Collections Revision

Difference b/w Collection(I) and Collections(C)?

Collection => It is an interface which should be used when we want to represent a group of individual object then we need to go for collection.

Collections => It is a utility class which defines in java.util which defines utility methods for Collection Objects**.(all methods are utility methods hence called as utility clas**s)

Collection is not prefered because

1. List l=new ArrayList(); **// default: 10 locations**

   if 11th element has to added, then

   a. create a list with 11 locations

   b. copy all the elements from the previous collection

   c. copy the new reference into reference variable

   d. call garbage collector and clean the old memory.

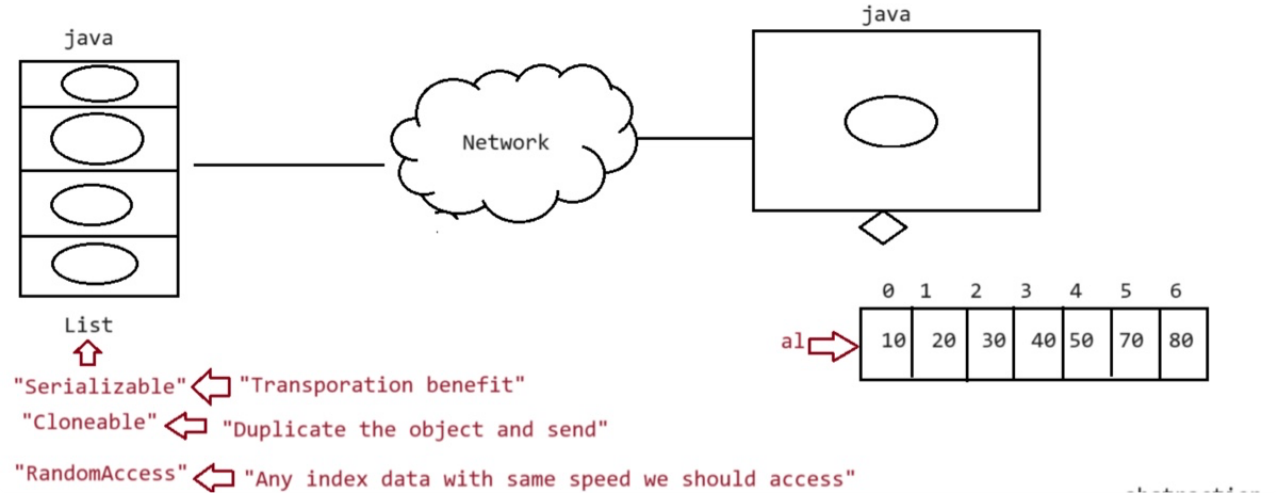
Note: Whenever we print any reference it internally calls toString() method. toString() of all Collection classes is implemented in such a way that it prints the Object in the following order.

o/p => [,,,]

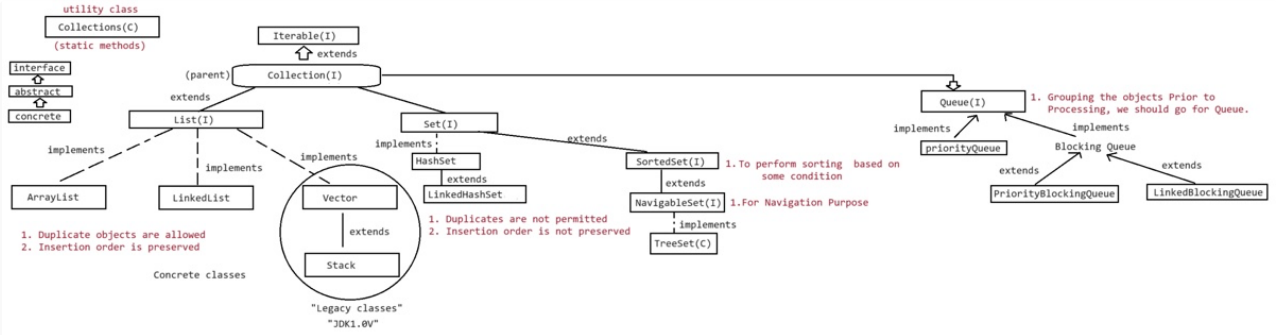
toString() of all Map Object is implemented in such a way that it prints the Object in the following order.

o/p => {k1=v1,k2=v2,k3=v3,....}

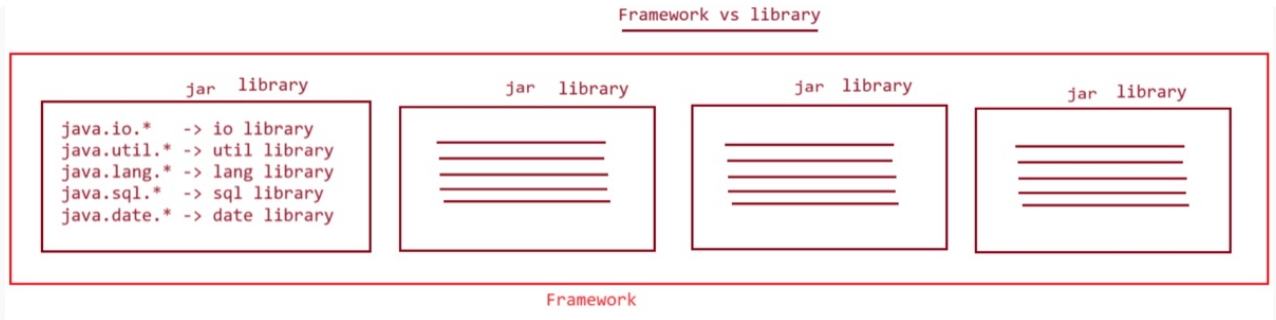
1. Every Collection class **by default implements Serializable.**
2. Every Collection class **by default implements Cloneable**



If no RandomAccess interface than it will iterate the first three items than it will return the value of 4th. So if a class implements RandomAccess interface than to access any index the time will be same



Framework vs Library



1)- List(I)

1. It is the child interface of Collection
2. To represent the group of collection objects where
   a. duplicates are allowed(meaning it is stored in index)
   b. insertion order is preserved.(meaning it is stored via index)
3. In list index plays a very important role

Vector and Stack are a part of jdk1.0 version so they are called as **"legacy classes"(old classes)**.

Method assoicated with List(I)

a. void add(int index,Object obj)

b. void addAll(int index,Collection c)

c. Object remove(int index)

d. Object get(int index)

e. Object set(int index,Object o)

f. int indexOf(Object obj)

g. int lastIndexOf(Object obj)

i. ListIterator listIterator()

a)- ArrayList(C)

1. DataStructure: GrowableArray /Sizeable Array
2. Duplicates are allowed through index

3. insertion order is preserved through index
4. Heterogenous objects are allowed.
5. null insertion is also possible.

Constructors

a. ArrayList al=new ArrayList()

Creates an empty ArrayList **with the capacity to 10.**

a. if the capacity is filled with 10, then what is the new capacity?

newcapacity= (currentcapacity * 3/2)+1

so new capacity is =16,25,38,.....

b. **if we create an ArrayList in the above mentioned order then it would result in performance issue.**

c. **To resolve this problem create an ArrayList using 2nd way apporach**.

b. ArrayList al=new ArrayList(int initalCapacity)

c. ArrayList l=new ArrayList(Collection c)

Methods →

l.add("A");

l.add(null);

l.add(10);

l.size(); **//3[A, 10, null]**

l.remove(2); **//[A,10]**

l.add(2,"sachin"); **//[A,10,sachin]**

ArrayList vs Vector

These 2 classes along with Serializable,Cloneable,it also implements RandomAccess. Any random elements present in ArrayList and Vector can be accesed through same speed, becoz it is accessed using "RandomAccess".

ArrayList and Vector is best suited when our frequent operation is read.

RandomAccess is a marker interface which is a part of java.util package,**where the required ability is provided automatically by the jvm.**

eg#1.

ArrayList l1= new ArrayList();

LinkedList l2=new LinkedList();

System.out.println(l1 instanceof Serializable);//true

System.out.println(l1 instanceof Cloneable);//true

System.out.println(l2 instanceof Serializable);//true

System.out.println(l2 instanceof Cloneable);//true

System.out.println(l1 instanceof RandomAccess);//true

System.out.println(l2 instanceof RandomAccess);//false

When to use ArrayList and when not to use?

ArrayList => it is best suited if our frequent operation is "retrieval operation",because it implements RandomAccess interface.

ArrayList => it is the worst choice if our frequent operation is "insert/deletion" in the middle because it should perform so many shift operations. To resolve this problem we should use "LinkedList".

Differences b/w ArrayList and Vector?

ArrayList => Most of the methods are not synchronized, not thread safe, performance is high becoz threads are not allowed to wait, not a legacy class.

Vector =>Most of the methods are synchronized, thread safe, performance is relatively low becoz thread are required to wait, It is a legacy class.

How to use ArrayList, but thread safety is required how would u get or how to get synchronized version of ArrayList?

ArrayList l=new ArrayList();**// now 'l' is nonsynchronized**

ArrayList l1=Collections.**synchronizedList**(l);**// now 'l1' is synchronized**

Note:: These methods are a part of Collections class(utility class)

public static List synchronizedList(List l);

public static Map synchronizedMap(Map m);

public static Set synchronizedSet(Set s);

b)- LinkedList

=> Memory management is done effectively if we work with LinkedList.

=> memory is not given in continous fashion.

a. DataStructure is **:: doubly linked list**

b. heterogenous objects are allowed

c. null insertion is possible

d. duplicates are allowed

e. linkedlist implements Serializable and Cloneable interface **but not RandomAccess.**

Usage

1. **If our frequent operation is insertion/deletion in the middle then we need to opt for "LinkedList".**
2. LinkedList is the worst choice if our frequent operation is retreival operation.**(that's why it does not implement RandomAccess interface)**

Constructors

a. LinkedList l=new LinkedList();

It creates a empty LinkedList object.

b. LinkedList l=new LinkedList(Collection c);

To convert any Collection object to LinkedList.

Methods associated with LinkedList

Normally we use LinkedList to implement stack and queue, to provide the support we use

stack -> push(),pop(),display()

queue -> insert(),delete(),display()

1. public E getFirst();
2. public E getLast();
3. public E removeFirst();
4. public E removeLast();
5. public void addFirst(E);
6. public void addLast(E);

```
LinkedList l=new LinkedList();
l.add(a);
l.add(10);
l.add(z);
l.add(2,'a');
l.remove(0); //[10,a]
ll.set(0,"naveen"); //[10,a,naveen]
ll.addFirst("dhoni"); //[10,a,dhoni]
ll.addLast("kohli"); //[10,a,kohli]
```

c)- Vector:

=> The Underlying Data Structure is **Resizable Array OR Growable Array.**

=> Insertion Order is Preserved.

=> Duplicate Objects are allowed.

=> Heterogeneous Objects are allowed.

=> null Insertion is Possible.

=> Implements Serializable, Cloneable and **RandomAccess** interfaces.

=> Every Method Present Inside Vector is Synchronized and Hence Vector Object is Thread Safe.

=> Vector is the Best Choice if Our Frequent Operation is Retrieval.

=> Worst Choice if Our Frequent Operation is Insertion OR Deletion in the Middle.

Constructors

1. Vector v = new Vector();

   => Creates an Empty Vector Object with **Default Initial Capacity 10.**

   Once Vector Reaches its Max Capacity **then a New Vector Object will be Created** with

   New Capacity = Current Capacity * 2

2. Vector v = new Vector(intinitialCapacity);

3. Vector v = new Vector(intinitialCapacity, intincrementalCapacity);
4. Vector v = new Vector(Collection c);

Methods:

1. **To Add Elements:**

   addElement(Object o) -> Vector
2. **To Remove Elements:**

   removeElement(Object o) -> Vector

   removeElementAt(int index) -> Vector

   removeAllElements() -> Vector
3. **To Retrive Elements:**

   Object elementAt(int index) -> Vector

   Object firstElement() -> Vector

   Object lastElement() -> Vector
4. **Some Other Methods:**

   int size()

   int capacity()

   Enumeration element()

What are the implemenatation classes of List(I)?

Ans. ArrayList :: It implements an interface called "Serializable, Cloneable,RandomAccess". Best suited if the operation is "retrieval Operation".

Constructors :: 3 constructors

LinkedList :: It implements an interface called "Serializable, Cloneable" Best suited if the Operation is "insertion and deletion".

Constructors :: 2 constructors

2 Legacy classes[JDK1.0V]

a. Vector :: It implements an interface called "Serializable, Cloneable,RandomAccess". Best suited if the operation is "retrieval Operation".

Constructors :: 4 constructors

Methods :: synchronized

b. Stack :: It is specially designed class for LIFO order.

Constructors :: 1 constructors

public class **java.util.Stack extends java.util.Vector** {

public java.util.Stack();

public E push(E);

public synchronized E pop();

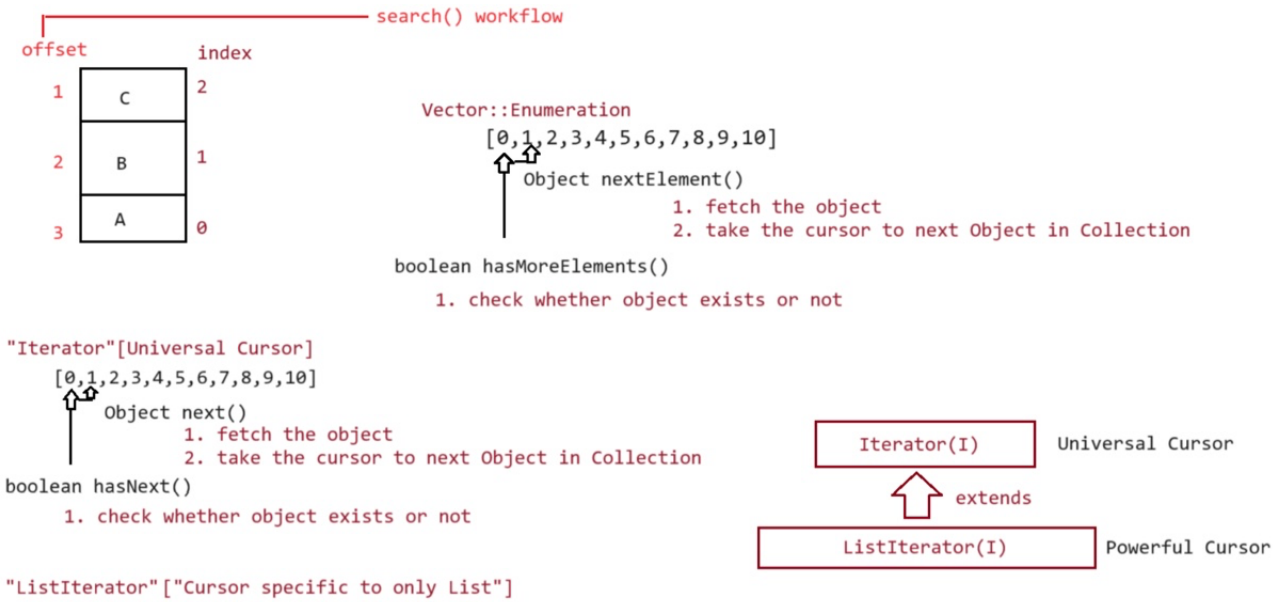
public synchronized E peek();

public boolean empty();

public synchronized int search(java.lang.Object);

```
}

Stack s =new Stack();
System.out.println(s.empty());  //true
s.push("A");
s.push("B");
s.push("C");
System.out.println(s);  //[A,B,C]
System.out.println("Pop element is :: "+s.pop()); // C
System.out.println(s.empty());
System.out.println(s.search("Z"));   //-1
System.out.println(s.search("A"));   //3
```

offset and index for stack + iterator, enumeration and ListIterator →



The 3 Cursors of Java:

=> If we want to get Objects One by One from the Collection then we should go for Cursors.

=> There are 3 Types of Cursors Available in Java.

1. Enumeration
2. Iterator
3. ListIterator

1. **Enumeration:**

We can Use Enumeration to get Objects One by One from the Collection. We can Create Enumeration Object by using elements().

public Enumeration elements();

Eg: Enumeration e = v.elements(); //v is Vector Object.

Methods:

1. public boolean hasMoreElements();
2. public Object nextElement();

```
Enumeration e = v.elements(); //v = vector
while (e.hasMoreElements())
{
    Integer obj =(Integer)(e.nextElement());
    if (obj%2==0)
        System.out.println(obj);//[0,2,4,6,8,10]
}
```

Limitations of Enumeration:

=> Enumeration Concept is **Applicable Only for Legacy Classes** and **it is Not a Universal Cursor.**

=> By using Enumeration we can Perform Read Operation **and we can't Perform Remove Operation**.To Overcome Above Limitations we should go for Iterator.

2. **Iterator:**

=> We can Use Iterator to get Objects One by One from Collection.

=> We can Apply Iterator Concept for any Collection Object. Hence **it is Universal Cursor**.

=> By using Iterator we can Able to Perform Both Read and Remove Operations.

=> We can Create Iterator Object by using iterator() of Collection Interface.

public Iterator iterator();

Methods:

1. public booleanhasNext()
2. public Object next()
3. public void remove() **//Extra method given by Iterator interface**

```
Iterator itr = al.iterator();
while (itr.hasNext())
{
    Integer i = (Integer)(itr.next());
    if (i%2==0)
    {
        System.out.println(i);//[0,2,4,6,8,10]
```

```
    }
    else
    {
        itr.remove();
    }
}
```

Limitations:

=> By using Enumeration and Iterator **we can Move Only towards Forward Direction and we can't Move Backward Direction**. That is these are Single Direction Cursors **but Not BiDirection.**

=> By using Iterator we can Perform Only Read and Remove Operations and **we can't Perform Addition of New Objects and Replacing Existing Objects.**

To Overcome these Limitations we should go for ListIterator.

Iterable

1. It is related to forEach loop
2. The target element in forEach loop should be Iterable.
3. Iterator present in java.lang package.
4. **contains only one method iterator().**
5. Introduced in 1.5 version.

Iterator

1. It is related to Collection.
2. We can use Iterator to get objects one by one from the collection.
3. Iterator present in java.util package.
4. **contains 3 methods hasNext(), next(), remove()**
5. Introduced in 1.2 version.

3. **ListIterator:**

=> ListIterator is the Child Interface of Iterator.

=> By using ListIterator we can Move Either to the Forward Direction OR to the Backward Direction. That is **it is a Bi-Directional Cursor.**

=> By using ListIterator we can Able to Perform Addition of New Objects andReplacing existing Objects. In Addition to Read and Remove Operations.

=> We can Create ListIterator Object by using listIterator().

```
public ListIteratorlistIterator();
```

Methods:

=> ListIteratoris the Child Interface of Iterator and Hence All Iterator Methods by Default Available to the ListIterator.

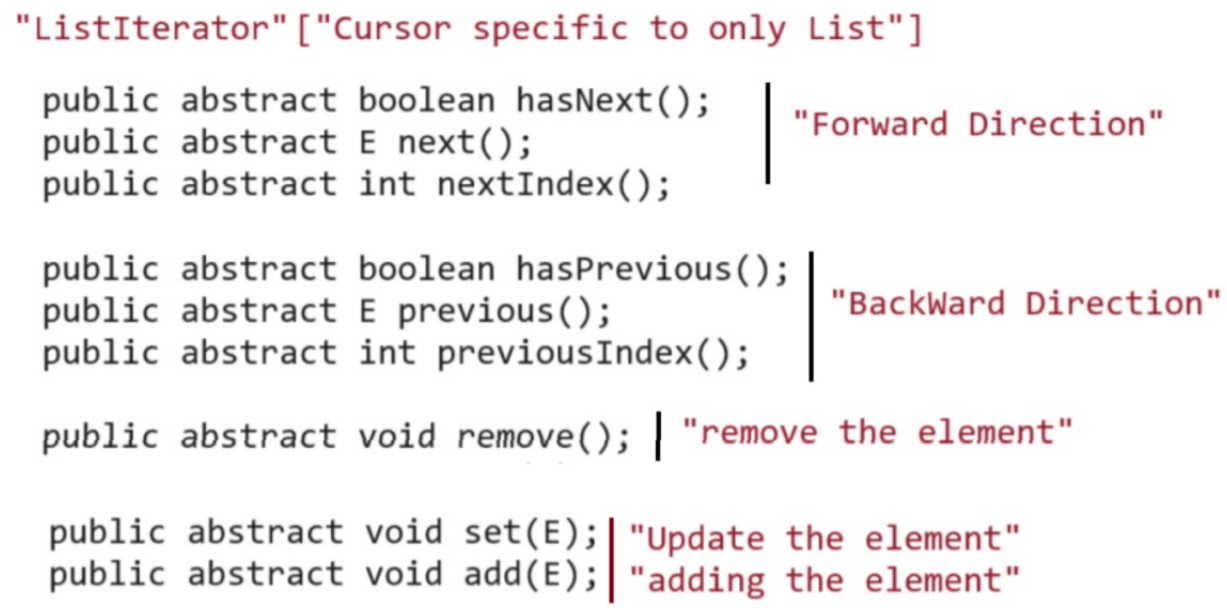
Iterator(I)

^

|

ListIterator(I)

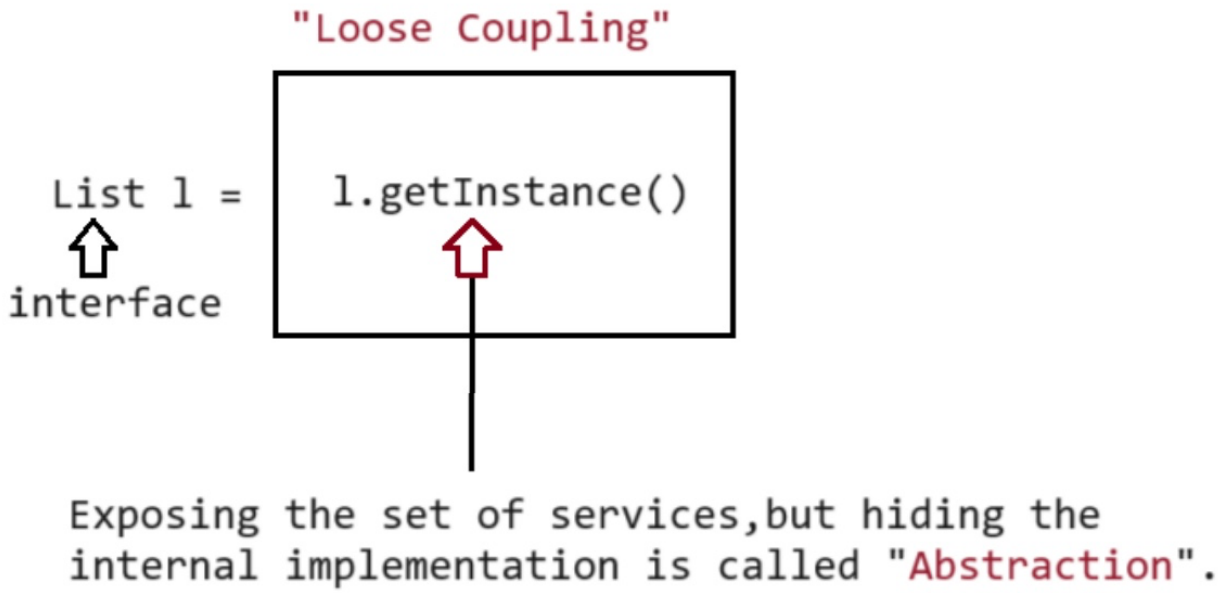
ListIteratorDefines the following 9 Methods.




```
ListIterator litr = ll.listIterator();
while(litr.hasNext())
{
    String name = (String)(litr.next());
    if (name.equals("dravid"))
    {
        litr.remove();
    }
    if (name.equals("sachin"))
    {
        litr.set("sachintendulkar");
    }
    if (name.equals("dhoni"))
    {
        litr.add("sakshi"); // will add next to dhoni
    }
}
```

Note: Why is Iterator required if we can use foreach?? → Ans - with Iterators we can perform updation tasks along with reading data, on the other hand with foreach we can only do reading operation.

Note: Will be easier while making projects(below diagram + example)




```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        Enumeration e = v.elements();
        System.out.println("Enumeration :: "+e.getClass().getName());
        Iterator itr = v.iterator();
        System.out.println("Iterator :: "+itr.getClass().getName());
        ListIterator litr = v.listIterator();
        System.out.println("ListIterator :: "+litr.getClass().getName());
    }
}
```

Enumeration :: java.util.Vector$1

Iterator :: java.util.Vector$Itr

ListIterator :: java.util.Vector$ListItr

Set:

=> It is the Child Interface of Collection.

=> If we want to Represent a Group of Individual Objects as a Single Entity **where Duplicates are Not allowed and Insertion Order is Not Preserved** then we should go for Set.

=> Set Interface doesn't contain any new Methods and Hence we have to Use Only Collection Interface Methods.

HashSet

1. Duplicates are not allowed,if we try to add it would not throw any error rather it would return false.
2. **Internal DataStructure: Hashtable**
3. null insertion is possible.
4. heterogenous data elements can be added.
5. If our frequent operation is search, then the best choice is HashSet.
6. It implements Serializable,Cloneable, **but not random access.**

Constructors

HashSet s=new HashSet(); Default initial capacity is 16

Default FillRation**/load factor is 0.75**

Note: In case of ArrayList, default capacity is 10, after filling the complete capacity then new ArrayList would be created. In case of HashSet, after filling 75% of the ratio only new HashSet will be created.

HashSet s=new HashSet(int intialiCapacity);**//specified capacity with default fill ration=0.75**

HashSet s=new HashSet(int initaliCapacity,float fillRatio)

HashSet s=new HashSet(Collection c);

LoadFactor(at what percentage the size should be increased → it is final so we cant change its value)

After loading how much ratio,a new object will be created is called as "LoadFactor".

eg#1.

```
HashSet hs = new HashSet();
        hs.add("A");
        hs.add("B");
        hs.add("C");
        hs.add("D");
        hs.add("A");
        hs.add(null);
        System.out.println(hs);//[null, A, B, C, D]
        System.out.println();
       //Underlying DataStructure :: Hashtable + LinkedList
        LinkedHashSet lhs = new LinkedHashSet();
        lhs.add("A");
        lhs.add("B");
        lhs.add("Z");
        lhs.add("C");
        lhs.add(10);
```

LinkedHashSet

It is the child class of "HashSet".

DataStructure: Hashtable + linkedlist

duplicates : not allowed

insertion order: preserved

null allowed : yes

All the constructors and methods which are a part of HashSet will be a part of "LinkedHashSet",but except "insertion order will be preserved".

Difference b/w HashSet and LinkedHashSet

HashSet =>ds: "HasTable", Duplicates are not allowed and insertion order is not preserved, 1.2V

LinkedHashSet => ds: "Hashtable + "linkedlist", Duplicates are not allowed,but insertion order is preserved, 1.4v

Note: insertion order is preserved, but duplicates are not allowed. Whenever we want to develp cache based application(web browsers),where duplicates are not allowed insertion order must be preserved then we go for "LinkedHashSet".

SortedSet(I)

It is the child interface of Set Group of individual objects, where duplicates are not allowed,**but the elements should be sorted in some order.**

eg: {3,2,1}

{1,2,3}

{3,1,2}

{2,1,3}

Some specific methods w.r.t SortedSet are

a. Object firstElement() => returns first element

b. Object lastElement() => returns last element

c. SortedSet headSet(Object obj)=> returns sortedset whose elements are < obj

d. SortedSet tailSet(Object obj)=> returns sortedset whose elements are >=obj

e. SortedSet subSet(Object obj1,Object obj2)=> returns subset whose elements are=obj1 and <obj2

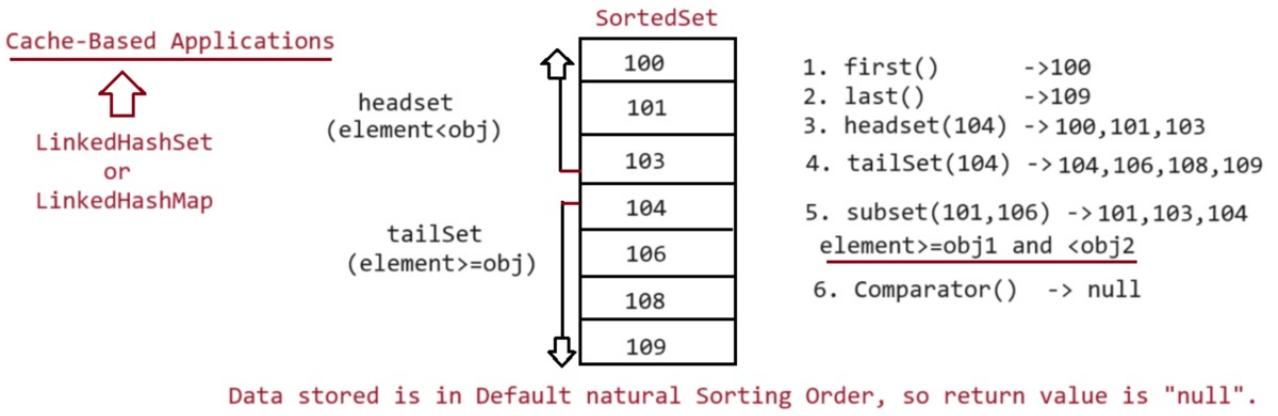
f. Compartor compartor() => returns Comparotor object that describes underlying sorting technique default natural sorting order means it returns null.

Note::

String Object => default natural sorting order is Alphabetical order[A to Z]

Number Object => default natural sorting order is Ascending order.[0 to 9]

SortedSet



Name the cursors available for iterating the Collection Objects

a. Enumeration(I) => Legacy Cursor.

b. Iterator(I) => Universal Cursor as it can be iterated on every collection Object.

c. ListIterator(I) => It can be used only on List(I) implementation class Object.

TreeSet

Underlying Datastructure: BalancedTree

duplicates : not allowed

insertion order : not preserved

heterogenous element: not possible,if we try to do it would result in "**ClassCastException**".

inserting null : NullPointerException.

Implements Serializable and Cloneable interface,but not RandomAccess.

All Objects will be inserted based on "some sorting order" or "customized sorting order".

Constructor

TreeSet t=new TreeSet();//All objects will be inserted based on some default natural sorting order.

TreeSet t=new TreeSet(Comparator);//All objects will be inserted based on some customized sorting order.

TreeSet t=new TreeSet(Collection c);

TreeSet t=new TreeSet(SortedSet);

Note::

Comparable => Default natural sorting order.

Comparator => Customized sorting order.

eg#2.

```
import java.util.TreeSet;

class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet t = new TreeSet();
        t.add(new StringBuffer("A"));
        t.add(new StringBuffer("Z"));
        t.add(new StringBuffer("L"));
        t.add(new StringBuffer("B"));
        System.out.println(t);
    }
}
```

Output:ClassCastException

Reason::

Default → Comparable(TreeSet t = new TreeSet();**[empty parenthesis]**)

TreeSet t=new TreeSet()

a. we inform jvm to use default natural sorting order

To sort the elements must be

a. Homogenous

b. Comparable(class should implement Comparable) otherwise it would result in "ClassCastException".

Note:: Object is said to be Comparable, iff the corresponding class implements "Comparable". **All Wrapper class and String class implements Comparable so we can compare the objects.**

But StringBuffer does not implements Comparable hence the exception and same code on higher versions of jdk(11+) will work.As now they have decided almost all the classes should implement Comparable.

Comparsion table of Set implemented Classes

HashSet => underlying data structure is HashTable duplicates not allowed

insertion order not preserved

Sorting order not preserved

duplicates not allowed

heterogenous elements allowed

null allowed

LinkedHashset => underlying data structure is linkedhashset and HashTable

duplicates not allowed

inserted order preserved

Sorting order not preserved

duplicates not allowed

heterogenous elements allowed

null allowed

TreeSet => underlying data structure is balanced Tree

duplicates not allowed

insertion order not preserved

Sorting order not preserved

duplicates not allowed

heterogenous elements not allowed by default

null not allowed.

Comparable(I)

=> It is a part of java.lang package

=> It contains only one method compareTo.

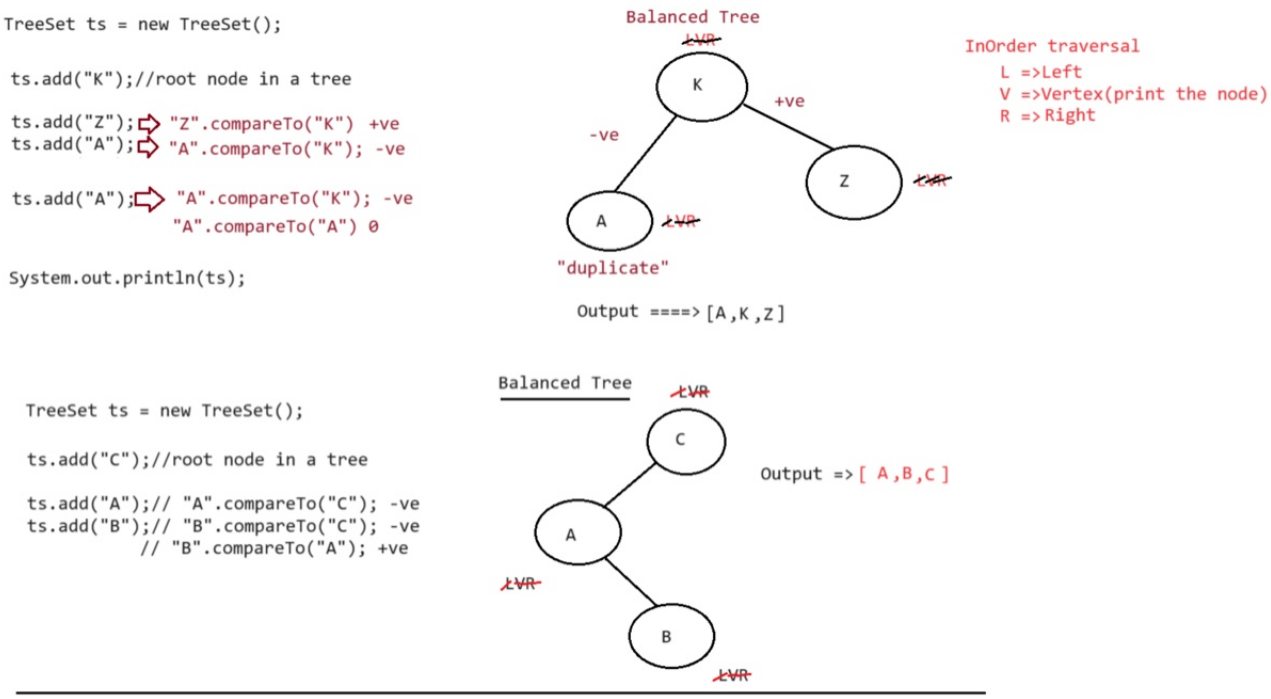
public int compareTo(Object o)

=> obj1.compareTo(obj2)

returns -ve iff obj1 has to come before obj2[B-Negative]

returns +ve iff obj1 has to come after obj2[A-Postivie]

returns 0 if both are equal[Both are equal]



Rule: obj1.compareTo(obj2)

obj1 => The object which needs to be inserted.

obj2 => The object which is already inserted.

Whenever we are depending on default natural sorting order,if we try to insert the elements then internally it calls compareTo() to IdentifySorting order.

Comparator

=> compare()

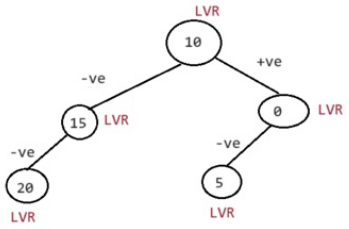
It is meant for customized sorting order.

Write a program to insert integer objects into the TreeSet where sorting order is descending order?

```
import java.util.TreeSet;
import java.util.Comparator;
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;
    }
```

```
}
public class TestApp{
    public static void main(String... args){
        TreeSet ts= new TreeSet(new MyComparator());
        ts.add(10);
        ts.add(0);
        ts.add(15);
        ts.add(5);
        ts.add(20);
        ts.add(20);
        System.out.println(ts);//[0,5,10,15,20]
    }
}
```

```
ts1.add(10);

ts1.add(0); ⇨ compare(0,10)

ts1.add(15); ⇨ compare(15,10)

ts1.add(5); ⇨ compare(5,10)
              compare(5,0)

ts1.add(20); ⇨ compare(20,10)
               compare(20,15)

ts1.add(20); ⇨ compare(20,10)
               compare(20,15)
               compare(20,20)
```



Output => [20,15,10,5,0]

```
if (i1<i2)
{
    //obj1 has to come before obj2 => Descending order
    return 1;
}
else if(i1>i2)
{
    //obj1 has to come after obj2 => Descending order
    return -1;
}
else
{
    //obj1==obj2
    return 0;
}
```

Various Possible combination implementation of compare()

```
class MyComparator implements Comparator{
    public int compare(Object obj1,Object obj2){
        Integer i1=(Integer)obj1;
        Integer i2=(Integer)obj2;
        return i1.compareTo(i2);  //ascending order
        return -i1.compareTo(i2);  //descending order
        return i2.compareTo(i1);  //descending order
        return -i2.compareTo(i2);   //ascending order
        return +1;   //insertion order is preserved
        return -1;   //reverse of insertion order
        return 0;   //only first elements is added,remaining all duplicates
    }
}
```

When obj1 is string buffer

String s1=obj1.toString(); **//we cant do (String)obj1 as there is no relationship between string and stringbuffer**

Note:

Comparable :: By default the object we add into treeSet, the corresponding class should implement "Comparable" interface and **the object should be homogenous** Otherwise it would result in "ClassCastException".

Comparator :: This interface is meant for Custom sorting, so the objects added to TreeSet **need not be "Homogenous" and need not implement "Comparable"**.

We can add Non-Homogenous and Non-Comparable objects into TreeSet.

Scenario

When to go for Comparable and Comparator?

1st category

Predefined Comparable classes like String and Wrapper class

=> Default natural sorting order is already available

=> If not satisfied, then we need to go for Comparator

2nd Category

Predefined NonComparable classes like StringBuffer

=> Default natural sorting order not available so go for Comparator only always

3rd Category

Our Own classes like Employee,Student,Customer

=>Person who is writing this classes are responsible for implementing comparable interface to promote Natural sorting order.

=>Person who is using this class,can define his own natural sorting order by implementing Comparator interface.

very ver imp → treeset doesn't implement comparable, the objects that we pass to treeset implements comparable

Comparable and Comparator

Comparable => Meant for default natural sorting order
Comparator => Meant for customized sorting order
Comparable => part of java.lang package
Comparator => part of java.util package
Comparable => only one method compareTo()
Comparator => 2 methods compare(),equals()
Comparable => It is implemented by Wrapper class and String class
Comparator => It is implemented by Collator and RuleBaseCollator(GUI based API)

Map

=> It is not a child interface of Collection.
=> If we want to represent group of Objects as key-value pair then we need to go for Map.
=> Both keys and values are Objects only
=> Duplicate keys are not allowed but values are allowed.
=> Key-value pair is called as "**Entry**".

Map interface

1. It contains 12 methods which is common for all the implementation Map Objects
   a. Object put(Object key,Object value)
   b. void putAll(Map m)
   c. Object get(Object key)
   d. Object remove(Object key)
   e. boolean containsKey(Object key)
   f. boolean containsValue(Object value)
   g. boolean isEmpty()
   h. int size()
   i. void clear()

views of a Map

   j. Set keySet()
   k. Collection values()
   l. Set entrySet()

Entry(I)

1. Each key-value pair is called Entry.
2. Without existence of Map,there can't be existence of Entry Object.
3. **Interface Entry is defined inside Map interface.**

```
interface Map{
    interface Entry{
        Object getKey();
        Object getValue();
        Object setValue(Object newValue);
    }
}
```

HashMap

Underlying DataStructure: Hashtable

insertion order : not preserved

duplicate keys : not allowed

duplicate values : allowed

Heterogenous objects : allowed

null insertion : **for keys allowed only once,but for values can be any no**.

implementation interface: Serializable,Cloneable.

Difference b/w HashMap and Hashtable

HashMap => All the methods are not synchronized.

Hashtable => All the methods are synchronzied.

HashMap => At a time multiple threads can operate on a Object, so it is not ThreadSafe.

Hashtable => At a time only one Thread can operate on a Object, so it is ThreadSafe.

HashMap => Pefromance is high.

Hashtable => performance is low.

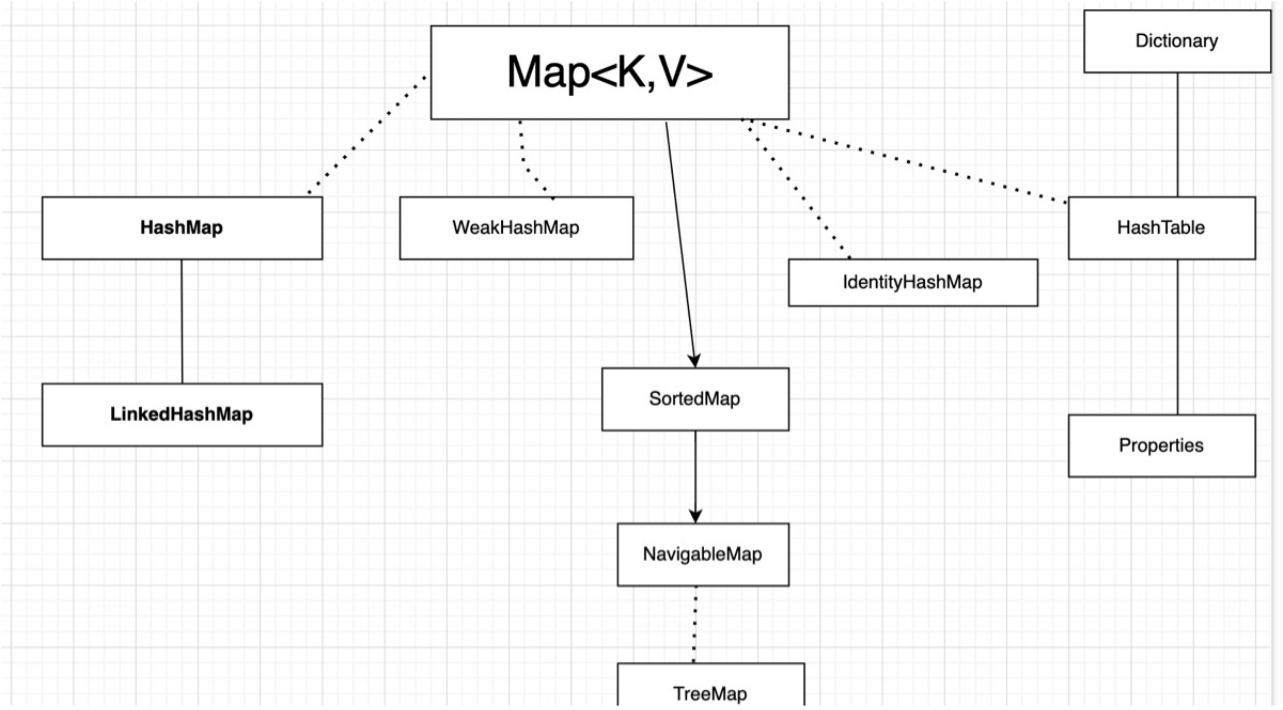
HashMap => null is allowed for both keys and values.

Hashtable => null is not allowed for both keys and values,it would result in NullPointerException.

HashMap => Introduced in 1.2v

Hashtable => Introduced in 1.0v

Note: By default HashMap is nonSynchronized,to get the synchronized version of HashMap we need to use synchronizedMap() of Collection class.



Dotted lines → implements

Straight lines → extends

Dictionary is Abstract class

Dictionary,Hashtable and properties are legacy classes

Constructors

1. HashMap hm=new HashMap()
   //default capacity => 16, **loadfactor => 0.75(at 75% usage of memory cpacity(or size) will be increased)**
2. Hashmap hm=new HashMap(int capacity);
3. HashMap hm=new HashMap(int capacity,flot fillration);
4. HashMap hm=new HashMap(Map m);

eg#1.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        //Underlying DS :: Hashtable
        //Insertion Order :: not preserved[uses Hashing technique]
        //Duplicates :: Key no, but value can be duplicated
        // null :: Key allowed only once,but value any no of times
        System.out.println("HASHMAP*****");
        HashMap hm = new HashMap();
        hm.put(10,"sachin");
        hm.put(7,"dhoni");
        hm.put(18,"virat");
        hm.put(1,"rahul");
        hm.put(45,"rohith");
        System.out.println(hm);  //{1=rahul, 18=virat, 7=dhoni, 10=sachin, 45=rohith}
        Set keys = hm.keySet();
        System.out.println(keys);  //[1, 18, 7, 10, 45]
        Collection c= hm.values();
        System.out.println(c);   //[rahul, virat, dhoni, sachin, rohith]
        Set set = hm.entrySet();
        //[1=rahul, 18=virat, 7=dhoni, 10=sachin, 45=rohith]
        System.out.println(set);

        System.out.println();

        //Getting the iterator
```

```
        Iterator itr = set.iterator();
        while (itr.hasNext())
        {
            Map.Entry m = (Map.Entry)itr.next();
            System.out.println("key is :: "+m.getKey());
            System.out.println("Value is :: "+m.getValue());

            System.out.println();

            //Method used for Updation
            if ((Integer)m.getKey() == 10)
            {
                m.setValue("sachinrameshtendulkar");
            }
        }
        System.out.println(hm);
        hm.put(null,null);
        System.out.println(hm);

        System.out.println();

        System.out.println("HASHTABLE*****");
        Hashtable ht = new Hashtable();
        ht.put(null,null);  //CE(NullPointerException)
        System.out.println(ht);
    }
}
```

Points →

1)- We **can't directly use iterator with maps** as it is not a part of collection. So how to use iterator??

2)- **Sol →**

```
Set set = hm.entrySet();
//[1=rahul, 18=virat, 7=dhoni, 10=sachin, 45=rohith]
System.out.println(set);
```

3)- Now since we have got the data in Set we can use iterator. Remember that //[1=rahul, 18=virat, 7=dhoni, 10=sachin, 45=rohith] → this is not a key value pair map. 1=rahul → it is just an object(an entry).

4)- **1=rahul → its an entry so while its access we will store it in Map.Entry type through which we can both value and key**

LinkedHashMap

=> It is the child class of HashMap.

=> It is same as HashMap,but with the following difference

HashMap => underlying datastructure is hashtable.

LinkedHashMap => underlying datastructure is LinkedList + hashtable.

HashMap => insertion order not preserved.

LinkedHashMap => insertion order preserved.

HashMap => introduced in 1.2v

LinkedHashMap => introduced in 1.4v

In the above program, if we replace HashMap with LinkedHashMap then the output would be**(where insertion order is preserved.)**

Note: for developing cache based applications, we use HashMap and LinkedHashMap where duplicates are not allowed,but insertion order preserved.

IdentityHashMap

It is same as HashMap, with the following differences.

a. In case of HashMap,jvm will use **equals()** to check whether the keys are duplicated or not.

equals() => meant for ContentComparison.

b. In case of IdentityHashMap,jvm wil use **==** operator to identify whether the keysare duplicated. or not.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        Integer i1 =new Integer(10);
        Integer i2 =new Integer(10);
        System.out.println("HashMap**");
        HashMap hm = new HashMap();
        hm.put(i1,"sachin");
        hm.put(i2,"messi");
        //{10=messi}
        System.out.println(hm);  //JVM  i1.equals(i2)  true

        System.out.println();

        System.out.println("IdentityHashMap**");
        IdentityHashMap ihm =new IdentityHashMap();
        ihm.put(i1,"sachin");
```

```
        ihm.put(i2,"messi");
        //{10=sachin,10=messi}
        System.out.println(ihm);  //JVM  i1 == i2  false
    }
}
```

IdentityHashMap does not violates the rule of maps, its just how jvm internally sees object in two types of maps.

WeakHashMap

It is exactly same as HashMap, with the following differences.

1. HashMap will always dominate Garbage Collector, that is if the Object is a part of HashMap and if the Object is Garbage Object, still Garbage Collector won't remove that Object from heap since it is a part of HashMap. HashMap dominates GarbageCollector.
2. Garbage Collector will dominate WeakHashMap, that is if the Object is part of WeakHashMap and if that Object is Garbage Object, then immediately Garbage Collector will remove that Object from heap even though it is a part of WeakHashMap, so we say Garbage Collector dominates "WeakHashMap".

eg#1.

```
import java.util.*;
public class Test
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("WeakHashMap*");
        WeakHashMap whm = new WeakHashMap();
        Temp t = new Temp();
        whm.put(t,"sachin");
        //Making object Garbage
        t =null; // there is no garuntee that gc will be called here immediately
        System.out.println("Signal to Garbage Collector...");
        //Hence Explicitly Calling Garbage Collector
        System.gc(); // before cleaning the object gc will call finalize method
        Thread.sleep(5000);
        System.out.println(whm);//{}
    }
}
class Temp
{
    @Override
    public String toString(){
```

```
        return "Temp";
    }
    @Override
    public void finalize(){
        System.out.println("Garbage Collector Cleaning the Object...");
    }
}
```

Output

WeakHashMap*

Signal to Garbage Collector...

Garbage Collector Cleaning the Object...

{}

In the above example if we change WeakHashMap to HashMap than instead of {} it will print {Temp=sachin}

SortedMap

1. It is the child interfae of Map
2. If we want Entry object to be sorted and stored inside the map,we need to used "SortedMap".

SortedMap defines few specific method like

a. Object firstKey()
b. Object lastKey()
c. SortedMap headMap(Object key)
d. SortedMap tailMap(Object key)
e. SortedMap subMap(Object obj1,Object obj2)
f. Compartor compartor()

TreeMap

1. Underlying datastructure is "redblacktree".
2. Duplicate keys are not allowed,where as values are allowed.
3. Insertion order is not preserved and it is based on some sorting order.
4. **If we are depending on natural sorting order,then those keys should be homogenous and it should be Comparable otherwise ClassCastException.**
5. If we are working on customisation through Comparator,then those keys can be heterogenous and it can be NonComparable.
6. No restrictions on values, it can be heterogenous or NonComparable also.
7. **If we try to add null Entry into TreeMap, it would result in "NullPointerException".**

Constructors of TreeMap

TreeMap t=new TreeMap();

TreeMap t=new TreeMap(Comparator c)

TreeMap t=new TreeMap(SortedMap m);

TreeMap t=new TreeMap(Map m)

TreeMap tm2 = new TreeMap(**new MyComparator()**);

tm2.put(100,"ZZZ");

tm2.put(103,"YYY");

tm2.put(101,"XXX");

tm2.put(104,106);

tm2.put(106,null);

System.out.println(tm2); **//{106=null, 104=106, 103=YYY, 101=XXX, 100=ZZZ}**

TreeMap tm1 = new TreeMap();

tm1.put(100,"ZZZ");

tm1.put(103,"YYY");

tm1.put(101,"XXX");

tm1.put(104,106);

tm1.put(106,null);

//tm.put("FFF",106); **//CCE - sorting is done on the basis of key(homogenous)**

//tm.put(null,107); **//NPE**

System.out.println(tm1); **//{100=ZZZ, 101=XXX, 103=YYY, 104=106, 106=null}**

Hashtable:

=> The Underlying Data Structure for Hashtable is Hashtable Only.

=> Duplicate Keys are Not Allowed. But Values can be Duplicated.

=> Insertion Order is Not Preserved and it is Based on Hashcode of the Keys.

=> Heterogeneous Objects are Allowed for Both Keys and Values.

=> null Insertion is Not Possible for Both Key and Values. Otherwise we will get Runtime Exception Saying NullPointerException.

=> It implements Serializable and Cloneable,but not RandomAccess.

=> Every Method Present in Hashtable is Synchronized and Hence Hashtable Object is Thread Safe**, so best suited when we work with Search Operation.**

Constructors:

1. Hashtable h = new Hashtable();
   Creates an Empty Hashtable Object with Default Initial Capacity 11 and **Default Fill Ratio 0.75.**
2. Hashtable h = new Hashtable(intinitialcapacity);
3. Hashtable h = new Hashtable(intinitialcapacity, float fillRatio);
4. Hashtable h = new Hashtable(Map m);

```
import java.util.Hashtable;
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");
        h.put(new Temp(23), "E");
        h.put(new Temp(16), "F");
        h.put("sachin",null); //RE: java.lang.NullPointerException
        System.out.println(h); //{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}
    }
}
class Temp{
    int i;
    Temp(int i){
        this.i=i;
    }
    public int hashCode(){
        return i;
    }
    public String toString(){
        return i+" ";
    }
}
```

Scenario2: public int hashCode(){ return i%9;}

Scenario3: Hashtable h=new Hashtable(25);
public int hashCode(){ return i;}

Properties:

=> It is the Child Class of Hashtable.

=> In Our Program if anything which Changes Frequently (Like Database User Name, Password, Database URLs Etc) Never Recommended to Hard Code in Java Program.

=> Because for Every Change in Source File we have to Recompile, Rebuild and Redeploying Application and Sometimes Server Restart Also Required, which Creates Business Impact to the Client.

=> To Overcome this Problem we have to Configure Such Type of Propertiesin Properties File.

=> The Main Advantage in this Approach is if a there is a Change in Properties File, to Reflect that Change Just Redeployment is Enough, which won't Create any Business Impact.

=> We can Use Properties Object to Hold Properties which are coming from Properties File.

=>Supports WORA(write once read anywhere).

Constructor:

Properties p = new Properties();

1. public String getProperty(String pname);
   To Get the Value associated with specified Property.
2. public String setProperty(String pname, String pvalue);
   To Set a New Property.
3. public Enumeration propertyNames();It Returns All Property Names.
4. public void load(InputStream is);
   To Load Properties from Properties File into Java Properties Object.
5. public void store(OutputStreamos, String comment);
   To Store Properties from Java Properties Object into Properties File

eg#1.

application.properties → properties file

```
#key for Oracle Environment
#Change values as per ur testing db environment
jdbcUrl = jdbc:oracle:thin:@localhost:1521:XE
user = System
password = pwskills123
```

Test.java

```
//By default searching happens in rt.jar → import
import java.util.*;
import java.io.*;
public class Test
{
    public static void main(String[] args)throws Exception
    {
        //Creating a Properties Object
        Properties p = new Properties();
        //Establishing Stream b/w java and .properties file
```

```
        FileInputStream fis = new FileInputStream("application.properties");
        //bind the data of .properties file to Properties Object
        p.load(fis);
        //Get the data from Properties Object
        String jdbcUrl = p.getProperty("jdbcUrl"); //not accessing key directly → enhancing oops
        String user = p.getProperty("user");
        String password = p.getProperty("password");
        //use this data as required by the application
        System.out.println(jdbcUrl);
        System.out.println(user);
        System.out.println(password);
    }
}
```

Output

jdbc:oracle:thin:@localhost:1521:XE

System

pwskills123

Now make changes in application.properties file and don't compile just run, we can directly see the new changes.

1.5 version enhancement of Queue

1. It is a child interface of Collection
2. If we want to represent a group of individual Objects before Processing then we should go for Queue.
3. From 1.5 version LinkedList also implements Queue
4. Usually Queue follows FIFO Order,Based on our requirement we can implement our own priority also.
5. LinkedList based implementation Queue also follows FIFO order.

eg: Before sending mail, we need to store the mail id in any one of the datstructure,the best suited datastructure is "Queue".

Important methods associated with Queue

1. **boolean offer (Object obj)**

   => to add object into the Queues

2. **Object peek()**

   => It return the head element of the Queue

   If Queue is empty it returns null.

3. **Object element()**

   => It return the head element of the Queue

   If Queue is empty it throws NoSuchElementException.

4. **Object poll()**

   => It remove and return the head element of the queue

   If Queue is empty it returns null.

5. **Object remove()**

   => It remove and return the head element of the Queue

   If Queue is empty it returns NoSuchElementException.

PriorityQueue

1. To process the elements before processing, we need to store the elements based on some priority order
2. Priority Order

   => natural sorting order

   => customized sorting order

3. insertion order => not preseved based on some sorting it will be added.
4. duplicate => not allowed.
5. null insertion => not allowed
6. heterogenous => if we depend on natural sorting order,no objects should homogenous and it should implements Comparable if it is customized sorting order,then Object can be heterogenous and it need not implements Comparable.

Constructor

1. PriorityQueue p=new PriorityQueue();

   //Default Capacity=> 11

   //Insertion order => based on default natural sorting order.

2. PriorityQueue p=new PriorityQueue(int initialCapacity);
3. PriorityQueue p=new PriorityQueue(int initialCapacity,Comparator comparator)
4. PriorityQueue p=new PriorityQueue(SortedSet s);
5. PriorityQueue p=new PriorityQueue(Collection c);

eg#1.

```
import java.util.PriorityQueue;

public class TestApp{
    public static void main(String... args){
        PriorityQueue p=new PriorityQueue();
        System.out.println(p.poll());//null
        System.out.println(p.element());//NoSuchElementException
        for (int i=0;i<=10 ;i++ ){
            p.offer(i);
```

```
        }
        System.out.println(p);  //[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        System.out.println(p.poll());  //0
        System.out.println(p);  //[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

        PriorityQueue q=new PriorityQueue(15,new MyComparator());
        q.offer("Z");
        q.offer("A");
        q.offer("L");
        q.offer("B");
        System.out.println(q);//[Z,L,B,A]
    }
}
class MyComparator implements Comparator{
    @Override
    public int compare(Object obj1,Object obj2){
        String s1=obj1.toString();
        String s2=obj2.toString();
        return s2.compareTo(s1);
    }
}
```

Note: Some Operating System wont provide support for PriorityQueue(so the above program's output could vary).

1.6 V enchancement of Collection

1. **NavigableSet**

=> It is the child interface of SortedSet

=> It defines several methods for Navigation purposes

floor(e) => it returns the highest element which is <=e

lower(e) => it returns the highest element which is <e

ceiling(e) => it returns the lowest element which is >=e

higher (e) => it returns the lowest element which is <e

pollFirst() => remove and return first element

pollLast() => remove and return last element

descendingSet() =>returns NavigableSet in descending order.

```
System.out.println(ts);//[1000, 2000, 3000, 4000, 5000]
System.out.println("Ceiling value :: "+ts.ceiling(2000));//2000
```

System.out.println("Higher value :: "+ts.higher(2000));//3000

System.out.println("Floor value :: "+ts.floor(3000));//3000

System.out.println("Lower value :: "+ts.lower(3000));//2000

System.out.println("Poll First :: "+ts.pollFirst());//1000

System.out.println(ts);//[2000,3000,4000,5000]

System.out.println("Poll Last :: "+ts.pollLast());//5000

System.out.println(ts);//[2000,3000,4000]

System.out.println("DescendingSet :: "+ts.descendingSet());//

2. **NavigableMap**

=> It defines several methods of Navigation purpose.

=> It is child interface of SortedMap.

NavigableMap defines the following methods

a. floorKey(e)

b. lowerKey(e)

c. ceilingKey(e)

d. higherKey(e)

e. pollFirstEntry()

f. pollLastEntry()

g. descendingMap()

tm.put(45,"rohith");

System.out.println(tm); //{7=dhoni, 10=sachin, 18=kohli, 19=dravid, 45=rohith}

System.out.println("Ceiling Key :: "+tm.ceilingKey(10)); //**10**

System.out.println("Higher Key :: "+tm.higherKey(10)); //**18**

System.out.println("Floor Key :: "+tm.floorKey(10)); //**10**

System.out.println("Lower Key :: "+tm.lowerKey(10)); //**7**

System.out.println("PollFirst :: "+tm.pollFirstEntry()); //**7=dhoni**

System.out.println("PollLast :: "+tm.pollLastEntry()); //**45=rohith**

System.out.println("DescendingMap :: "+tm.descendingMap()); // **{19=dravid, 18=kohli, 10=sachin}**

System.out.println(tm); //**{10=sachin, 18=kohli, 19=dravid}**

Collection vs Collections
Collections(c)

=> It is a utility class present in java.util package.
=> It defines the method meant for sorting,searching and reversing the elements

Note:
Collection(I)
|=> List
a. It won't speak about sorting,so use Collections(c).
|=> Set
a. If we want sorting then we can opt for TreeSet.
|=> Queue
a. If we want sorting then we can opt for PriorityQueue.

To sort the elemets of List

1. public static void sort(List l)**(default sorting if no 2nd argument is mentioned)**
   1. It sorts the element in ascending order/alphabetical order
   2.The elements should be homogenous and it should comparable otherwise it leads to ClassCastException.
   3.If it contains null,it would result in "NullPointerException".
2. public static void sort(List l,Comparator c)
   1. It sorts the elements based on our customization.

ArrayList al = new ArrayList();

Collections.sort(al);

Collections.sort(al,**new MyComparator()**);

binarySearch()

Searching Elements of List:

1. public static int binarySearch(List l, Object target);
   If we are Sorting List According to Natural Sorting Order then we have to Use this Method.
2. public static int binarySearch(List l, Object target, Comparator c);
   If we are Sorting List according to Comparator then we have to Use this Method.

Conclusions:

=> Internally the Above Search Methods **will Use Binary Search Algorithm.**

=> **Before performing Search Operation Compulsory List should be Sorted.** Otherwise we will get Unpredictable Results.

=> Successful Search Returns Index.

=> Unsuccessful Search Returns Insertion Point.

=> Insertion Point is the Location where we can Insert the Target Element in the SortedList.

=> If the List is Sorted according to Comparator then at the Time of Search Operation Also we should Pass the Same Comparator Object. Otherwise we will get Unpredictable Results.

```
import java.util.*;
//Client Code
public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add("Z");
        al.add("A");
        al.add("M");
        al.add("K");
        al.add("a");
        //Collections
        Collections.sort(al);
        System.out.println("After Sorting :: "+al);//[A, K, M, Z, a]
        //BinarySearch :: success case -> index
        //BinarySearch :: failure case -> insertion point
        System.out.println("Index of Z is :: "+Collections.binarySearch(al,"Z"));
        System.out.println("Index of J is :: "+Collections.binarySearch(al,"J")); //-2
        System.out.println("Index of X is :: "+Collections.binarySearch(al,"X"));

        ArrayList al = new ArrayList();
        al.add(15);
        al.add(0);
        al.add(20);
        al.add(10);
        al.add(5);
        //Collections
        Collections.sort(al,new MyComparator());
        System.out.println("After Sorting :: "+al);//[20,15,10,5,0]
        //BinarySearch :: success case -> index
        //BinarySearch :: failure case -> insertion point
        System.out.println(Collections.binarySearch(al,10,new MyComparator()));//index
        System.out.println(Collections.binarySearch(al,13,new MyComparator()));//insertionpoint
        System.out.println(Collections.binarySearch(al,17,new MyComparator()));//insertionpoint
```

```
    }
}
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        //logic for sorting
        Integer i1= (Integer)obj1;
        Integer i2= (Integer)obj2;
        return -i1.compareTo(i2);
    }
}
```

0 1. 2. 3. 4. → **success case**

[A, K, M, Z, a]

-1 -2. -3 -4. -5 → **failure case**

Eg: For the List of 3 Elements

A B Z →

1. Range of Successful Search: 0 To 2
2. Range of Unsuccessful Search: **-4 To -1**
3. Total Result Range: **-4 To 2**

Note: For the List of n Elements

1. Successful Result Range: 0 To n-1
2. Unsuccessful Result Range: -(n+1) To -1
3. Total Result Range: -(n+1) To n-1

Reversing the Elements of List:

public static void reverse(List l);

reverse() Vs reverseOrder():

=> We can Use reverse() to Reverse Order of Elements of List.

=> We can Use reverseOrder() **to get Reversed Comparator.**

Comparator c1 = Collecctions.reverseOrder(Comparator c);

| |

Descending Order Ascending Order

```
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        //Sort:: Descending order
        Integer i1 = (Integer) obj1;
        Integer i2= (Integer) obj2;
        return -i1.compareTo(i2);
    }
}
```

Comparator c2 =Collections.reverseOrder(c1); **//c2 → ascending order, c1 → Descending order**

Collections.**sort(al,c2)**;

System.out.println("ReverseOrder Sorting :: "+al);//[0,5,10,15,20]

Arrays

Sorting Elements of Array:

1. public static void sort(primitive[] p); To Sort According to Natural Sorting Order.
2. public static void sort(Object[] o); To Sort According to Natural Sorting Order.
3. public static void sort(Object[] o, Comparator c);**To Sort According to Customized Sorting Order.**

Note → we can Sort primitive[] Only Based on Natural Sorting(not with Comparator c).

String[] names = {"sachin","saurav","dhoni","kohli","azarudin"};

Arrays.sort(names);

Arrays.sort(names,new MyComparator());

Searching the Elements of Array

1. public static int binarySearch(primitive[] p, primitive target);
   If the Primitive Array Sorted According to Natural Sorting Order then we have to Use this Method.
2. public static int binarySearch(Object[] a, Object target);
   If the Object Array Sorted According to Natural Sorting Order then we have to Use this Method.
3. public static intbinarySearch(Object[] a, Object target, **Comparator c**);
   If the Object Array Sorted According to Comparator then we have to Use this Method.

Note: All Rules of Array Class binarySearch() are Exactly Same as Collections Class binarySearch().