Collection framework

- **Collection**
- **Map**
- **Collections class**
- **Arrays class**

Collection framework

- An array is an indexed Collection of fixed number of homogeneous data element.

➢ Limitation of object array:

1. Arrays are fixed in size i.e. once we created an array there is no chance of increasing or decreasing size based on our requirement. Hence, to use array concept we should know the size in advance, which may not possible always.

Arrays can hold only homogeneous data elements i.e. (same type)

Example:

```
Student [] s= new Student[1000];
s[0]= new Student[];    //Valid
s[1]= new Student[];    //Valid
s[2]= new Customer[];   //Invalid    C.E.- Incompatible type
                                      required: student
                                      found: customer
```

- But we resolve this problem by using object type arrays.

  Example:

```
Object[] a= new Object[1000];
    a[0]= new Student[];    // Valid
    a[0]= new Customer[];   // Valid
```

1) Array concept is not built on some data structure. Hence, readymade method support is not available for every requirement. Compulsory programmer is responsible to write the logic.

- To resolve the above problems Sun people introduced "Collections" concept.

➢ Advantages of Collections over array:

1) Collections are growing able in nature. Hence, based on our requirement we can increase or decrease the size.
2) Collections can hold both Homogeneous and Heterogeneous object.
3) Every collection class is implemented based on same data structure. Hence, readymade method support is available for every requirement.

➢ Disadvantages of Collections:

Performance point of view Collections are not recommended to use this is the limitation of Collections.

➢ Difference between Array and Collections:

Array	Collection
1) Arrays are fixed in size	1)Collections are grow able in nature
2) Memory point of view arrays concept is not recommended to use.	2)Memory point of view collections concept is highly recommended to use
3) Performance point of view arrays concept is highly recommended to use.	3) Performance point of view collections is not recommended to use.
4) Arrays can hold only homogeneous data elements.	4) Collections are hold both homogeneous and heterogeneous objects.
5) There is no underlying data structure for arrays. Hence, readymade method support is not available.	5) Underlying data structure is available for every collection class. Hence, readymade method support is available.

6) Arrays can be used to hold both primitives and object.	6) Collections can be used to hold only objects but not for primitive.

➢ Collection:

Definition: - A group of individual objects as a single entity is called "Collection".

➢ Collection Framework:

- It defines several classes and interfaces, which can be used to represent a group of objects as a single entity.

➢ Terminology:

Java	C++
Collection	Container
Collection Framework	STL (Standard Template Library)

➢ Key Interfaces of Collection Framework:

1) Collection (Interface):

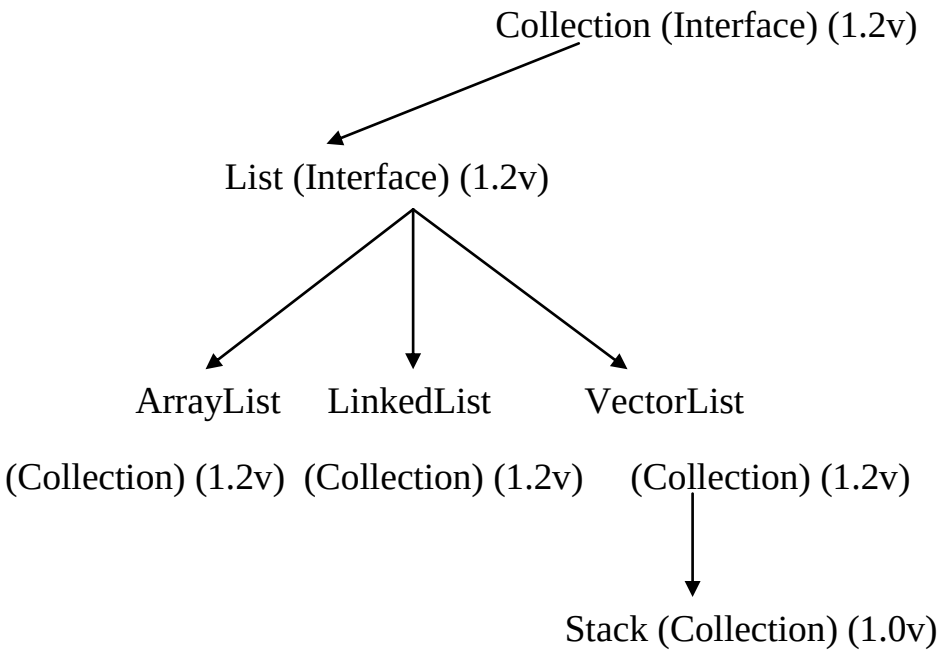
- If we want to represent a group of individual objects as a single entity then we should go for Collection.
- In general collection interface is considered as root interface of collection framework.
- Collection interface defines the most common methods which can be applicable for any collection object.

Collection Vs Collections:

- Collection is an interface, can be used to represent a group of individual object as a single entity.
- Collection is an "Utility class" present in java util package, to define several utility methods for collections.

2) List (Interface):

- It is the child interface of collection.
- If we want to represent a group of individual objects where insertion order is preserved and duplicate are allowed. Then we should go for List.

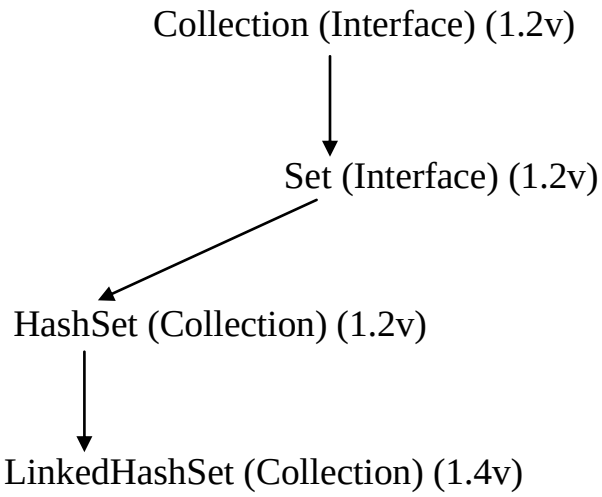


- Vector and stack classes are reengineered in 1.2 version to Set into collection framework.

3) Set (Interface):

- It is the child interface of interface of collection.
- If we want to represent a group of individual objects where "duplicates not allowed and insertion order is not preserved". Then we

should go for “Set”.

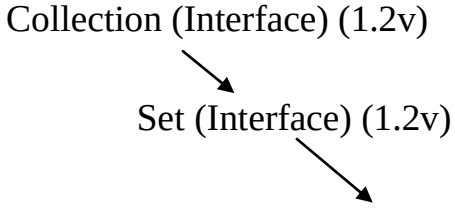


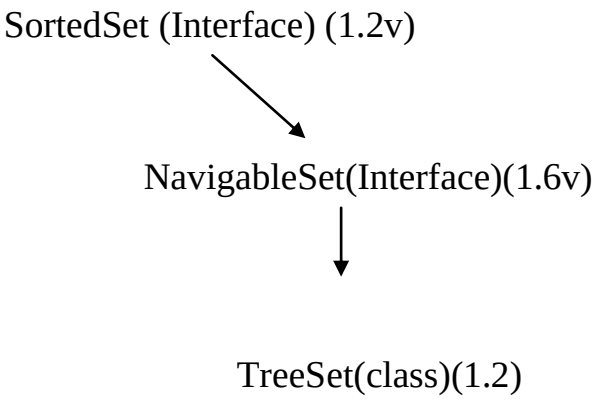
4) SortedSet (Interface):

- It is the child interface of Set.
- If we want to represent a group of individual objects. According to some sorting order then we should go for Sortedset.

Navigable Set (Interface):

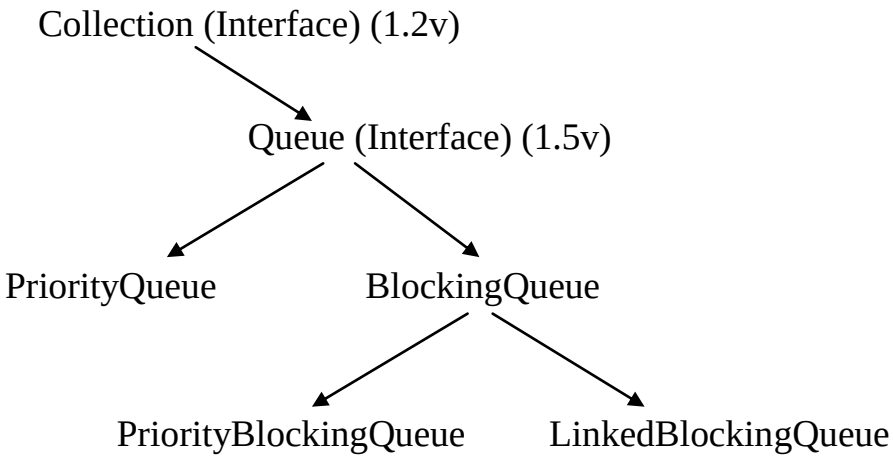
- It is the child interface of SortedSet. To provide several methods for Navigation purposes.
- It is introduced in 1.6version.





5) Queue (Interface)(1.5v):

- It is the child interface of Collection.
- If we want to represent a group of individual objects prior to processing. Then we should go for queue.



Note:

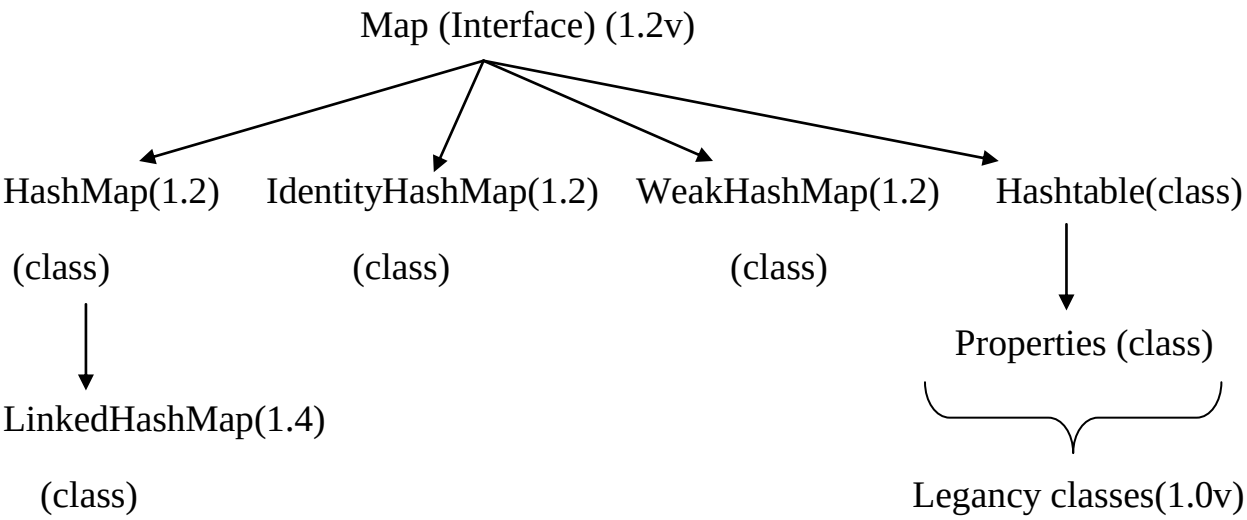
All the above Interfaces (Collection, List, Set, SortedSet, NavigableSet, Queue) development for representing a group of individual objects.

- If we want to represent a group of objects as key-value pairs. Then we should go for "Map".

6) Map (Interface):

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

- Both key and value are objects only.
- Duplicate keys are not allowed. But values can be duplicated.



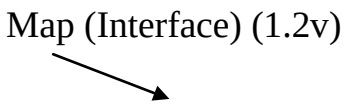
Note: Map is not child interface of collection.

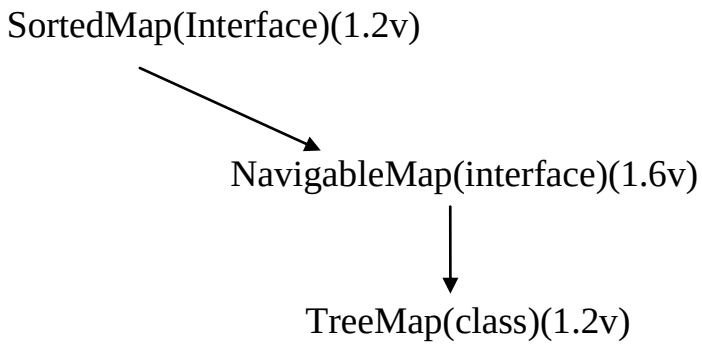
7) SortedMap (Interface):

- If we want to represent a group of objects as key-value pairs according to same sorting order. Then we should go for SortedMap.
- Sorting should be done only based on keys, but not based on values.
- SortedMap is child interface of Map.

8) Navigable Map (Interface):

- It is the child interface of SortedMap and defines several methods for Navigation purposes.





➢ Collection Framework:

Collection (Interface):

- If we want to represent a group of individual objects as a single entity then we should go for "Collection".
- Collection interface defines the most common methods which can be applied for any collection object.
- The following is the List of methods present in collection interface.

1) boolean add(Object o)
2) boolean addAll(Collection c)
3) boolean remove(Object o)
4) boolean removeAll (Collection c)
5) boolean retainAll (Collection c)

To remove all objects except those present in c.

6) void clear()
7) boolean isEmpty ()
8) int size()
9) boolean Contains (Object o)
10) boolean ContainsAll (Collection c)
11) Object[] to Array()
12) Iterator iterator()

➢ List(Interface):

- List is the child interface of collection.
- If we want to represent a group of individual objects where duplicate objects are allowed and insertion order is preserved by mean of Index.
- We can differentiate duplicate objects by using Index. Hence, Index place as very important role in List.
- List interface defines the following methods.

1) boolean add(int indext, Object o)
2) boolean addAll(int indext, Collection c)
3) Object remove(int index)
4) Object get(int index)
5) Object(old) set(int indext, Object new)
6) int indexOf(Object o)
7) int lastIndexOf(Object o)
8) ListIterator listiterator()

- It contains classes:
  1) ArrayList(c)
  2) LinkedList(c)
  3) VectorList(c)
  4) Stack(c)

1) ArrayList(c):-

- *The underlying data* structure for ArrayList is resizable array or Growable Array.
- Insertion order is preserved.
- Duplicate objects are allowed.
- Heterogeneous objects are allowed.
- Null insertion is possible.

Constructors:-

1) ArrayList Al = new ArrayList();

- Creates an empty ArrayList object, with default initial capacity '10'.
- Once Al reaches its max capacity. Then a new Al object will be created with

New capacity = currentCapacity*3/2+1

2) ArrayList l = new ArrayList(int initialCapacity);

- Creates an empty ArrayList object with the specified initial capacity.

3) ArrayList l = new ArrayList(Collection c);

- Creates an equivalent ArrayList object for the given collection object i.e. this constructor is for dancing between collection objects.

Example:

```
import java.util.*;
class ArrayListDemo
 {
public static void main(String[] args)
  {
  ArrayList a = new ArrayList();
   a.add("A");
    a.add(10);
   a.add("A");
   a.add(null);
   System.out.println(a);  //Output=[A,10,A,null]
   a.remove(2);
   System.out.println(a);//Output=[A,10,null]
   a.add(2, "M");  // [A,10,M,null]
   a.add("N");  // [A,10,M,null,N]
```

```
        System.out.println(a);// Output=[A,10,M,null,N]
        System.out.println(a.size()); //5
    }
}
```

Note:

- In every collection class toString() is overridden to return its content directly in the following format.
  [obj1, obj2, obj3, ….]
- Usually we can use collection to store and transfer objects to provide support for requirement every collection class implements serializable and clonable interfaces.
  Advantage:
- ArrayList and vector classes implements "RandomAccess" interface, so that any random element we can access with same speed. Hence, if our frequent operation is retrievable operation the best suitable data structure is ArrayList.
  Disadvantage:
- If our frequent operation is insertion or deletion operation in the middle then ArrayList is the worst choice, because it required several sift operations.

➢ **Difference between ArrayList And Vector:**

ArrayList	Vector
1)No method is synchronized.	1) Every method is synchronized.
2)Multiple threads can access ArrayList simultaneously, hence ArrayList object is not threadsafe.	2) At any point only one thread is allowed to operate on vector object at a time. Hence, vector object is threadSafe.

3)Thread are not required to wait , and hence performance is high.	3) It increases waiting time of threads and hence performance is low.
4)Introduced in 1.2 version and hence it is non-legancy.	4) Introduced in 1.0 version and hence it is legancy.

Qu)How to get Synchronized version of arraylist?

Ans:By usin collections class SynchronizedList() we can get Synchronized version of arraylist.

public static List SynchronizedList(List l)

Example:

ArrayList l=new ArrayList();

List l1=collections SynchronizedList(l);

Synchronized Non synchronized

- Similarly we can get Synchronized version of set and map objects by using the following methods respectively.

1) public static Set SynchronizedSett(Set s);
2) public static Map SynchronizedMap(Map m)

Note:Of our frequent operation is insertion or deletion in the middle then ArrayList is not recommended to handle this requirementwe should go for LinkedList.

2)Linked List(c):-

- The underlying datastructure is Double LinkedList.
- Insertion oreder is preserved.
- Duplicate objects areallowed.
- Heterogeneous is possible.
- Null insertion is possible.
- Implements serializable and clonnable but not random access interface.
- Best suitable if an frequent operation insertion or deletion in middle.
- Worst choice if our frequent operation is retrival

Constructors:

1)LinkedList l=new LinkedList();

Create an empty linked list object.

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

For Interconversionbetween collection objects.

LinkedList

Specific method:

- Usually we can use LInkedList to implement stack and Queue to support this requirement LinkedLIst class defines the following six specific methods.
  1) void addFirst(Object o);
  2) void addLast(Object o);
  3)Object removeFirst();
  4) Object removeLast();
  5)Object getFirst();
  6) Object getLast();

  Example:
  import java.util.*;

```
class LinkedListDemo
{
public static void main(String ar[])
{
LinkedList l=new LInkedList();
l.add("CST");
l.add(30);
l.add(null);
l.add("CST");
l.set(0,Software);
l.add(0,vencky);
l.removeLast();
l.addFirst("ccc");
System.out,println(l);

}
}
```

Output:ccc,vencky,Software,30,null

2)Vector(c):-

- The underlying data structure is Resizable Array or Growable Array.
- Insertion oreder is preserved.
- Duplicate objects are allowed.
- Heterogeneous object is possible.
- Null insertion is possible.
- Implements serializable and clonnable and random access interface.
- Best suitable if our frequent operation retrival
- Worst choice if our frequent operation is insertion and deletion in middle.
- Every method in vector is Synchronized hence vector object is threadsafe.

Constructors:

1)Vector v=new Vector();

Create an empty vector object with default initial capacity 10.

Once vector reaches its maximum capacity a new vector object will be created with double capacity.

New capacity=2*current capacity.

2)Vector v=new Vector(int initialCapacity);

3)Vector v=new Vector(int initialCapacity,int inceremental Capacity);

4)Vector v=new Vector(Collection c);

Vector specific method:

1)To add Objects:

- add(Object o) ⟶ (c)
- add(int index,Object o) ⟶ (l)
- addElement(Object obj) ⟶ (v)

2)To remove elements or Objects:

- remove(Object o) ⟶ (c)
- removeElement(Object o) ⟶ (v)
- remove(int index) ⟶ (l)
- removeElementAt(int index) ⟶ (v)
- clear() ⟶ (c)
- removeAllElement() ⟶ (v)

3)To retrive elements or Objects:

- get(int index) ⟶ (l)
- elementAt(int index) ⟶ (v)
- firstElement() ⟶ (v)
- lastElement() ⟶ (v)

4)Other methods:

- int size();
- int capacity();
- Enumeration elements()

Example:

```
import java.util.*;
clas vdemo
{
Public static void main(String ar[])
{
Vector v=new Vector();
System.out,println(v.capacity());
for(int i=1;i<=10;i++)
{ v.addElement(i)
}
System.out,println(v.capacity());
v.addElement("A");
System.out,println(v.capacity);
System.out,println(v);

}

}
```

Otput:

10
10
20
[1,2,3,4,-----10,A]

4)Stack(c):[LIFO]:

- It is a child class of vector contains only one constructor.
  Constructor:
  Stack s=new Stack();

Methods:

1)Object push(Object o):

To insert an object into stack.

2) Object pop():

To remove and returns top of stack.

3)) Object peek():

To returns top of stack.

4)boolean empty();

returns true when stack is empty.

5)int search(Object o);

returns the offset from top of the stack.

If the object is available,otherwise returns -1

Example:

```
import java.util.*;
class StackDemo
{
public static void main(String args[])
    {
Stack s=new Stack();
s.push ("A");
s.push ("B");
s.push ("C");
```

```
System.out.println(s);
System.out.println(s.Search ("A"));        //3
System.out.println(s.Search ("Z"));        //-1
}
}
```

Limitation & Enumeration:

- Enumeration concept is applicable only for legacy classes & hence it is not a universal cursor.
- By using enumeration we can get only ReadAccess & we can't perform any remove operation.
- To over come these limitation SUN people introduced Iterator in 1.2 version.

Iterator:

- We can apply Iterator concept for any collection object It is universal cursor.
- While iterating we can perform remove operation also ,in a addition to read operation.
- We can get Iterator object by Iterator() of collection interface.

  Iterator itr=c.iterator()

- Iterator interface defines the following 3 Methods.
  - i) public boolean hasNext();
  - ii) public object next();
  - iii) public void remove();

Limitation of Iterator:

- In the case of iterator & enumeration we can always move towords direction & we can't move backword direction. i.e These cursor are single direction cursor but not bidirectional.
- While performing iteration we can perform only read & remove operations.

- We can't perform replacement & addition of new objects.
  To resolve these problem SUN people introduced ListIterator in 1.2 version.

  Example:

```
import java.util.*;
    class iteratorDemo
     {
     public static void main(String args[])
     {
   ArrayList l=new arrayList();
for(int i=0;i<=10;i++)
{
l.add(i);
}
  System.out.println(l);
Iterator itr =i.iterator();
While(itr.hasNext())
{
```

```
Integer I=(Integer)itr.next();
if(I%2==0)
{
System.out.println(I);
}
else
{
itr.remove();
}
}
 //System.out.println(l);      [0,2,4,6,8,10]
```

List Iterator:

- List Iterator is the child interface of iterator.
- While Iterator object by ListIterator we can always move to the forword or to thee backword direction. i.e ListIterator is a bidirectional cursor.
- While iterating by ListIterator we can perform replacement & addition of new objects also in addition to read & remove operations.
- We can create ListIterator objects by using ListIterator of List interface.
  ListItearator litr=l.listIterator();
- ListIterator interface defines the following 9 methods
  i) public boolean hasNext();
  ii) public object next();
  iii) public int nextIndex();

iv) public boolean hasPrevious();
v) public object Previous ();
vi) public int PreviousIndex();

vii) public void remove();
viii) public void set(object new);
ix) public void add(object new);

Example:

```
import java.util.*;
class ListIteratorDemo
{
public static void main(String args[])
{
LinkedList l=new LinkedList();
l.add("balkrushna");
l.add("venky");
l.add("chiru");
l.add("nag");
System.out.println(l);                [balkrushna,venky,chiru,nag]
ListIterator ltr=l.ListIterator();
while(ltr.hasNext())
{
String s=(String)ltr.next();
if(s.equals("venki"))
```

```
    {
   ltr.remove();
    }
    if(s.equals("chiru"))
   {
   ltr.set("charan");
  }
  if(s.equals("nag"))
 {
 ltr.add("chottu");
 }
 }
System.out.println(l);                    [Balakrushna,charan,nag,,chottu]
}
}
```

Note:-

-Among three cursors ListIterator is the most powerful cursor But it is applicable only for List objects.

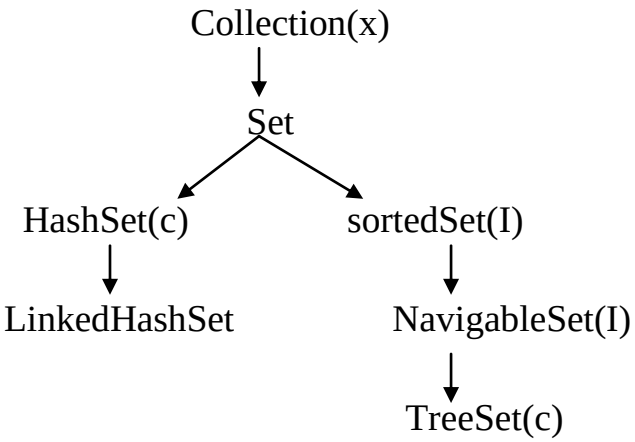
Comparison table of three Cursors:-

Property	**Enumeration**	**Iterator**	**ListIterator**
1)It is legacy	Yes	No	No
2)It is applicable	Only for legacy classes	For any collection objects.	Only for List objects.
3)Movement	Single direction. (only forword)	Single direction.(only	Both direction.(forword

		forword)	and backword)
4)How to get it?	By using elements() method.	By using iterator()	By using ListIterator()
5)Accessibility	Only read	Read and remove	Read,remove, replace,add
6)Method	hasMoreElements() nextElement()	hasNext(),next(), remove()	methods

Set(x):

- Set is a child interface of collection.
- If we want to represent a group of objects where duplicates are not allowed & insertion order is not preserved.Then we should go for Set.



Set Interface does'nt contain any method we have to use only collection interface method.

i) HashSet(c):

- The underlying data structure is HashTable.
- Heterogheneous objects are allowed.
- Duplicate objects are not allowed but.
- Insertion order is not preserved & all objects are inserted accending.

- If we are trying to add duplicate objects we want to get any C.E. or R.E. add() simply return false.
- Null insertion is allowed(only once) because Duplicate objects are not allowed but.
- HashSet implements serializable & clonable interfaces.

Example:

```
import java.util.*;

class HashSetDemo

{

public static void main(String args[])

    {

HashSet s=new HashSet ();

h.add("B");

h.add("C");

h.add("D");

h.add("Z");

h.add(null);

h.add(10);

System.out.println(h.add("Z");       //false

System.out.println(h);                  //[null,D,B,C,10,Z]

}

}
```

O/P-false

[null,D,B,C,10,Z]

Note:

Insertion order is not preserved.

Constructor:

1)HashSet h=new HashSet();

Creates an empty HashSet object with default initial capacity level is '16' & default fillRatio 0.75(75%).

2)HashSet h=new HashSet(int initialCapacity);

Creates an empty HashSet object with specified initial capacity & default fillRatio 0.75(75%).

3) HashSet h=new HashSet(int initialCapacity,float fillRatio);

4) HashSet h=new HashSet(Collection c);

Fill ratio:

After completing the specified ratio then only a new HashSet object will be created that particular ratio is called fillratio of load factor.

The default fillratio is 0.75 but we can customized this value.

Example:

ii) LinkedHashSet(c):

LinkedHashSet is the child class of HashSet.

It is exactly same as HashSet except the following differences.

HashSet	LinkedHashSet
i)The underlying D.S. is HashTable.	i)The underlying D.S. is a combination of HashTable & LinkedList.
ii)Insertion order is not preserved.	ii)Insertion order is not preserved.
iii)Introduced in 1.2 version.	iii)Introduced in 1.4 version.

In the above program if we are replacing HashSet with Linked HashMap the following is the o/p.

o/p-[B,C,D,Z,null,10]

i.e insertion order is preserved.

Note:

The main important application area of LinkedHashSet & LinkedHashMap is implementing cache applications.where duplicates are not allowed & insertion order must be preserved.

ii) SortedSet(I):

- It is the child interface of Set.
- If we want to represent a group of individual objects according to some sorting order. Then we should go for SortedSet.
- SortedSet interface Defines the following 6 specific methods.
    - i) Object first()
      Returns the first element of sortedSet.
    - ii) Object last()
      Returns the last element of sortedSet.
    - iii) SortedSet headset(Object obj)

Returns the SortedSet whose elements are less than obj.

iv) SortedSet tailSet(object obj):

Return the SortedSet whose element are greater than equal to obj.

iv) SortedSet subSet(object obj1, object obj2):

Return the SortedSet whose elements are greater than equal to obj1 but obj2.

vi) Comparator comparator():

Return the comparator object describe underling Sorting technique.
If we used default natural sorting order then we will get null.

Note:
The default natural sorting order for the no's in asending order.
The default natural sorting order for characters string is alphabetical order(dictionary based order).

TreeSet(c):

- The underlying data structure is balanced tree.
- Duplicate objects are not allowed.
- Insertion order is not preserved .because objects will be inserted according to some sorting order.
- Heterogeneous objects are not allowed. Otherwise we will get ClassCastException & Null insertion is not possible .

Constructions:

- TreeSet t=new TreeSet();
  Creates an empty TreeSet object where the sorting order is default natural sorting order.
- TreeSet t=new TreeSet(Comparator c);
  Creates an empty TreeSet object where the sorting order is customized sorting order specified by comparator object.
- TreeSet t=new TreeSet(Collection c);
- TreeSet t=new TreeSet(SortedSet c);

Example1:

```
    import java.util.*;
class TreeSetDemo
 {
 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));        ClassCastException
// t.add(null);         NPE
  System.out.println(t);          //[A,B,Z,L,a]
    }
 }
```

Null acceptance:

- For The non empty tree set if we are trying to insert null we will get NullPointerException(NPE).
- For The non empty tree set add the first element null insertion is always possible.
- But after inserting that null if we are trying to insert any other we will get NPE.

Example1:

```
import java.util.*;
class TreeSetDemo1
   {
public static void main(String args[])
{
TreeSet t=new TreeSet();
 t.add(new StringBuffer("A");
t.add(new StringBuffer ("Z");
t.add(StringBuffer ("L");
t.add(new StringBuffer ("B");
  System.out.println(t);
        }
      }
```

O/P-CCE

- If we are depending on default natural sorting order compulsory objects should be homogeneous & comparable otherwise we will get ClassCastException(CCE).
- An object is said to be comparable iff the corresponding class implements comparable interface.
- String class & all wrapper class already implements comparable interface where as StringBuffer doesn't implements comparable interfaceHence In the above example we get CCE.

Comparable Interface:

- This interface present in java.lang.package & contains only one method is "compareTo".

public int compareTo(Object obj)

Obj1. compareTo(obj2)

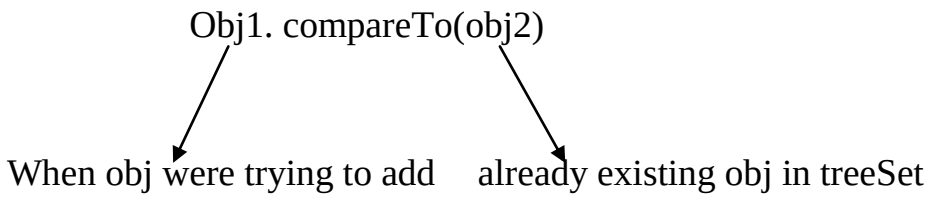
- Returns –ve iff obj1 has to come before obj2.
- Returns +ve iff obj1 has to come before obj2.
- Returns 0 iff obj1 & obj2 are equal(duplicate).

Example1:

```
import java.util.*;
class Test
  {
public static void main(String args[])
{
  System.out.println("A".compareTo("Z"));      //-ve     -25
  System.out.println("Z".compareTo("K"));      //+ve       15
```

```
System.out.println("A".compareTo("A"));       //0            0
    }
  }
```

- When were are depending on default natural sorting order internally jvm calls compareTo().
- Based on the return type jvm identifies the location of the element in sorting order.



- Returns –ve iff obj1 has to come before obj2.
- Returns +ve iff obj1 has to come before obj2.
- Returns 0 iff obj1 & obj2 are equal(duplicate).

Example1:

```
TreeSet t=new TreeSet();
 t.add("Z");
t.add("K");              //"K".compareTo("Z");   -ve
t.add("D");            //"D".compareTo("K");   -ve
t.add("M");            //"M".compareTo("D");   +ve
t.add("D");            //"M".compareTo("Z");   -ve
//t.add(null);          ////"D".compareTo("D");   0
```

```
System.out.println(t);        //[D,K,M,Z]
    }
    }
```

- If we are not satisfied with default natural sorting order on if the default natural sorting order is not already available. Then we can define our own customized sorting by using comparator.
- Comparable meant for default natural sorting order
- Comparable meant for customized natural sorting order.

Comparator(I):

- This interface present in java.util.package & defines the following 2 method .
  1) public int compare(Object obj1, Object obj2)
- Returns –ve iff obj1 has to come before obj2.
- Returns +ve iff obj1 has to come before obj2.
- Returns 0 iff obj1 & obj2 are equal(duplicate).

Obj1=Which obj were trying to add

Obj2=already existing object.

1) public Boolean equals(Object obj)

- When ever we are implementing comparator interface compulsory we should provide implementation for compare() 2nd method .equls() implementation is optional because it is already available for our class from object class through inheritance.

Example:

```
import java.util.*;
class TreeSetDemo
 {
 public static void main(String args[])
 {
TreeSet t=new TreeSet (new MyComparator());
t.add(20);
 t.add(0);
  t.add(15);
t.add(5);
t.add(10);
System.out.println(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 +100;
else if(I1>I2)                return ((I1<I2)?=1:(I1>I2?-1:0));
return -1000;
else
return 0;
}
}
```

- If we are not passing comparator object at line1.Then JVM internally calls compareTo() which is ment for default natural sorting order.In this case the output is[0,5,10,15,20].
- If we are passing comparator object at 1 then our own compare method will be executed which is ment for customized sorting order.These case the output is [20,15,10,5,0]

Various alternatives of implementing compare();

```
   class MyComparator implements Comparator
  {
   public int Compare(Object obj1,Object obj2)
  {
   Integer I1=(Integer)obj1;
   Integer I2=(Integer)obj2;
//return I1.compareTo(I2);=>[0,5,10,15,20]
//return- I1.compareTo(I2);=>[20,15,10,5,0]
```

```
//return I2.compareTo(I1);=>[ 20,15,10,5,0]

//return- I2.compareTo(I1);=>[0,5,10,15,20]

//return -1;=>[10,5,15,0,20] Reverse of insertion order

//return +1;=>[20,0,15,5,10]Insertion order

//return 0; =>[20]
```

Que) Write a programe to insert String objects into the Treeset where the sorting order is reverse of alphabetical order

```
import java.util.*;

class TreeSetDemo

 {

 public static void main(String args[])

 {

TreeSet t=new TreeSet (new MyComparator());

t.add("A");

 t.add("Z");

  t.add("K");

t.add("B");

t.add("a");

System.out.println(t);

 }

 }

class MyComparator implements Comparator

{
```

```
public int Compare(Object obj1,Object obj2)
{
String s1=(String)obj1;
String s2=obj2.toString(s2);
return –s1.compareTo(s2);
}
}
```

Note:-

-In objects and StringBuffer there is no compareTo(),So we can convert into object.

Que) Write a program to insert String and StringBuffer objects into the Treeset where the sorting order increasing Length order.If two objects having the same Length then consider their alphabetical order.

```
import java.util.*;
class TreeSetDemo
{
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");
System.out.println(t);       [A,AA,XX,ABC,ABCD]
}
}
class MyComparator implements Comparator
{
public int Compare(Object obj1,Object obj2)
{
String s1=obj1.toString();
String s2=obj2.toString();
int l1=s1.length();
int l2=s2.length();
if(l1<l2)
return -1;
else(l1>l2)
return +1;
else
return s1.compareTo(s2);
}
}
```

Que) Write a program to insert StringBuffer objects into the Treeset where the sorting order alphabetical order.

```
import java.util.*;
class TreeSetDemo
 {
 public static void main(String args[])
 {
 TreeSet t=new TreeSet (new MyComparator());
 t.add("A");
 t.add("Z");
 t.add("K");
 t.add("L");
System.out.println(t);       [A,K,L,Z]
 }
 }
 class MyComparator implements Comparator
 {
public int Compare(Object obj1,Object obj2)
{
 String s1=obj1.toString();
 String s2=obj2.toString();
 return s1.compareTo(s2);
```

```
  }
 }
```

Output:-[A,K,Z,L]

Note:-

- In StringBuffer There is ni compareTo() method so we can not convert into String.
- If we are depending on default natural sorting order Compuslary object should be Homogeneous and Comparable,otherwise we will get ClassCastException.
- If we are depending on our own sorting by Comparator the objects need not be Comparable and Homogeneous.

Comparable Vs Comparator:-

- For predefined comparable classes default natural sorting order is already avilable if we are not satisfied with that we can define our own customized sorting by using comparator.
  e.g-String
- For predefined Non-Comparable classes default natural sorting order is not available compulsary we should define sorting by using Comparator object only.
  e.g-StringBuffer.
- For our own customized classes to define default natural sorting order we can go for Comparable and to define Costomized Sorting we should for Comparator.
  e.g-Employee,Student,Customer.

Example:

```
import java.util.*;

class Employee implements Comparable

{
```

```
int eid;
Employee(int eid)
{
this.eid=eid;
}
public String toString()
{
return “E-” +eid;
}
public int compareTo(Object obj)
{
int eid1=this.eid;
Employee e2=(Employee)obj ;
int eid2=e2.eid;
if(eid1<eid2)
return -1;
elseif(eid1>eid2)
return +1;
else
return 0;
}
}
class CompCompDemo
```

```
{
public static void main(String args[])
{
Employee e1=new Employee(200);
Employee e2=new Employee(500);
Employee e3=new Employee(500);
Employee e4=new Employee(700);
TreeSet t1=new TreeSet();
t1.add(e1);
t1.add(e2);
t1.add(e3);
t1.add(e4);
t1.add(e5);
Sustem.out.println(t1);     [E-100,E-200,E-500,E-100]
TreeSet t2=new TreeSet(new MyComparator());
t2.add(e1);
t2.add(e2);
t2.add(e3);
t2.add(e4);
t2.add(e5);
Sustem.out.println(t2);     [E-700,E-500,E-200,E-100]
}
}
```

```
class MyComparator implements Comparator
{
public int Compare(Object obj1,Object obj2)
{
Employee e1=(Employee)obj1;
Employee e2=(Employee)obj2;
return e2.compareTo(e1);
return -e1.compareTo(e2);
}
}
```

Comparison between Comparable and Comparator

Comparable	**Comparator**
1)We can use Comparable to define default natural sorting order.	1)We can use Comparator to define Customized Sorting order.
2)This interface present in java.lang. package.	2)This interface present in java.util.package.
3)Defines only one method i-e compareTo().	3)Defines two methods 1)compare() 2)equals().
4)All wrapper class and String class implements comparable interface	4)No predefined class implements Comparator interface.

Comparison table for Set implemented classes:-

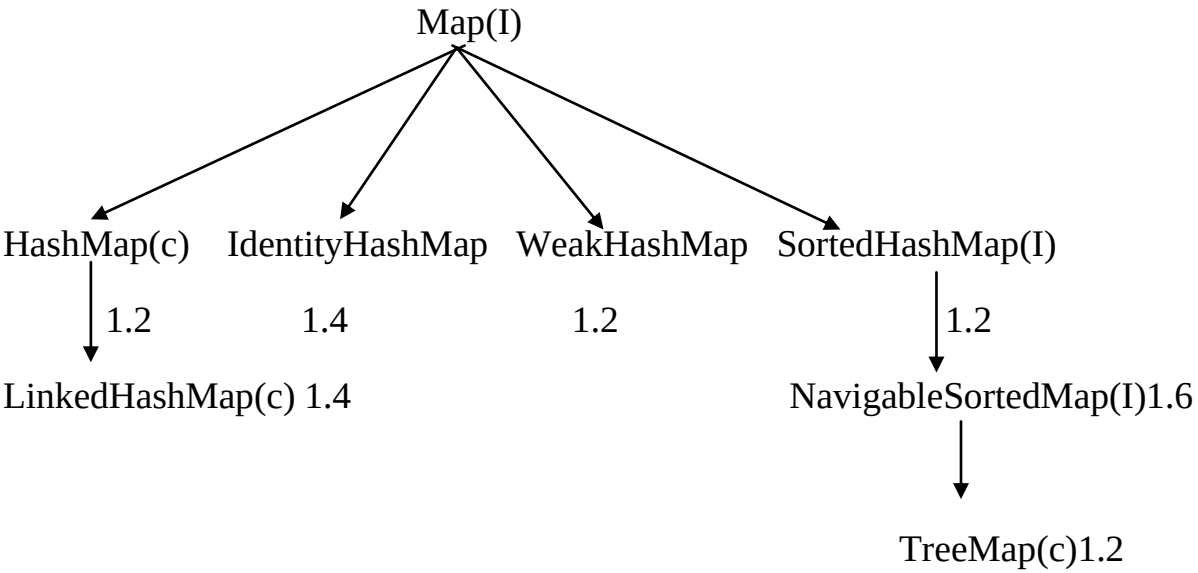
Property	HashSet	LinkedHashSet	TreeSet
1)Underlying Data structure	Hash table	Hashtable + LinkedList	Balanced Tree
2)Insertion Order	Not preserved	Preserved	Not preserved.
3)Sorting Order	Not Allowed	Not Allowed	Preserved
4)Heterogeneous objects	Allowed	Allowed	Not Allowed
5) Duplicate Objects	Not Allowed	Not Allowed	Not Allowed
6)Null Aceptance	Allowed(1)	Allowed(1)	For the Empty TreeSet add the first element null insertion is possible.In all other case we will get NullPointerException.

Map(I):

- If we want to represent a group of objects as key value pairs. Then we should go for Map.Both key & value are objects.
- Both key & values are objects.
- Duplicate values are not allowed.But values can be duplicated.
- Each key value pair is called Entry.
  Example:

Roll no	Name
101	Durga
102	Sinu
103	Ravi
104	Shambu
105	Sundar

- There is no relationship between collection & map.
- Collection meant for a group of individual objects where as map meant for a group of key value pairs.
- Map is not child interface of collection.



Method of Map Interface:

1) Object put (Object key,object value);

   To add key value pair to the Map.

   If the specified key is already available. The old value will be replaced with new value & old value will be returned.

2) Void putAll(Map m)
   To add a group of key value pairs.
3) Object get(Object key)
   Return the value associated with specified key.
   If the key is not available then we will get null.
4) Object remove(object key)

5) boolean containsKey(object key)
6) boolean containsValue(object value)
7) int size();
8) boolean isEmpty();
9) void clear()
   I. set keyset();
   II. collection value();
   III. set entrySet();

Entry(Interface):

Each key value pair is called one entry.
Without existing Map object there is no chance of entry object. Hence, Interface Entry is define inside Map interfaces.
Example:

```
interface Map
  {
   Interface Entry
     {
     1)object getKey();
     2)object getValue();
     3)object setValue();
  }
}
```

1)HashMap:

- The underlying data structure is HashTable.
- Heterogheneous objects are allowed for both keys & Values.
- Duplicate keys are not allowed but the values can't duplicated.
- Insertion order is not preserved because it is based on hashcode of keys.
- Null key is allowed(only once)
- Null values are allowed(any no. of times).

Difference between HashMap & HashTable:

HashMap	**HashTable**
1)No method is Synchronized.	1)Every method is Synchronized.

2)Multiple Thread can operates simultaneously & Hence HashMap Object is not Tread safe.	2)At a time only one Thread is allowed to operate an HashTable object.Hence it is thread safe.
3)Thread are not required to wait & hence relatively performance is high.	3)It increases waiting time of the thread & hence performance is low.
4)null is allowed for both key value.	4)null is not allowed for both key values otherwise we will get NP Exception.
5)Introduced in 1.2 version& it is non-legacy.	5)Introduced in 1.0 version& it is legacy.

Q)How to get Synchronized version of HashMap?

Bydefault HashMap object is not Synchronized,but we can get Snchronized version by using synchronizedMap() of collection classes.

Map M=Collections.SynchronizedMap(hashMap hm);

Constructor:

1)HashMap M=new HashMap();

Creates an emptyHashMap object with default initial capacity level is '16' & default fillRatio 0.75(75%).

2)HashMap m=new HashMap(int initialCapacity)

3) HashMap m=new HashMap(int initialCapacity,float fillRatio)

4) HashMap m=new HashMap(Map m)

Example:

```
import java.util.*;
    class HashMapDemo
     {
     public static void main(String args[])
     {
```

```
HashMap m=new HashMap();
m.put("chiranjeevi",700);
m.put("Raj",800);
m.put("Viraj",1000);
m.put("Durga",500);
System.out.println(m);

//Viraj=1000;Raj=800; chiranjeevi=700;Durga=700;

Set s=m.entrySet();
System.out.println(s);

//Viraj;Raj; chiranjeevi;Durga;
Collection c=m.values();
System.out.println(c);
// 1000;800;6000;500;
Set s1=m.entrySet();
Iterator its=s1.iterator();
while(its.hasNext())
{
Map.Entry m1=(Map.Entry)its.next();
System.out.println(m1.getKey()+"  "+m1.getValue());
if(m1.getKey().equals("Durga"))
```

```
    m1.setValue(10000);
     }
    System.out.println(m);

  //Durga=10000;Raj=1000;Viraj=800; chiranjeevi=1000;

  }
}
```

2)LinkedHashMap:

It is the child class of HashMap.
It is Exactly same as Hashmap except the following difference.

HashMap	**LinkedHashMap**
1)The underlying data structure is hashTable.	1)The underlying data structure is hashTable+LinkedList.
2)Insertion order is not preserverd.	2)Insertion order is preserverd.
3)Introduced in 1.2 version.	3)Introduced in 1.4 version.
In above program if we are HashMap , The following is the o/p. {chiranjeevi=700,Raj=800,viraj=1000,Durga=500} i.e. insertion order is preserved.	

Note:

The main application area of LinkedHashSet & LinkedHashMap are cache application implementation where duplication is not allowed & insertion order must be preserved.

2)IdentityHashMap:

It is exactly same HashMap except the following difference.

In the case HashMap to identify duplicate keys jvm always uses .quals(); which is mostly meant for content comparision.
If we want to use == operator instead .equals() to identify duplicate keys we have to use IdentifyHashMap(==operator always meant for reference comparision)

Example:

```
HashMap m=new HashMap();

Integer  i1=new Integer(10);

Integer  i1=new Integer(10);

m.put(i1,"pavan");

 m.put(i2,"kalyan");

 System.out.println(m);   {10=kalyan}
```

In the above code i1 & i2 are duplicate keys because i1.equals(i2) returns true.

If we replace HashMap with IdentifyHashMap Then the o/p is

{10=pavan,10=kalyan}

i1 & i2 are not duplicate keys because i1==i2 returns false.

WeakHashMap:

- It is exactly same as HashMap except the following difference.
- In the case of following object is no eligible for g.c. Eventhough it doesn't have any external references if it is associated with HashMap.i.e.HashMap dominates Garbage collection(g.c).
- But in the case of WeakHashmap Eventhough object associated with HashMap is eligible for g.c,If it is doesn't have any external references .i.e. g.c. dominates WeakHashmap.

Example:

```
    import java.util.*;
  class WeakHashMapDemo
   {
   public static void main(String args[])throws interruptedException
   {
   HashMap m=new HashMap();
    Temp t=new Temp();
   m.put(t,"durga");
  System.out.println(m);
   //temp=durga
   t=null;
   System.gc();
   Thread.sleep(5000);
  System.out.println(m);}}
class Temp
{
public String toString()
{
return "temp";
public void finalize()
{
```

```
System.out.println("finalize method called");
}
}
```

O/P-

{temp=durga}

{temp=durga}

If we replace HashMap with WeakHashMap Then the O/P is

{temp=durga}

Finalize method called

{}

SortedMap(I):

- If we want to represent a group of entries according to some sorting order.Then we should go for SortedMap. The sorting should be done based on the keys but not based on the values.
- SortedMap interface is the child interface of map.
- SortedMap interface defines the following 6 specific method.
    1. Object firstKey();
    2. Object lastKey();

3. SortedMap headMap(object key1);
4. SortedMap bailMap(object key1);
5. SortedMap subMap(object key1, object key2);
6. Comparator comparator();

TreeMap(I):

- The underlying data structure is RED-BLACK Tree.
- Insertion order is not preserved & all entries are inserted according to some sorting order of keys.
- If we are depending on default natural sorting order then the keys should be Homgeneous & comparable. Otherwise we will get ClassCastException(CCE).
- If we are defining our own Sorting order by comparator Then the keys need not be Homogeneous & Comparable.
- There are no reinstructions on values ,they can be Homogeneous & Comparable.
- Duplicate keys are not allowed but values can be duplicated.

Null acceptance:

- For the empty TreeMap as the first entry with null key is allowed.but after inserting that entry if we are trying to insert any other entry we will get NullPointerException(NPE).
- For The Non-Empty TreeMap if we are trying to insert entry with null key we will get NullPointerException(NPE).
- There are no re-instruction on null values i.e. ,we can use null any no.of times any where for map values.

Constructors:

1. TreeMap t=new TreeMap()
   For default natural sorting order
2. TreeMap t=new TreeMap(comparator c)
   For customized sorting order
3. TreeMap t=new TreeMap(Map m)
4. TreeMap t=new TreeMap(SortedMap m)

Example1:

```
    import java.util.*;
  class TreeMapDemo
   {
   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(107,null);
      //m.put("ffff","xxx");       //CCE
      //m.put(null,"xxx");        //NPE
      System.out.println(m);
//100=zzz,101=xxx,103=yyy,104=106,107=null
          }
```

```
            }
```

O/P-100=zzz,101=xxx,103=yyy,104=106,107=null

Example2:

```
        import java.util.*;
    class TreeMapDemo
     {
     public static void main(String args[])
     {
     TreeMap t=new TreeMap(new MyComparator());
           t.put("xxx",10);
         t.put("AAA",20);
       t.put("zzz",30);
      t.put("LLL",40);
      System.out.println(t);
      }
     }
      class MyComparator implements Comparator
         {
         public int compare(object obj1,object obj2)
        {
         String s1=obj1.toString();
         String s2=obj2.toString();
```

```
        Return s2.compareTo(s1);
        }
        }
```

O/P - zzz=30,xxx=10,LLL=40,AAA=20

Hashtable(c):

- The underlying data structure is HashTable.
- Heterogeneous objects are allowed for both keys & values
- Insertion order is not preserved & it is based on HashCode of the keys.
- Null is not allowed for both key & values otherwise we will get NullPointerException(NPE).
- Duplicate keys are not allowed but values can be duplicated.
- All method are Synchronized & hence HashTable object is ThreadSafe.

Constructors:

1. HashTable h=new HashTable()
   Creates an empty Hashtable object with default initial capacity is '11' & default fill ratio75%(0.75).
2. Hashtable h=new Hashtable(int initialCapacity)
3. Hashtable h=new Hashtable(int initialCapacity,float fillratio)
4. Hashtable h=new Hashtable(Map m);

Example:

```
import java.util.*;
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("durga",null);    //NPE
System.out.println(h);
}
```

```
}
class Temp
{
int i;
Temp(int i)
{
this.i=i;
 public int hashCode()
{
return i;
}
public String toString()
{
return i+" ";
}
}
```

Properties(c):

- It is the child class of HashTable.
- In our program if any thing which changes frequently (Like database username,passwords,Url)never recommended to hashcode .The value in the java program.because for every change ,we have recompile ,rebuild,redeploy.The application & sometime even server restart .Which creates a big business impact to the client.
- We have to configure those variable inside properties files & we have to read those values from java code.

- The main advantage of this approach is,If any change in the properties file just redeployment is enough which is not business impact to the client.

Constructor:

- Properties p=new properties();
  In the case of properties both key & value should be string type.

Methods:

- String getProperty(String Propertyname)
  Returns the value associated with specified property.
- String setProperty(String Pname,String pvalue);
  To set a new property.
- Enumeration propertyNames();
- void load(InputStream is)
  To load the properties from properties files into java properties object.
- void store(outputStream os,String comment)
  To update properties from properties object into properties files.

Example:

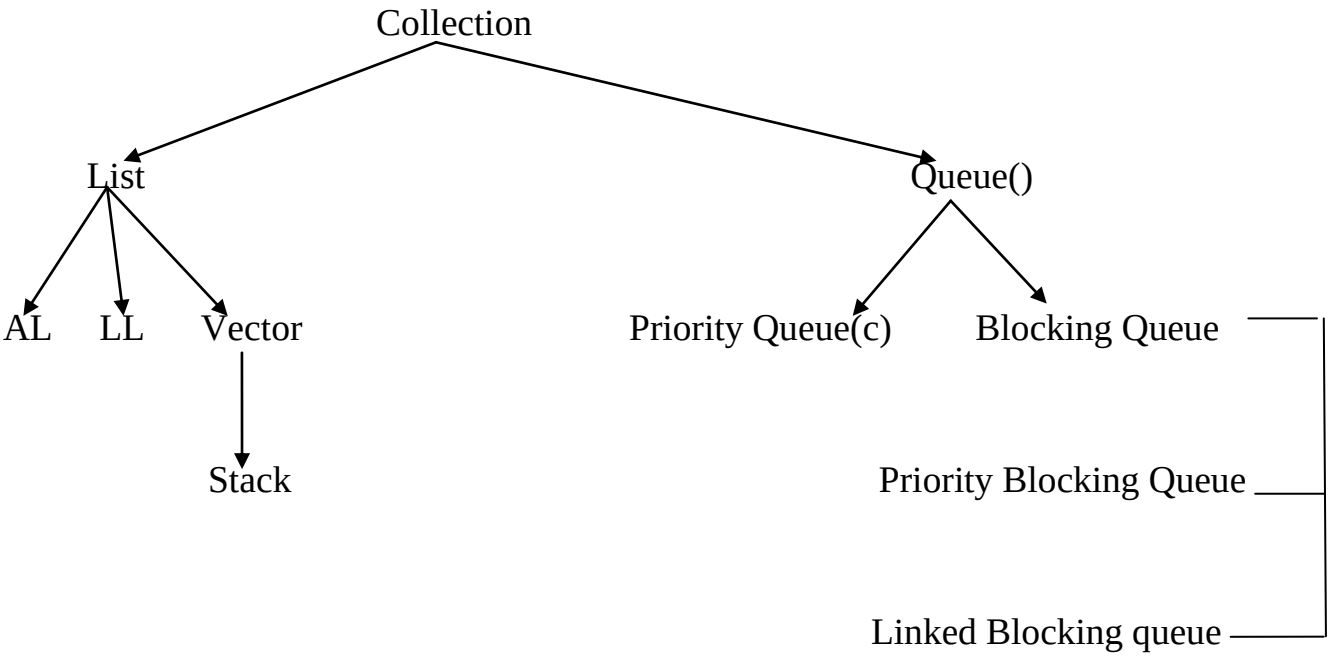
```
import java.util.*;
import java.io.*;
class PropertiesDemo
{
```

```
public static void main(String args[])throws interruptedException
{
Properties p=new Properties();
FileInputStream fis=new FileInputStream("abc.properties");
 p.load(fis);
System.out.println(p);
String s=p.getProperty("durga");
System.out.println(s);
p.setProperty("nag","qqqqq");
FileOutputStream fis=new FileOutputStream("abc.properties");
p.store(fos, "updated by durga for scjp demo class");
 }
 }
```

1.5 Version Enhancement:

Queue(I):

- It is the child interface of collection.
- It we want to Represent a group of individuals objects prior to processing Then we should go for queue.



- Usually queue follows FIFO(first in first out),But Based on our requirement we can change our order.
- From 1.5 version onwords LinkedList implements queue Interface.
- LinkedList based implementation of queue always follows FIFO.

Queue Interface method:

- Boolean offer(object obj)
  To add an object into the queue.
- Object peek();

To return head element of the queue.if queue is empty then this method returns null.

- Object element();
  To return head element of the queue.if queue is empty then we will get Runtime Exception saying NoSuchElementException.
- Object poll();
  To remove & return head element of the queue.If the queue is empty then this method returns null.
- Object remove();
  To remove & return head element of the queue.If the queue is empty then we will get Runtime Exception saying NoSuchElementException.

For the empty TreeMap as the first entry with null key is allowed.but after inserting that entry if we are trying to insert any other entry we will get NullPointerException(NPE).

Priority Queue :

- This is the data structure to hold a group of individual objects prior to processing according to some priority.

- The priority can be either default natural sorting order or customized sorting order.
- If we are depending on default natural sorting compulsory object should be homogeneous & comparable otherwise we will get ClassCastException.
- If we are defining our own customized sorting by comparator then the object need not be homogeneous & comparable.
- Duplicate objects are not allowed .
- Insertion order is not preserved.
- Null insertion is not possible even as 1st element also.

Constructor:

- PriorityQueue q=new PriorityQueue();
  Creates an empty priority queue with default iniyial capacity & priority order is default natural sorting order.
- PriorityQueue q=new PriorityQueue(int initialcapacity);
- PriorityQueue q=new PriorityQueue(int initialcapacity,comparator);
- PriorityQueue q=new PriorityQueue(Collection c);
- PriorityQueue q=new PriorityQueue(SortedSet s);

Example 1:

```
import java.util.*;

class PriorityQueueDemo
```

```
{
public static void main(String args[])
{
PriorityQueue q=new PriorityQueue();
System.out.println(q.peek());            //null
System.out.println(q.element());      // NoSuchElementException.
for(int i=0;i<=10;i++){
q.offer(i);
}
System.out.println(q);               //[0,1,2,….10]
System.out.println(q.poll());           //0
System.out.println(q);                //[1,2,3,….10]
}
}
```

Example 2:

```
import java.util.*;
class PriorityQueueDemo
{
public static void main(String args[])
{
PriorityQueue q=new PriorityQueue(15,new MyComparator());
q.offer("A");
```

```
q.offer("Z");
q.offer("L");
q.offer("B");
System.out.println(q);          //[Z,L,B,A]
}
}
```

```
class MyComparator implements Comparator
{
public int compare(Object obj1.Object2)
{
String s1=(String)obj1;
String s2=obj2.toString();
return s2.compareTo(s1);
}
}
```

O/P-Z,L,B,A

1.6 Version Enhancement :

1.NavigableSet(I):

- It is the child interface of SortedSet.
- This interface defines several method to provide support for navigation for the tree set object.
- The following list of various method presenting in NavigableSet.

1.ceiling(e):
Returns the lowest element which is >=e
2.higher(e):
Returns the lowest element which is >e
3.floor(e):
Returns the highest element which is <=e
4.lower(e):
Returns the highest element which is <e
5.pollFirst:
Remove & Returns the first element
6.pollLast:
Remove & Returns the last element
7.desendingSet:
Returns the NavigableSet in reverse order.

Example 2:

```
import java.util.*;
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.println(t);
System.out.println(t.ceiling(2000));   //2000
System.out.println(t.higher(2000));   //3000
System.out.println(t.floor(3000));    //3000
System.out.println(t.lower(3000));    //2000
System.out.println(t.pollFirst());      //1000
System.out.println(t.pollLast());      //5000
System.out.println(t.desendingSet());      [4000,3000,2000]
System.out.println(t);                          [2000,3000,4000]
}
}
```

2.NavigableMap(I):

- It is the child interface of SortedMap to defines several method for navigation purpose.
- The following list of various method presenting in NavigableMap.
  1.ceilingKey(e):
  2.higherKey(e):
  3.floorKey (e):
  4.lowerKey (e):
  5.pollFirstEntry:
  6.pollLastEntry:
  7.desendingMap:

Example 1:

```
import java.util.*;
class NavigableMapDemo
{
public static void main(String args[])
{
TreeMap <String,String> t=new TreeMap <String,String>();
t.put( "b","banana");
t.put( "c","cat");
t.put( "a","apple");
t.put( "d","dog");
t.put( "g","gun");
System.out.println(t);          //{a=apple,b=banana,c=cat,d=dog,g=gun}
System.out.println(t.ceilingKey("c"));   //c
System.out.println(t.higherKey("e"));   //g
```

```
System.out.println(t.floor Key("e"));    //d
System.out.println(t.lower Key("e"));    //d
System.out.println(t.pollFirstEntry());      //a=apple
System.out.println(t.pollLastEntry());      //g=gun
System.out.println(t.desendingmap());      {d=dog,c=cat,b=banana}
System.out.println(t);                          {b=banana,c=cat,d=dog }
}
}
```

Collections class:

Collections class:

- It is an utility class present in java.util package.
- It defines several utility methods for collection implement class objects.

Sorting the element of a list:

- Collection class defines the following method to sort elements of a List.
  1. public static void sort(List l):
     We can use these method to sort according to natural sorting order.
     Be homogeneous & comparable.Otherwise we will get ClassCaseException.
     List should not contain null ,otherwise we will get NullPointerException.

  2. public static void sort(List l,Comparator c):
     To sort elements of a list according to customized sorting order.

Searching the element of a list:

- Collection class defines the following method to search elements of a List.
  1 public static int binarySearch(List l,Object obj):
  If the list is sorted according to natural sorting order then we have to use this method.
  2 public static int binarySearch(List l,Object key,comparator c):
  If the List is sorted according to comparator then we have use this method.

Conclusion:

- Internally binary search method uses BinarySearch algorithm.
- Before calling binarySearch() method compulsory the list should be sorted.otherwise we will get unpredictable results.
- Successful search returns index.
- Unsuccesfull search returns insertion point.
- Insertion point is the location where we can place element in the sortedList.

- If the List is sorted according to comparator then at the time of search also we should pass the same comparator otherwise we will get unpredictable result.

Example 1

```
import java.util.*;
class CollectionDemo
 {
 public static void main(String args[])
 {
 ArrayList l=new ArrayList();
l.add( "Z");
l.add( "A");
l.add( "M");
l.add( "K");
l.add( "a");
System.out.println(l);          //[Z,K,M,Z,a]
Collections.sort(l);
System.out.println(l);
System.out.println(Collections.binarySearch(l,"Z"));   //3
System.out.println(Collections.binarySearch(l,"j"));   //-2
}
}
```

Example 2:

```
import java.util.*;
class CollectionSearchDemo
 {
 public static void main(String args[])
 {
 ArrayList l=new ArrayList();
l.add( 15);
l.add( 0);
l.add( 20);
l.add( 10);
l.add( 5);
System.out.println(l);          //15   0   20   10   5
Collections.sort(l);
System.out.println(l,new MyComparator());
System.out.println(l);           //20    15    10    5     0
System.out.println(Collections.binarySearch(1,10,new MyComparator());   //2
System.out.println(Collections.binarySearch(1,13,new MyComparator());   //-3
System.out.println(Collections.binarySearch(1,17));     //-6
}
}
```

```
class MyComparator implements Comparator
{
public int compare(Object obj1.Object2)
{
Integer i1=(Integer)obj1;
Integer i2=(Integer)obj;
return i2.compareTo(i1);
}
}
```

Note:

For the List contains n element range of successful search

- Range of successful search:0 to n-1
- Range of unsuccessfull search: -(n+1)to-1
- Total range : -(n+1)ton-1

Reversing the elements of a List:

Collection class defines the following reverse method for this

public static void reverse(List l);

Example 1: To Reverse Element of List

```
import java.util.*;
class CollectionReverseDemo
 {
 public static void main(String args[])
 {
 ArrayList l=new ArrayList();
l.add( 15);
l.add( 0);
l.add( 20);
l.add( 10);
l.add( 5);
System.out.println(l);          //15   0   20   10   5
Collections.reverse(l);
System.out.println(l);           //20    15    10    5     0
}
}
```

reverse() Vs reverse Order():

- We can use reverse() method to reverse the element of a list & this method contain list argument.
- Collection class defines reverse order method also to return comparator object for reversing original sorting order.
  Comparator c1=Collections.reverseOrder(Comparator c)

Desending order asending order

- Reverse Order() method contain comparator argument where as reverse contain list arguments.

\ Example 1: To Reverse Element of List

```
import java.util.*;
class CollectionReverseDemo
 {
 public static void main(String args[])
 {
 ArrayList l=new ArrayList();
l.add( 15);
l.add( 0);
l.add( 20);
l.add( 10);
l.add( 5);
System.out.println(l);          //15   0   20   10   5
```

```
Collections.reverse(l);
System.out.println(l);          //20    15    10    5     0
}
}
```

Arrays Class

Arrays class:

It is an utility class present in util package,To define several utility methods for arrays for both primitive arrays and object type arrays.

Sorting the Elements of Arrays:

Arrays class define the following method for this.

1)public static void Sort(primitive[] p);

-To sort elements of primitive Arrays according to natural sorting order.

2)public static void Sort(Object[] a);

-To sort elements of object Arrays according to natural sorting order.

-In this case compulsary the elements should be Homogeneous and Comparable otherwise we will get ClassCastException.

3)public static void Sort(Object[] a,Comparator c);

-To sort elements of object[] according to cutomized sorting order.

Note:-

Primitive arrays can be sorted only by natural sorting order where as object arrays can be sorted either by natural sorting order by customized Sorting order.

Example 1:

To sort Elements of Arrays

```
import java.util.Arrya;
import java.util.Comparator;
class ArraysSortDemo
{
```

```
public static void main(String[] args)
{
int[] a={10,5,20,11,6};
System.out.println("Primitive arrays before sorting:");
for(int a1=a)
{
System.out.println(a1);
}
Arrays.Sort(a);
System.out.println("Primitive arrays after sorting:");
for(int a1=a)
{
System.out.println(a1);
}
String[] s={"A", "Z", "B"};
System.out.println("Object arrays before sorting:");
for(String a2=s)
{
System.out.println(a2);
}
Arrays.Sort(s);
System.out.println("Object arrays after sorting:");
for(String a1=s)
{
System.out.println(a1);
}
Arrays.Sort(s,new MyComparator);
System.out.println("Object arrays after sorting by Comparator:");
for(String a1=s)
{
System.out.println(a1);
}
}
}
```

```
class MyComparator implements Comparator
{
public int Compare(Object o1,Object o2)
{
String s1=a1.toString();
String s2=a2.toString();
return s2.CompareTo(s1);
}
}
```

Searching the elements of Array:-

-Arrays class defines the following search methods for this.

1)public static int binarySearch(primitive() p,Primitive Key)

2)public static int binarySearch(Obiject() o,Object Key)

3)public static int binarySearch(Obiject() o,Object Key Comparator c)

Note:-

-All rules of these binarySearch() method are exactly same as collections class binarySearch() method.

Example 1:

```
import java.util.*;
import static java.util.Arryas.*;
class ArraysSearchDemo
{
public static void main(String[] args)
{
int[] a={10,5,20,11,6};
Arrays.Sort(a);  //Sorted by natural order.[5,6,10,11,20]
System.out.println(Arrays.binarySerach(a,6));    //1
System.out.println(Arrays.binarySerach(a,14));   //-5
String[] s={“A”, “Z”, “B”};
```

```
Arrays.Sort(s);
System.out.println(binarySerach(s, "Z"));    //2
System.out.println(binarySerach(s, "S"));   //-3
Arrays.Sort(s,new MyComparator());
Arrays.Sort(s, new MyComparator());
System.out.println(binarySerach(s, "Z", new MyComparator()));    //0
System.out.println(binarySerach(s, "S", new MyComparator()));   //-2
System.out.println(binarySerach(s, "N"));   //Unpredictable result
}
}
class MyComparator implements Comparator
{
public int Compare(Object o1,Object o2)
{
String s1=a1.toString();
String s2=a2.toString();
return s2.CompareTo(s1);
}
}
```

Converting Arrays to List:-

- public static List asList(Object[] a)
- By using this method we are not creating an independent List Object just we are creating List view for the existing Array object.
- By using List reference if we perform any operation.The changes will be reflected to the Array Reference.Similarlly,By using Arrays reference if we perform any changes those changes will be reflect to the List .
- By using List reference we cannot perform any operation which varies the size(i-e add or remove)otherwise we will get RunTimeException saying "Unsupported-Operation-Exception".
- By using List reference we can perform replacement operation.But replacement should be with the same type of element only otherwise we will get RunTimeException saying "ArraysStoreException".

Example 1:

To view Array in List Form

```
import java.util.*;
class ArrayAsListDemo
{
public static void main(String[] args)
{
String[] s={“A”, “Z”, “B”};
List l=Arrays.asList(s);
System.out.println(l);    //[A,Z,B]
S[0]= “K”;                 //[K,Z,B]
System.out.println(l);    //[K,Z,B]
l.set(l, “L”);              //[K,L,B]
for(String s1:s)
System.out.println(s1);    //[K,L,B]
l.add(“durga”);            // Unsupported-Operation-Exception
l.remove(l);               // Unsupported-Operation-Exception
l.set(l, “S”);             //[K,S,B]
l.set(1,10);               // ArraysStoreException
}
}
```