constructors :→

① LinkedList l = new LinkedList ();
② LinkedList l = new LinkedList (Collection);

* LinkedList Class specific Methods :

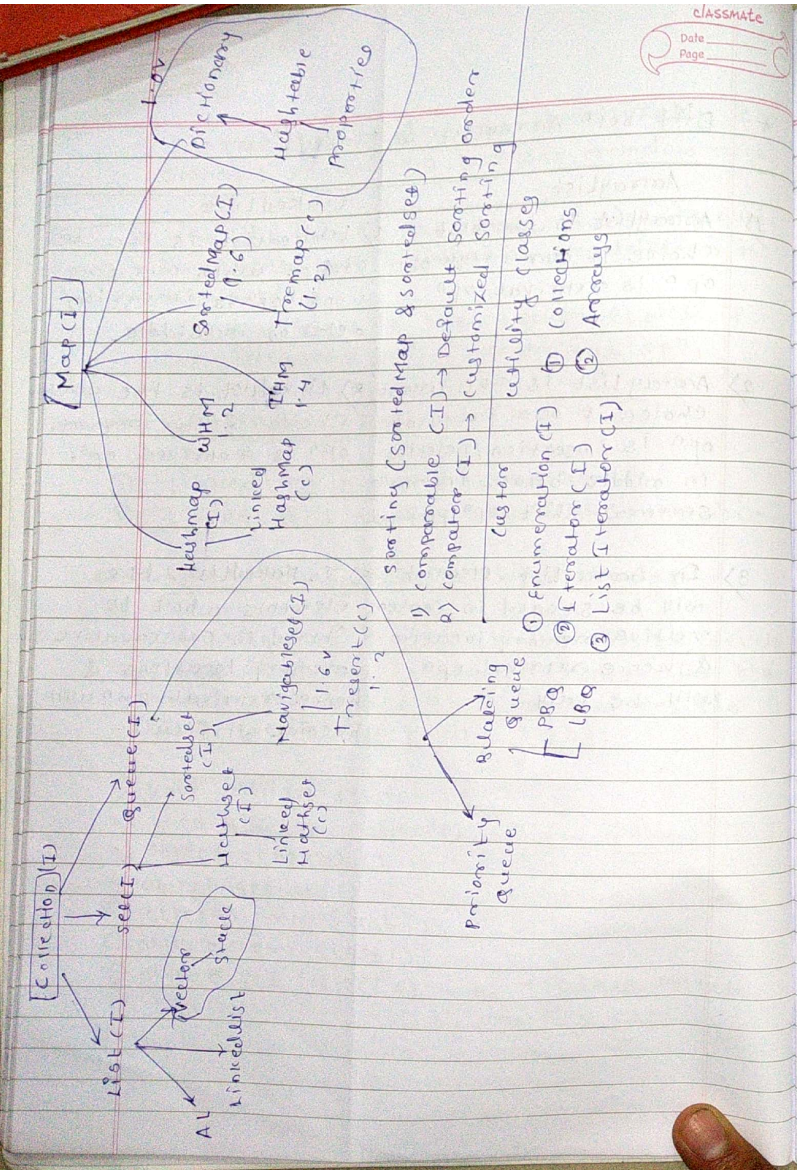
- usually we can use linkedList to develop Stacks & Queues to provide support for this requirement linkedList class defines the follⁿ specific methods

1) void addFirst (object o)
2) void addLast (object o)
3) object getFirst ()
4) object getLast ()
5) object removeFirst ()
6) object removeLast ()
7) object set (object o) → to replace object by another.

* Diff betⁿ ArrayList & LinkedList.

ArrayList	LinkedList
1) ArrayList is the best choice if our frequent opⁿ is retrival opⁿ.	1) LinkedList is the best choice when our frequent opⁿ is insertion/deletion in middle.
2) ArrayList is the worst choice if our frequent opⁿ is insertion/Deletion in middle becaus internally several shift opⁿ performed.	2) LinkedList is the worst choice if our frequent opⁿ is retrived opⁿ.
3) In ArrayList, element will be stored in conservative memory location & hence retrival opⁿ will be easy.	3) In linkedList, the elements won't be stored in conservative memory locations & hence retrival opⁿ will become difficult.

Scanned by CamScanner



* Vector

- The underlined data structure is resizable or growable Array.
- Insertion order is preserved
- Duplicates are allowed
- Heterogeneous objects are allowed
- Null insertion is possible
- It implements serizable, clonable & random Access I.
- Every method present in the vector is synchronized & hence vector object is thread safe

* Constructors :→

① vector v = new vector();
- Creates an empty vector object with default initial capacity 10. once vector reaches it's max capacity then new vector object will be created.

New Capacity = Current capacity * 2

② vector v = new vector (int initial Capacity)
- Creates an empty vector object with specified initial C.

③ vector v = new vector (Collection c)
- Creates an equivalent vector for given Collection.

Scanned by CamScanner

This constructor ment for interconversion between collection object.

④ vector v = new vector (int initial capacity, int incremental capacity);

* **Vector specified Method :-**

to add object

① add (object o) → collection
② add (int index, object o) → List
③ addElement (object o) → vector.

- **to remove object :**

① remove (object o) → collection
② removeElement (object o) → vector
③ remove (int index x) → List
④ removeElementAt (int index) → vector
⑤ clear () → collection
⑥ removeAllElements () → vector

- **to get object :**

① object get (int index) → List
② object elementAt (int index) → vector
③ object findElement () → vector
④ object lastElement () → vector

Other Methods

① int size ()
② int capacity ()
③ Enumeration elements ()

* **Stack (c) :-**

- It is a child class of vector
- It is a specially designed class for last in first out order (LIFO).

Constructors :→

stack s = new stack ();
default constructor

			→ index
1	C	2	
2	B	1	s.search ("A") = 2
3	A	0	s.search ("Z") = -1

Other Methods :→

① object push (object o) - insert an object into stack
② object pop () to remove & return top of stack
③ object peek () — to return top of stack without removed
④ boolean empty () — returns true if stack is empty
⑤ int search (object o) → returns offset if element is available otherwise returns -1

Scanned by CamScanner

* Cursors :→

- if we want to get objects one by one from the colln then we should go for cursor.
- there are three types of cursor available in java.

ⓐ Enumeration
ⓑ Iterator
ⓒ ListIterator

ⓐ Enumeration :→

- we can use enumeration to get object one by one from legacy collection objects.
- we can create enumeration obj by using elements method of vector class.

```
public Enumeration elements ();

Enumeration e = v.elements();
```

Methods :→

① public boolean hasMoreElements();
② public object nextElement();

```
ex:→ public static void main (String [] args)
{
  vector v = new vector ();
  for( int i=0; i<=10; i++)
  {
    v.addElement (i);
  }

  system.out.print (v); //[0,1,2,3,4,5,6,7,8,9,10]
  Enumeration e = v.elements();
  while ( e.hasMoreElement())
  {
    Integer I = (Integer) e.nextElement();

    if (I%2 == 0)
    System.out.print (I); // 0 2 4 6 8 10
  }
  System.out.print ; // [0,1,2,3,4,5,6,7,8,9,10]
}
}
```

* Limitations of Enumeration (1.0v)

① we can apply enumeration concept only for legacy classes & it is not a universal cursor.

② By using enumeration, we can get only read access & we can't perform remove opn.

To overcome above limitations, we should go for ~~Interato~~ Iterator.

Scanned by CamScanner

(b) Iterator (I):→

① we can applying Iterator concept for any Collection object & hence it is universal cursor.

② By using Iterator, we can perform both read & remove operations.
- we can create Iterator object by using iterator() of Collection interface.

public Iterator iterator();

e.g: Iterator itr = c.iterator();
any coll^n object

* Method :→
① public boolean hasNext();
② public object next();
③ public void remove();

ex:- public static void main(String[] args)
{
ArrayList l = new ArrayList();
for(int i=0; i<=10; i++)
{
l.add(i);
}
Sysout(l); // [0,1,2,...,10]
Iterator itr = l.iterator();
while(itr.hasNext())
{
Integer I = (Integer) itr.next();
if (I % 2 == 0)
Sysout(I); 0,2,4,6,8,10
else
{ itr.remove();
}
Sysout(l); [0,2,4,6,8,10]
}
}

* limitation of Iterator

(a) By using ~~enume~~ enumeration & iterator we can always only towards forward direction & we can't move towards backward direction.
- This are single direction cursor but not bidirectional cursor.

(b) By using iterator we can perform only read & remove operations & we can't perform replacement & addition of object.

To overcome above limitations we should go for list iterator.

Scanned by CamScanner

(c) List Iterator (I) :-

a) By using List Iterator, we can move either to the forward direction or to the backward direction & Hence it is bidirectional cursor.

b) By using ListIterator, we can perform replacement & addition of new object in addition to read & remove operations.

- we can create ListIterator obj by using List Iterator () of List (I)

public ListIterator listIterator()

ex:- ListIterator lit = l.listIterator();
(l → any List object)

* Method :-

- ListIterator (I) is the child interface of Iterator & hence all methods present in Iterator by default available to the list iterator.
- It defines following methods :-

a) public boolean hasNext ()
b) public object next()
c) public int nextIndex()
} Forward movement

d) public boolean hasPrevious()
b) public object previous()
c) public int previousIndex()
} Backward movement

a) public void remove ()
b) public void add (object o)
c) public void set (object o)
} Extra operations

Ex:- public static void main (String[] args)
{
LinkedList l = new linkList();
l.add ("balakrishna");
l.add ("venki");
l.add ("chiru");
l.add ("nag");

~~Sout~~ sysout (l);
// [balakrishna, venki, chiru, nag]
ListIterator ltr = l.listIterator();

while (ltr.hasNext())
{
String s = (String)ltr.next();
if (s.equals ("venki"))
{
ltr.remove ();
// balakrishna, chiru, nag]
}

Iterator (I)
|
ListIterator (I)

Scanned by CamScanner

```
else if (s.equals ("nag"))
{ itr.add ("Charitu");
    // [balakrishna, chiru, nag, Charitu]
}
else if (s equals ("chiru"))
    // [balakrishna, charan, nag, Charitu]
{ itr.set ("charan");
}
}
Sysout (l);
// [balakrishna, charan, nag, Charitu]
}
```

* **limitations of ListIterator**

@ The most powerful cursor is List Iterator but its limitations is, it is applicable for only List object.

* **Diff^n bet^n Cursors :->**

property	Enumeration	Iterator	ListIterator
1) where we can apply	only for legacy classes	for any collection object	for only List object
2) It is legacy?	yes (1.0v)	No (1.2v)	No (1.2v)
3) movement	Single direction (only forword direc^n)	single direction (only forword)	Bidirectional
4) Allowed op^n	only Read	Read / Remove	Read / Remove Replace / Add
5) How we can get?	By using elements() of vector class	By using iterator() of collection (I)	By using ListIterator() of List (I)
6) Methods	2 - methods hasmoreElements() nextElement()	3 - methods hasNext () next() remove ()	9 methods

Scanned by CamScanner

* Internal Implementation of cursor

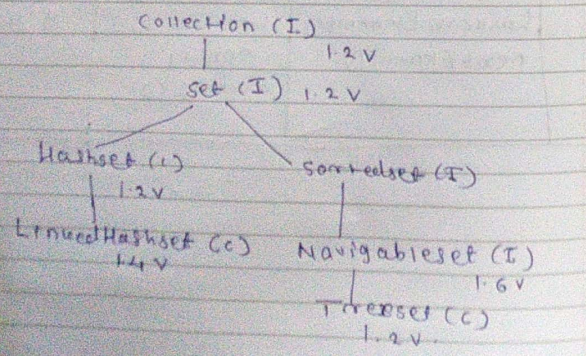
Prog → p.s v main (String [] args)

```
{
    vector v = new vector();
    Enumeration e = v.elements();
    Iterator itr = v.Iterator();
    ListIterator litr = ~~v.iterator();~~ v.ListIterator();
    sysout (e.getclass().getName());
    sysout (itr.getClass().getName());
    sysout (litr.getclass().getName());
}
}
```

o/p

```
java.util.vector $1
java.util.vector $Itr
java.util.vector $ListItr
```

* Set (I) :→



- Set is child Interface of collection (I)
- If we want to represent a group of Individual objects as a single entry where duplicated are not allowed & insertion order not preserved.

- Set Interface doesn't contain any new method & we have to use only collection Interface methods.

① **Hashset (c)**

- The underlying data structure is Hash table
- Duplicate objects are not allowed.
- Insertion order is not preserved & it is based on hashcode of objects.
- Null insertion is possible (only once)
- Heterogenous object are allowed
- implements Serializable, Clonable, but not Random Access interface.
- Hashset is the best choice if our frequent opn is search operation.

* **Note**

In hashset, duplicates are not allowed, if we are trying to insert duplicates then we won't get any CE or RE errors & add() simply returns false.

Scanned by CamScanner

Ex: Hashset h = new Hashset();
Sopln (h.add("A"); ✓ true
Sopln (h.add("A"); ✗ false.

* Constructors :→

① Hashset h = new Hashset();

- Creates an empty hashset object with default initial capacity 16 & default fill ratio 0.75.

② Hashset h = new Hashset object with specifi initial capacity & defaut fill ratio 0.75.

③ Hashset h = new Hashset(int initial capacity, float fillratio)

- creates an empty hashset object with specified initial capacity & specified fill ratio.

④ Hashset h = new (hashset (collection ();

- creates an equivalent Hashset for the given collection.
- This constructor meant for interconversion between collection object.

* Fill Ratio | Load factor :→

- After filling how much ratio, a new hashset obj will be created, this ratio is called fill ratio or load factor.

For exc :→ fill ratio 0.75 means after filling 75% ratio a new hashset obj will be created.

Ex :→ p.s.v. main (String [] args)
{
Hashset h = new Hashset();
h.add("B");
h.add("C");
h.add("D"); , h.add("2"); , h(add(null);
h.add(10); sout (h.add("Z"); // false.
Sysout (h); // [null, D, B, C, 10, Z]
}
}

② LinkedHashset (c)

- It is Child class of Hashset
- It is exactly same as Hashset (including constructor & methods) except the follo differences.

Scanned by CamScanner

Hashset	Linked Hashset
① The underlying ds is hashtable	① The underlying data structure is combination of LinkedList & Hashtable.
② Insertion order not preserved	② Insertion order preserved
③ Introduced in 1.2V	③ Introduced in 1.4V

- In above program, if we replace Hashset to the LinkedHashset then the output is [B, C, D, Z, null, 10] i.e. insertion order preserved.

Note :-

- In general, we can used LinkedHashset to develop cache based application, where duplicates are not allowed & insertion order preserved.

③ SortedSet (I)

- It is the child interface of set.
- If we want to represent a group of individual objects according to some sorting order without duplicates then we should go for sortedset.

Methods :→

SortedSet (I) defines the foll^n specefic methods.

① object first();→ returns first element of sortedset.

② object last();→ returns last element of sortedset

③ SortedSet headSet (object obj);→ returns SortedSet whose elements are less than obj

④ SortedSet tailSet (object obj);→ returns SortedSet whose elements are >= obj.

⑤ SortedSet subset (object obj1, object obj2)
- returns SortedSet whose elements are >= object1 & < obj2.

⑥ Comparator comparator()
- returns Comparator object that described underlying sorting Technique. If we are using default natural sorting order then we will get null

Note :→

Default sorting order for Numbers → Ascending order.
Default sorting order for String → Alphabetical order.

Scanned by CamScanner

ex:→

① first () → 100
② last () → 120
③ headset (106) → [100, 101, 104]
④ tailset (106) → [106, 110, 115, 120]
⑤ subset (101, 115) → [101, 104, 106, 110]
⑥ comparator () → null

④ Treeset : E()

- The underlined data structure is Balanced Tree.
- Duplicates objects are not allowed
- Insertion order not preserved.
- Heterogeneous objects are not allowed otherwise we will get RE saying class Cast Exception.
- Null insertion is possible (only once)
- Treeset implement serilizable & Clonable but not random Access.

- All objects will be inserted based on some sorting order it may be default natural sorting order or customised sorting order.

Constructors :→

1) Treeset t = new Treeset ();
- creates an empty treeset object where elements will be insert according to default natural sorting order.

2) Treeset t = new Treeset (Comparator);
- creates an empty treeset object where the elements will be inserted according to customized sorting order specified by comparator object.

3) Treeset t = new Treeset (Collection c);

4) Treeset t = new Treeset (SortedSet s);

ex:→
```
public static void main (String[] args)
{
    Treeset t = new Treeset ();
    t.add ("A");
    t.add ("a");
    t.add ("B");
    t.add ("z");
    t.add ("L");

// t.add(new Integer(10)); //CCE
// t.add (null); // NPE
Sysout (t); // [A, B, L, z, a]
}
}
```

Scanned by CamScanner

Null Acceptance :→

1) For non-empty treeset, if we are trying to to insert null then we will get null pointer exception (NPE).

2) For empty treeset as the first element null is allowed, but after inserting that null if we are trying to insert any other then we will get runtime exception saying NPE.

* Note :→ until 1.6v null is alwed as first element to the empty treeset but 1.7v onwards, null is not allowed even as the first element.

Ex 2 :→ public static void main(String[] args)

```
{
    Treeset t = new Treeset();
    t.add (new StringBuffer("A"));
    t.add (  ——  "  ——  ("Z"));
    t.add (  ——  "  ——  ("L"));
    System.out.print (t);
}
}
```

O/P :- RE - Class Cast Exception

- if we are depending on default natural sorting order, compolsory the object should be homogenous & the comparable otherwise we will get RE saying CCE
- An object is to be comparable if & only if corresponding class impliment Comparable interface.
- String Class & all wrapper classes already implement comparable interface but StringBuffer class does not implement Comparable interface.
- Hence we got class cast exception in above example.

* Comparable :→ (I)

- It is present in java.lang package & it contain only one method compareTO()

Syntax :→ public int compareTO (Object obj)
object 1. compareTO (obj 2)

→ returns -ve iff obj has to come before obj 2
→ returns +ve iff obj 1 has to come after obj 2
→ returns 0 iff obj 1 & obj 2 are equal

Ex :- psv main (String [] args)

```
{
    Sysout ("A". compareTO ("Z")); -ve
    Sysout ("Z". compareTO ("K")); +ve
    Sysout ("A". compareTO ("A")); 0
```

Scanned by CamScanner

sysout ("A".compareTo(null)); RE : NPE

}

}

- if we are depending on natural sorting order then while adding object to treeset JVM will call compareTo()

Ex: Treeset t = new Treeset ();
t.add ("K");
t.add ("Z"); → True "Z" compareTo ("K");
t.add ("A"); → -ve "A" compareTo("K");
t.add ("A"); → -ve "A" compareTo ("K");
→ 0 "A" compareTo ("A");

s.o.pln(t);
[A, K, Z]

obj1.compareTo (obj2)

The obj. which is to be inserted

The object which is already inserted

Note:→ if default natural sorting order not available or if we are not satisfied with default natural sorting order then we can go for customised sorting by using comparator.

Comparable :→ meant for default natural sorting order where as

Comparator :→ meant for customised sorting order.

* Comparator (I) :→

- It is present in java.util package & it defines two methods.

1) compare ()
2) equals ()

① public int compare (object obj1, object obj2)

→ returns -ve if obj1 has to come before obj2

→ returns +ve if obj1 has to come before obj2

→ returns 0 if obj1 & obj2 are equal

② public boolean equals (object obj)

- whenever we are implementing comparator (I), compulsory we should provide implementation only for compare method & we are not required to provide implementation for equals method because it is already available to our class from object class through inheritance.

Scanned by CamScanner

* write an program to insert integer object into treeset where the sorting order is desending order.

```
Class TreesetDemo3
{
  public static void main(String[] args)
  {
   Treeset t = new Treeset (new MyCom-
                              parator());
    t.add(10); t.add(0); t.add(15);
  t.add(5); t.add(20); t.add(20);
  Sysout (t);

// if we not using new mycomparator

}  O/P is [0,5,10,15,20]
}

Class MyComparator implements Comparator
{
  public int compare (object obj1, object obj2)
  {
    Integer I1 = (Integer) obj1;
    Integer I2 = (Integer) obj2;

    if (I1 < I2)
      return +1;
  else if (I1 > I2)
        return -1;
  else
        return 0;
}}  &O/P → [20,15,10,5,0]
```

```
Treeset t = new Treeset (new MyComparator);
                                        — ①
t.add (10);
t.add (0); +ve → compare (0,10)

t.add (15); -ve → compare (15,10)

t.add (5); +ve → compare (5,10)
           -ve → compare (5,10)

t.add (20); -ve → compare (20,10)
            -ve → compare (20,15)

t.add (20); -ve → compare (20,10)
            → compare (20,15)
            → compare (20,20)
  Sopln (t); [20,15,10,5,0]

Class MyComparator implements
                        Comparator
{ public int compare (object obj1, object
                              obj2)
  {
    Integer I1 = (Integer) obj1;
    Integer I2 = (Integer) obj2;
        if (I1 < I2)
        return +1;
      if (I1 > I2)
          return -1;
      else
          return 0;
  }
}
```

Scanned by CamScanner

* Various possible implementation of compare method s →

```
public int compare (object obj1, object
                                obj2)
{
    Integer I1 = (Integer) obj1;
    Integer I2 = (Integer) obj2;
```

① return I1.compareTo (I2); → Default natural sorting order [Ascending order] [0, 5, 10, 15, 20]

② return -I1.compareTo (I2); → [Descending order] [20, 15, 10, 5, 10]

③ return I2.compareTo (I1); → [Descending order] [20, 15, 10, 5, 10].

④ return I2.compareTo (I1); → [Ascending order] [0, 5, 10, 15, 20]

⑤ return 1; [Insertion order] [10, 0, 15, 5, 20, 20]

⑥ return -1; [Reverser of Insertion order] [20, 20, 15, 15, 0, 10]

⑦ return 0; [only first element will be inserted & all remaing are duplicate] [10]

```
}
Treeset t = new Treeset (new my
                        comparator);
t.add (10); t.add (0); t.add (15);
t.add (15), t.add (5), t.add (10),
t.add (20); S.opln (t);
```

* write a program to insert String object into the treeset where all elements should be inserted according to reverse of alphabetical order.

```
public static void main( String [] args)
{
    Treeset t = new Treeset (new mycompa-
                            rator());
    t.add (" Roja");
    t.add (" ShobhaRani");
    t.add (" Riyalamari");
    t.add (" GangaBhavani");
    t.add (" Ramulamma");
    sysout (t);
}
} class MyComputator implements comparator
{
    public int compare (object obj1, object obj2)

    String S1 = obj1.toString();
    String S2 = (String) obj2;
    return S2.compareTO (S1);

    // return -S1.CompareTO(S2);
}
}
```

O/P → [ShobhaRani, Roja, Ramulamma, Rajkumari, Gangabhavani]

Scanned by CamScanner

* write a program to insert StringBuffer object into a treeset where sorting order is alphabetical order.

```
public static void main (String [] args)
{
  Treeset t = new Treeset (new mycomparator());
  t.add (new StringBuffer ("A"));
  t.add (new StringBuffer ("Z"));
  t.add (new StringBuffer ("K"));
  t.add (new StringBuffer ("L"));
  Sysout (t);
}
}
Class mycomparator implements Comparator
{
  public int comparator (object obj1, object obj2)
  {
    String S1 = obj1.toString();
    String S2 = obj1.toString();
    return S1.compareTO(S2);
  }
}
```

O/P :-> [A, K, L, Z].

Note :-> if we are depending on natural sorting order compolsory objects should be homogeneous & comparable.
- otherwise we will get rentime exception saying CCE.

If we are defining our own sorting by comparator then object A.B comparable & homogeneous i.e. we can add heterogeneous & non comparable obj also.

* write a program to insert String & StringBuffer objects into treeset where sorting order is increasing length order. If two objects having same length then consider their alphabetical order.

```
public static void main (String [] args)
{
  Treeset t = new Treeset (new mycomparator());
  t.add ("A");
  t.add (new StringBuffer ("ABC"));
  t.add (new StringBuffer ("AA"));
  t.add ("xx");
  t.add ("ABCD");
  t.add ("A");
  Sysout (t);
}
}
Class mycomparator (implements Comparator)
{
  Public int compare (object obj1, object obj2)
  {
```

Scanned by CamScanner

```
String s1 = obj1.toString();
String s2 = obj2.toString();
int l1 = s1.length();
int l2 = s2.length();
if (l1<l2)
    return -1;
else if (l1>l2)
    return 1;
    else
    return s1.compareTO(s2);
    }
}
```

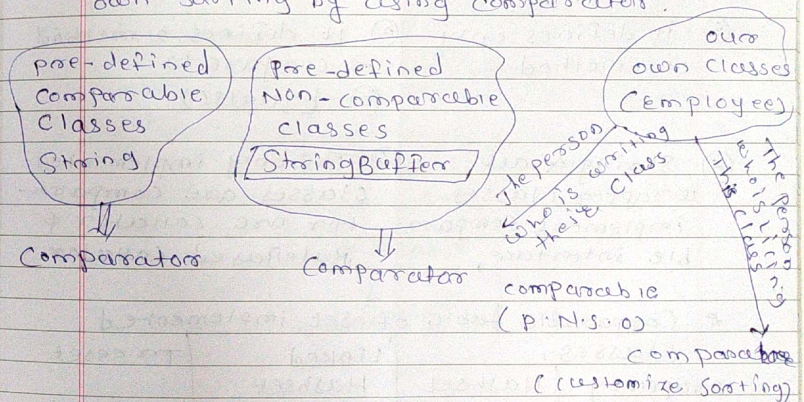
O/P :→ [A, AA, XX, ABC, ABCD]

* <u>Comparable</u> <u>VS</u> <u>Comparator</u> :→

- For predefined Comparable Classes default natural Sorting order already available.
- If we are not Sattisfied with that default natural sorting order then we can define own natural Sorting order by using Comparator
- For predefined non Comparable Classes like StringBuffer default natural Sorting order not already available.
- we can define our own Sorting by using Comparable.

- For our own classes like employee, the person who is writing that class is responsible to define default natural Sorting order by Implementing Comparable interface.

- The person who is using our class, if is not Satisfied with default natural sorting order then he can define his own Sorting by using comparator.



Scanned by CamScanner

* Comparison of comparable & Comparator.

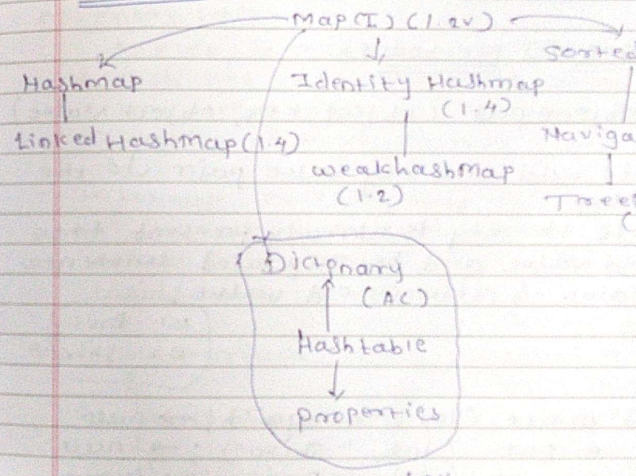
Comparable	Comparator
① It is meant for default natural sorting order.	① It is meant for customized sorting orders.
② present in java.lang package	② present in java.util package
③ it defines only one method	③ it defines 2 method ⓐ compare() ⓑ equals()
④ String & all wrapper classes implement Comparable interface.	④ The only implemented classes are comparator are collator & RuleBased collator.

* Comparable fabic of set implemented classes.

property	Hashset	Linked Hashset	Treeset
① Underlying Data structure	Hashtable	The combination of LL & hashtable	Balanced Tree
② Duplicate object	Not allowed	Not allowed	Not allowed
③ Insertion order	Not preserved	preserved	Not preserved
④ sorting order	Not Applicable	NA	Applicable
⑤ Heterogeneous obj	Allowed	Allowed	Not allowed
⑥ Null acceptance	Allowed	Allowed	For empty Treeset as first element null is allow

Notes:→ For empty treeset as first element null is allowed but this rule is applicable until 1.6 V only.

* Map (I)



- Map is not child Interface of collection
- if we want to represent a group of object as key value pairs then we should go for Map.

Ex:

key	value
101	Durga
102	Ravi
103	Shiva
104	Pavan

Scanned by CamScanner

- Both keys & value are objects only
Duplicate keys are not allowed, but values can be duplicated.
- Each key value pair is called entry
Hence Map is consider as a collection of entry objects.

Map (I) Methods :→

① object put (object key, object value)

- to add one key value pair to the Map
- if the key is already present then old value will be replaced with new value & returns old value

101 - Ravi
102 - Shiv

Ex: → m.put (101, "durga"); → null
m.put (102, "Shiva"); → null
m.put (101, "Ravi"); → durga.

② void putAll (map m)
③ object get (object key) → return value associated with specified key.
④ object remove (object key) → removes entry associated with specified key
⑤ boolean containsKey (object key)
⑥ boolean containsValue (object value)
⑦ boolean isEmpty ();
⑧ int size ();

⑨ void clear ();

① Set keyset ()
② Collection values () } Collection views of map.
③ Set entryset ()

* Entry :→ (I)
- A map is a group of key value pair & each key value pair is called an entry. Hence map is considered as a collection of entry objects.
- without existing map object, there is an chance of existing entry object. Hence entry interface is defined inside map interface.

Ex: interface Entry
{
interface Entry
{
object getkey ()
object getvalue ()
object setvalue(object name)
}
}

Entry specific method & we can apply only on entry obj.

Scanned by CamScanner

① **HashMap** :

- The underlying data structure is hash table.
- Insertion order is not preserved & it is based on hashcode of keys.
- Duplicate keys are not allowed but values can be duplicated.
- Heterogeneous objects are allowed for key & value.
- Null is allowed for key (only once)
- Null is allowed for value (any no. of times)
- Hashmap implements serizable & Clonable interface but not random access.
- Hashmap is best choice if our frequent opn is search operation.

Constructors :

① Hashmap m = new Hashmap();
- creates an empty hashmap object with default initial capacity 16 & default fill ratio 0.75.

② Hashmap m = new Hashmap(int initial capacity);
- creats an empty hashmap object with specified initial capacity & default fill ratio 0.75.

③ Hashmap m = new Hashmap(int initial capacity, float fillratio);

④ Hashmap m = new Hashmap(map m);

Ex:-
```
class HashmapDemo
{
public static void main(String[] args)
{
  Hashmap m = new Hashmap();
  m.put("chiranjeevi", 700);
  m.put("balajah", 800);
  m.put("venkatesh", 200);
  m.put("nagarjuna", 500);
  sysout(m); // {k=v, k=v, ...}
  ~~sysout(m);~~ //
  Sysout(m.put("chiranjeevi", 1000));
  Set s = m.keyset();
  sysout(s); // [k, k1, ...]
  Collection c = m.values();
  Sysout(c);
  Set s1 = m.entryset();
  sysout(s1); // [k=v1, k=v1, k=v1, ...]
  Iterator itr = s1.iterator();
  while (itr.hashNext())
  {
    Map.entry m1 = (Map.entry) itr.next();
    Sysout(m1.getkey() + "..." + m1.getvalue);

  if (m1.getkey().equals("nagarjun"))
    {
      m1.setvalue(10000);
    }
    syout(m);
}
```

Scanned by CamScanner

* Differences Between Hashmap & Hashtable

Hashmap	Hashtable
① Every method present in Hashmap is non Synchronised	① Every method present in Hashtable is Synchronised.
② At a time multiple threads allowed to operate on Hashmap obj & thread safe.	② At a time only one Thread is allowed to operate on
③ relatively performance is high bcs threads are not required to wait to operate on hashmap object.	③ relatively performance is low bcs threads are required to wait to operate on ht. obj.
④ null is allowed for both key & value	④ null is not allow for keys & values otherwise we get NPE
⑤ introduced in 1.2v & it is not legacy.	⑤ introduced in 1.0v & it is legacy.

* How to get synchronised version of Hashmap object :→

- By default Hashmap is non-synchronised but we can get synchronised version of Hashmap by using synchronised map method of collections class.

ex:- Hashmap m = new Hashmap();
map m1 = Collections.synchronised(m);

synchronised

non synchronised.

② Linked Hashmap (C)

- It is child class of Hashmap.
- It is exactly same as Hashmap (including methods & constructors) except the foll^n differences.

Hashmap	Linked Hashmap
① The underlying data structure is hash table	① The underlying data structure is Linked List
② Insertion order is not preserved & it is based on Hashcode of keys.	② Insertion order is preserved.
③ Introduced in 1.2v	③ Introduced in 1.4v.

Scanned by CamScanner

Ex:→ In above Hashmap program, if we replace hashmap with linked Hashmap then o/p is :→

{
Chiranjeevi = 700;
balajah = 800; venkatesh = 200,
nagarjuna = 1000}

i.e. insertion order is preserved.

Notes→ Linked Hashset & linked Hashmap are commonly used for developing Cache based applications.

* Difference between == operator & .equals method.

- In general == operator meant for reference comparision (address comparison) whereas .equals method meant for content comparison.

$I_1 \rightarrow 10$
$I_2 \rightarrow 10$

Ex:

```
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);
S.o.pln(I1 == I2); // false
S.o.pln(I1.equals(I2)); // true.
```

③ Identity Hashmap (c)

- It is exactly same as Hashmap (including methods & constructors except the foll^n diff^n)

- In the case of normal Hashmap JVM will use .equals method to identify duplicates keys, which is meant for content comparison, But in the case of IdentityHashmap, JVM will use == operator to identify duplicate keys which is meant for reference comparison (address comparison).

$I_1 - 10$
$I_2 - 10$

Ex:→ Hashmap m = new Hashmap();
Integer I_1 = new Integer(10);
Integer I_2 = new Integer(10);
m.put(I_1, "pawan");
S.o.pln(m); // {10 = kalyan}

{ I_1 & I_2 are duplicate keys bcs I_1.equals(I_2) returns true

- If we replace Hashmap with Identify Hashmap then I_1 & I_2 are not duplicate keys bcs I_1 == I_2 returns false.

- In this case o/p is
{10 = pawan, 10 = kalyan}

Scanned by CamScanner

④ weakHashmap (c)

- It is exactly same as hashmap (except the following differences)
- In the case of hashmap even though object doesn't have any reference it is not eligible for GC, if it is associated with Hashmap i.e. Hashmap dominates GC
- But in case of weakHashMap, if object does't contain any references it is eligible for GC, eventhough object is associated with weapmap i.e. GC dominates weakHashmap.

Ex:

```
class weakHashmapDemo
{
public static void main(String[] args)
                        throws exception
{
Hashmap m = new Temp ();
m.put (t, "durga");
sysout (m);
t = null;
system.gc();
Thread.sleep (5000);
system(m);
}
}
```

```
class Temp
{
public String testing()
{
return "temp";
}
public void Finalize()
{
system.println ("Finalize method
                is called");
}
}
```

- In the above example temp object not eligible for GC, bcz it is associated with Hashmap.
- In this case output is {temp = durga} {temp = durga}
- In the above program, if we replace Hashmap with the weakHashMap then temp object eligible for GC In this case o/p is

```
{temp = durga}
Finalize method is called
{}
```

Scanned by CamScanner

⑤ SortedMap (I)

- It is the Child Interface of Map.
- If we want to represent a group of objects as a group key value pair according to some sorting order of keys then we should go for Sorted Map.
- Sorting is based on key but not based on value.

* SortedMap Defines the foll^n specific Methods :→

1) object FirstKey();
2) object lastKey();
3) SortedMap headMap (object key)
4) SortedMap tailMap (object key)
5) SortedMap subMap(object key1, object key2)
6) Comparator comparator()

ex:

① FirstKey() → 101
② LastKey() → 136
③ headMap (107) → {101 = A, 103 = B, 104 = C}
④ tailMap (107) → {107 = D, 125 = E, 136 = F}
⑤ subMap (103, 125) → {103 = B, 104 = C, 107 = D}
⑥ comparator() = null

101 → A
103 → B
104 → C
107 → D
125 → E
136 → F

⑥ TreeMap (C)

- The underlying datastructure is RED - BLAK tree.
- Insertion order is not preserved & it is based on some sorting order of keys.
- Duplicate keys are not allowed but values can be duplicated.
- if we are depending on default natural sorting order than keys should be homogenious & comparable otherwise we will get runtime exception saying CCE.
- if we are defining our own sorting by Comparator then keys need not be homogeneous & comparable we can take heterogeneous, non-comparable object also.
- whether we are depending on default natural sorting order or customised sorting order there are no restriction for values. we can take heterogenious non-comparable objects also.

Scanned by CamScanner

* null - acceptance :→

① For non-empty treemap, if we are trying to insert an entry with null key then we will get runtime exception saying NPE.

2) For empty-treemap as the first entry with the null key is allowed, but after inserting than entry if we are trying to insert any other entry then we will get RE saying NPE.

Note :→

- The above null-acceptance rule applicable until 1.6v only from 1.7v onwards null is not allowed for key.
- But for values, we can use null any no. of times, there is no restrication, whether it is 1.6v or 1.7v.

* Constructors :→

① Treemap t = new Treemap();
- for default natural sorting order.

② Treemap t = new Treemap(comparator());
for customized sorting order.

③ Treemap t = new Treemap(sortedMap m);

④ Treemap t = new Treemap(map m);

ex:→
```
Class TreemapDemo
        //Demo program for default sorting
{
  public static void main(String[] args)
  {
    Treemap m = new Treemap();
    m.put(100, "ZZZ");
    m.put(103, "YYY");
    m.put(101, "XXX");
    m.put(104, 106);
  // m.put("FFFF", "XXX"); // CCE
  // m.put(null, "XXX"); // NPE
    Sysout(m);
  // {100=222, 101=XXX, 103=XXX, 104=106}
  }
}
```

ex:→
```
Class TreemapDemo
        // Demo Prog. for Customized sorting
{ p.s.v.m(String[] args)
  {
    Treemap t = new Treemap(new
                    MyComparator());
    t.put("XXX", 10);
    t.put("AAA", 20);
    t.put("ZZZ", 30);
    t.put("ZZZ", 40);
```

Scanned by CamScanner

```
System.out.print(t);
// {zzz=30, xxx=10, LLL=40, AAA=20}
}}
Class MyComparator implements Comparator
{
public int compare(object obj1, object obj2)
{
    String s1 = obj1.toString();
    String s2 = obj2.toString();

    return s2.compareTo(s1)
}
}
```

⑦ Hashtable (c)

- The underlying DS for Hashtable is hashtable.
- Insertion order is not preserved & it is based on hashcode of key's
- Duplicate keys are not allowed & values can be duplicated.
- Heterogenous objects are allowed for both keys & values.
- Null is not allowed for both key & value otherwise we will get runtime exception saying NPE.
- It implements Serilizable & Clonable interfaces but not random access.
- Every method present in hashtable is synchronized & Hence hashtable object is thread safe.
- Hashtable is the best choice if our frequent opn is search opn.

* Constructors :→

① ~~Hashset~~
Hashtable t = new Hashtable();
- Creates an empty hashtable object with default initial capacity 11 & default fill ratio 0.75

② Hashtable t = new Hashtable(int initial capacity);

③ Hashtable t = new Hashtable(int initial capacity, float fillratio);

④ Hashtable t = new Hashtable(map m);

ex:→
```
Class HashtableDemo
{
    public static void main(String[] args)
    {
        Hashtable h = new Hashtable();
        h.put(new Temp(5), "A");
        h.put(new Temp(2), "B");
        h.put(new Temp(6), "C");
        h.put(new Temp(15), "D");
```

Scanned by CamScanner

```
h.put(new Temp(23), "E");          23%11=1
h.put(new Temp(16), "F");          16%11=5
//h.put("durga", null); // NPE

sysout.println(h);
}
}

Class Temp
{
  int i;
  Temp(int i)
  {
    this.i=i;
  }

  public int hashcode()
  {
    return i;
  }
  public String tostring()
  {
    return i+"";
  }
}
```

10	
9	
8	
7	
6	6=C
5	5=A, 16=F
4	15=D
3	
2	2=B
1	23=E
0	

O/P :- { 6=C, 16=F, 5=A, 15=D, 2=B, 23=E }

* if we Change Hashcode method of temp class as :

```
public int hashcode()
{
  return i%9;
}
```

10	
9	
8	
7	16=F
6	6=C, 15=D
5	5=A, 23=E
4	
3	
2	2=B
1	
0	

O/P :- { 16=F, 15=D, 6=C, 23=E, 5=A, 2=B }

* if we configure initial capacity as 25 that is :

```
Hashtable h = new hashtable(25);
```

O/P = { 23=E, 16=F, 15=D, 6=C, 5=A, 2=B }

24	
23	
⋮	
16	16=F
15	15=D
⋮	
6	6=C
5	5=A
4	
3	
2	2=B
0	

Scanned by CamScanner

(8) properties (c) :→

- In our program if anything which Changes frequently (like username, password, mail id, mobile no etc) are not recommend to hard code in java program, bcz if there is any change to reflect that change recompilation, rebuild & redeploy appn is required even sumtimes server restart also required which creates a big business impact to client.

- we can overcome this problem, by using properties file. such type of variable things we have to configure in the properties file from that properties files we have to read into java program & we can use those properties.

- The main advantage of this approach is if there is change in prop. files to reflect that change just redeployment is enough which won't create any business impact to the client.

- we can use java properties object to hold properties which are coming from property file.

- In normal map (like Hashmap, Hashtable, tree map) key & value can be any type but in the case of properties key & value should be string type.

* Constructor :→

properties p = new properties();

* Methods :→

(1) String setproperty (String pname, String pvalue) → to set a new property

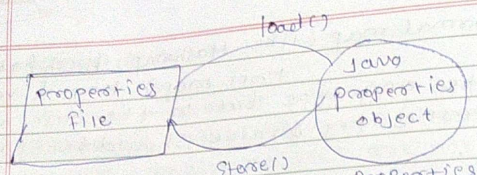
(2) String getproperty (String pname);
- to get value associated with specified property.

(3) Enumeration propertyNames()
- returns all propertyNames present in properties object.

(4) void load (Input stream is)
- to load properties from prop file into java prop object.

(5) void store (outputStream os, String comment)
- to store properties from java properties object into properties file.

Scanned by CamScanner



abc.properties

Properties p = new properties();

* ex :-

```
class propertiesDemo
{
  public static void main (String[] args)
        throws exception
  {
    properties p = new properties();
  FileInputStream fis = new FIS
                ("abc.properties");
    p.load (fis);
  System.Out.Print (p);
                // {PN = pv, ...}
  String s = p.getproperty ("venki");
  Sysout (s);   // gggg

  p.setproperty ("nag", "8888");
  FileOutputStream Fos = new Fos ("abc.
                properties");
  p.Store (Fos, "updated by durga for
        SCJP class");
  }
}
```

abc.properties
user = scott
pwd = tiger
venki = gggg

ex:

```
class propertiesDemo02
{
public static void main (String[] args)
{
  properties p = new properties();
FileInputStream fis = new FIS ("db.
                properties");
  p.load (fis);
String url = p.getproperty ("url");
String user = p.getproperty ("user");
String pwd = p.getproperty ("pwd");

Connection con = Driver.manage.getconne-
                ction (url1, user1, pwd);
}
}
```

Scanned by CamScanner

```
{
    String S1 = (String) obj1;
    String S2 = obj2.toString();
    return S2.compareTo(S1);
}
}.
```

* **1.6v Enhancement in Collection Framework**

1. Navigableset (I)
2. NavigableMap (I)

① **Navigableset (I) :→**

- it is the child interface of sortedset & it is defines several method for navigation purposes.

```
Collection (I)
    |
  Set (I)
    |
Sortedset (I)
    |
Navigableset (I)
    |
  Toset ()
```

Scanned by CamScanner

* Navigableset defines the following methods:

① floor(e)
- it returns highest element which is <= e

② lower(e)
- it returns highest element which is < e

③ ceiling(e)
- it returns lowest element which is >= e.

④ higher(e)
- it returns lowest element is > e

⑤ pollfirst()
- remove & return first element

⑥ poll Last()
- remove & return last element

⑦ descendingset
- it returns Navigableset in reverse order.

Ex:

```
Class NavigableSetDemo
{
public static void main(String[] args)
{
Treeset<Integer> t = new Treeset
                    <Integer>();
t.add(1000);
t.add(2000);
t.add(3000);
t.add(4000);
t.add(5000);

system.out.print(t); //[1000,2000,3000,4000,5000]
sysout(t.ceiling(2000)); // 2000
sysout(t.higher(2000)); //3000
sysout(t.floor(3000)); // 3000
sysout(t.lower(3000)); // 2000
sysout(t.pollfirst()); //1000
sysout(t.descendingset());
                    //[4000,3000,2000]
sysout(t); [2000,3000,4000]
}}
```

② NavigableMap (I)

- it is child interface of Sorted Map. It defines several methods for navigation purpose.

Scanned by CamScanner