Collections(Main Part - 6)

Collections

int x=10; int y=20; int z=30;
In this approach, if i want to keep 10000 values then we can't remember variables to access them. To resolve this problem we use arrays.

Arrays

It refers to indexed collection of homogenous data elements.

Advantage of Arrays

1. we can represent multiple values by using single varaible, so that readability of the code will be improved.
   eg::
   int arr[] =new int[1000];
   we resolved the problem, but array is having limitation.
   Student[] s=new Student[100];
   s[0] =new Student();
   s[1] =new Employee(); **//incompatible type: found Employee required:Student**

To resolve this problem we can use
Object[] obj =new Object[1000];
obj[0]=new Student();
obj[1]=new Employee();

Limitations of Arrays

1. Array is fixed in size, we can't increase or decrease the size of array.
2. To use the array compulsorily we should know the array size at the begining itself.
3. Array can hold only homogeneous datatype elements.
4. Array is not implemented using standard datastructure,so we don't have ready made methods to perform our task.

eg: based on some condition, if we want to sort the student object in student[] direct methods are not available so it increases complexity of programmer. To Overcome the limitations of Arrays we use "Collections".

Collections

1. They are growable in nature(we can increase and decrease)
2. They can hold both hetereogenous and homogenous data elements.
3. Every collection class is implemented using some standard datastructure, so ready methods are available, as a programmer we need to implement rather we should just know how to call those methods.

Which one is prefered over Arrays and Collections?

Arrays is prefered, because performance is good.

Collections 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:

=> To get something we need to compromise something, so if we use Collections performance is not upto the mark.

=> Array is language level concept(memory wise it is not good, perfomance is high)

=> Collection is API level(**memory wise it is good,perfomance is low**)

Difference b/w Arrays and Collection

Arrays =>

1)- It is used only when Array size is fixed

2)- memory wise not recomended to use.

3)- Performance wise recomended to use.

4)- It can hold only homogenous objects

5)- We can hold both primitive values and Objects

eg: int[] arr=new int[5];

Integer[] arr=new Integer[5];

6)- It is not implements using any standard datastructure, so no ready made methods for our requirement,it increases the complexity of programming

Collection =>

1)- It is used only when size is not fixed(dynamic)

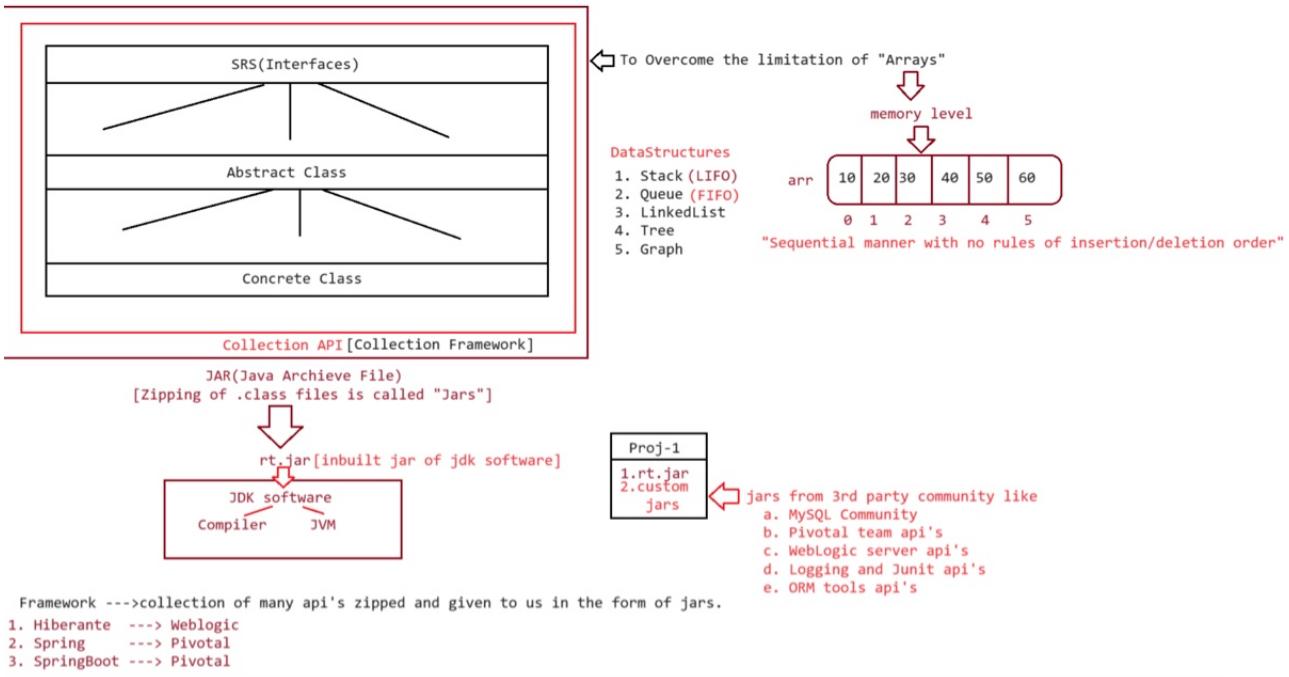
2)- memory wise recomended to use.

3)- Perforamance wise it is not recomended to use.

4)- It can hold both heterogenous and homogenous Objects

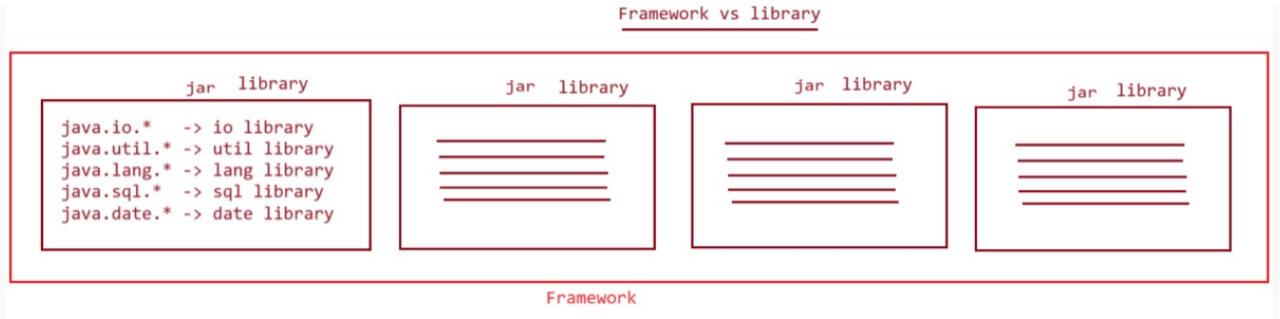
5)- It is capable of holding only objects not primitive types.

6)- It is implemented using standard datastructure, so ready made methods are available for our requirement,it is not complex.



For arrays we work with direct memory level hence performance is good while in collections there are rules(LIFO,FIFO, etc) and methods which decreases the complexity of program.

Framework vs library



Limitations of Arrays

1. fixed size, can't increase nor decrease.
2. it can't hold hetergenous elements.
3. it doesn't follow any datastructure, so ready made methods are not available to perform specific task.

Solution : Go for Collections

Benefits of Collections

1. It is dynamic in nature(can grow in size)
2. it can hold both homogenous and heteregenous objects.
3. Every collection object is implemented using standard datastructure, so ready made support of methods are avaiable so progrmaming is easy.

Which one is good either collections or Arrays?

Ans.

Performance :: Arrays are recomended.

:: Collections are not recomended.

memory :: Arrays are not recomended.

:: Collections are recomended.

Difference b/w Collection vs Arrays

Arrays -> it can hold both primtive type data and object type data.

Collection -> it can hold only Objects type of data.

What is Collection?

In Order to represent a group of individual object as a single entity then we need to go for Collection.

CollectionFramework

Group of classes and interface, which can be used to represent group of individual object as a single entity, then we need to go for **"CollectionFramework".**

Java **C++**

Collection => container

CollectionFramework => STL(standard template library)

To know more information about the framework,then we need to know the specification(interface)

9 key interfaces of Collection framework

a. Collection(I)

b. List(I)

c. Set(I)

d. SortedSet(I)

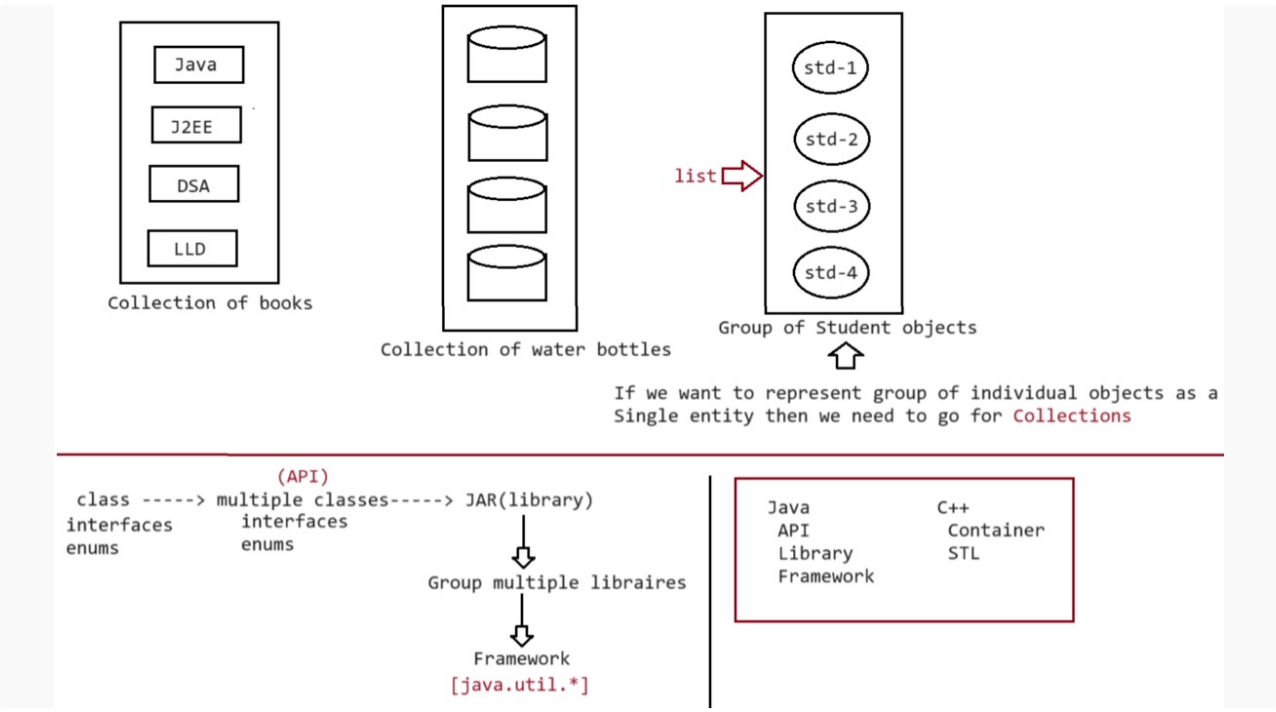
e. NavigableSet(I)

f. Queue(I)

g. Map(I)

h. SortedMap(I)

i. NavigableMap(I)



Collection

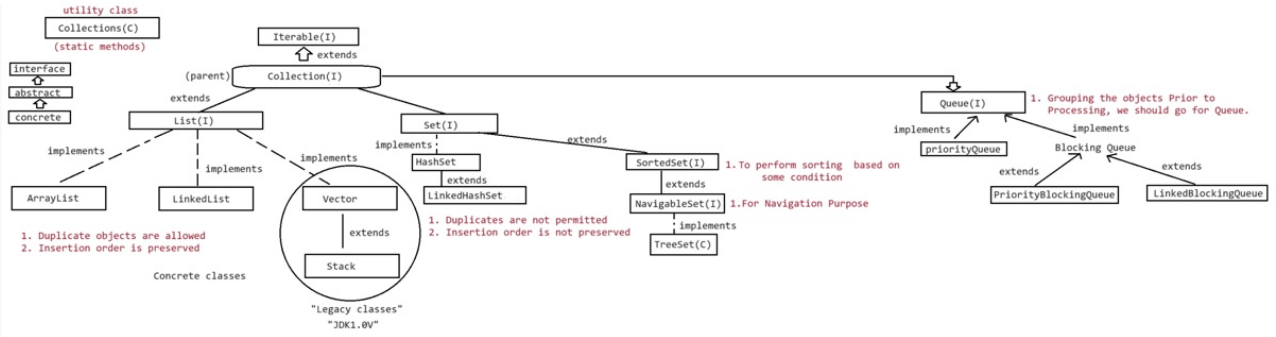
1. In order to represent a group of individual object, then we need to go for "Collection".
2. It is a root interface of collection framework
3. All the commonly used method required for all the collection is a part of Collection(I).

Note: There is no concrete class which would implement the interface Collection(I) directly.

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)



List(I)

1. Insertion order must be preserved.
2. Duplicates are allowed.
3. It is the child interface of Collection.

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

Set(I)

It is used to represent a group of individual objects as a single entity such that

1. Duplicates are not allowed
2. Insertion is not preserved
3. It is the child interface of "Collection", then we need to go for "Set".

SortedSet(I)

It is used to represent a group of individual objects as a single entity such that

1. Duplicates are not allowed
2. elements should be added based on some sorting order
3. It is the child interface of "Set". then we need to go for "SortedSet".

NavigableSet(I)

1. It is a child class of SortedSet.
2. Various methods are a part of NavigableSet for navigation purpose.
3. Implementation class for NavigableSet is TreeSet.

What is the difference b/w List and Set?

List => Duplication are allowed, insertion order is preserved.

Set => Duplication are not allowed, insertion order not preserved.

Queue (I):

=> 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.

Eg: Before sending a Mail we have to Store All MailID's in Some Data Structure and in which Order we added MailID's in the Same Order

Only Mails should be delivered (FIFO). For this Requirement Queue is Best Suitable.

Map(I)

a. To represent a group of individual objects as keyvalue pair then we need to opt for Map(I).

8. **SortedMap (I):**

   => It is the Child Interface of Map.

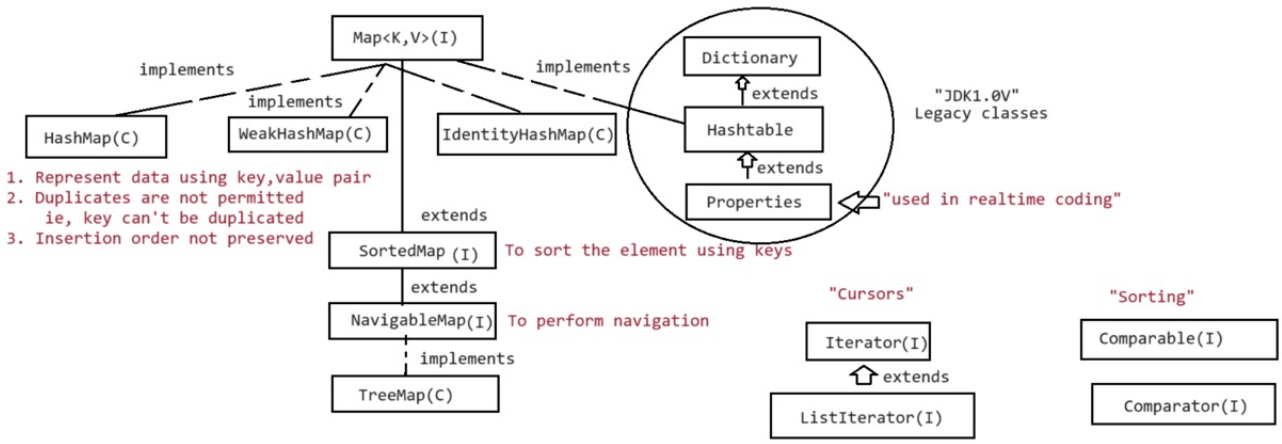
   => If we want to Represent a Group of Objects as Key- Value Pairs according to Some Sorting Order of Keys then we should go for SortedMap.

   => Sorting should be Based on Key but Not Based on Value

9. **NavigableMap (I):**

   => It is the Child Interface of SortedMap.

   => It Defines Several Methods for Navigation Purposes.



Legacy Characters

1. Enumeration
2. Dictionary
3. Vector
4. Stack
5. Hashtable
6. Properties

Utility classes

1. Collections
2. Arrays

Cursors[Used for traversing/iterating the objects in collection]

1. Enumeration(I)
2. Iterator(I)
3. ListIterator(I)

Sorting

1. Comparable(I)
2. Comparator(I)

Collection(I)

1. Inside this interface, the commonly used method required for all the collection classes is present

a. boolean add(object o)

Only one object

b. boolean addAll(Collection c)

To add group of Object

c. boolean remove(Object o)

To remove particular object

d. boolean removeAll(Collection c)

To remove particular group of collection

e. void clear()

To remove all the object

f. int size()

To check the size of the array

g. boolean retainAll(Collection c)

except this group of objects remaining all objects should be removed.

h. boolean contains(Object o)

To check whether a particular object exists or not

i. boolean containsAll(Collection c)

To check whether a particular Collection exists or not

j. boolean isEmpty()

To check whether the Collection is empty or not

k. Object[] toArray()

Convert the object into Array.

l. Iterator iterator()

cursor need to iterate the collection object

Note : There is no concrete class which implements Collection interface directly.

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

Why add() is different in Object and List(and also some other methods)??

Ans.

- `Collection`'s `add` method adds the element to an unspecified location in the collection.
- `List` inherits this behavior and adds another `add` method with two parameters to insert the element at a specific index.

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()

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)

It is used to create an equivalent ArrayList Object based on the Collection Object

eg#1.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        //List : index plays a role[internally Array]
        ArrayList l = new ArrayList();
        System.out.println("The size of list is :: "+l.size());
        l.add("A");
        l.add(10);
        l.add("A");
        l.add(null);
```

```
        System.out.println("The size of list is :: "+l.size());
        l.remove(2);
        System.out.println("The size of list is :: "+l.size());
        System.out.println(l);
        l.add(2,"sachin");
        System.out.println(l);
    }
}
```

Output

The size of list is :: 0

The size of list is :: 4

The size of list is :: 3

[A, 10, null]

[A, 10, sachin, null]

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.

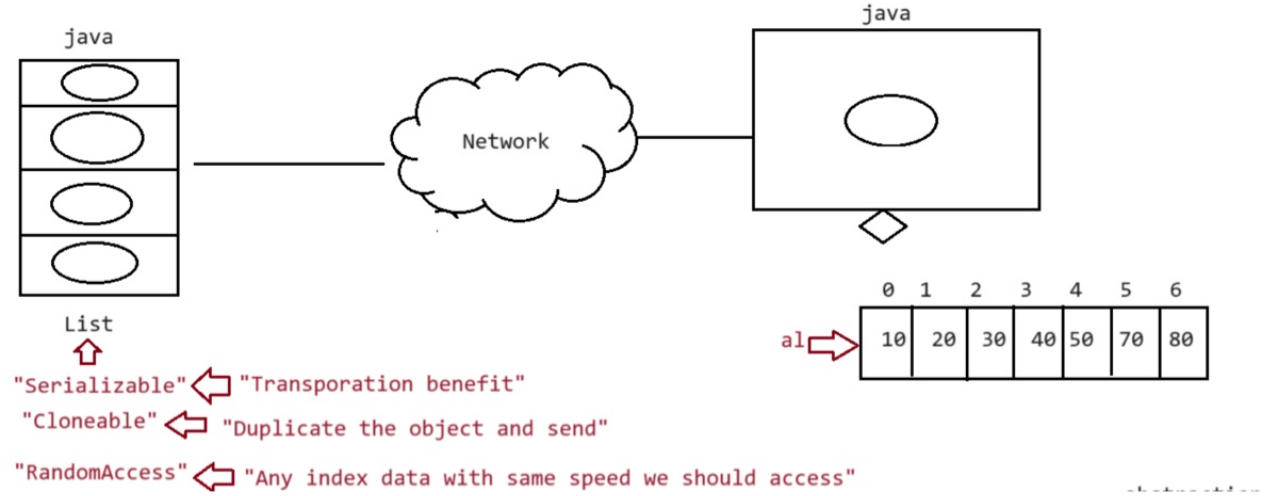
o/p => {k1=v1,k2=v2,k3=v3,....}

Usally we use Collection to store mulitple objects into single entity.

Collection => contanier

To transport the collection over the network,compulsorily the Object should be "Serializable".

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

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.

Vector => Most of the methods are synchronized.

ArrayList =>It is not thread safe becoz mulitple threads can operate on a object.

Vector =>It is thread safe becoz only one thread is allowed to operate.

ArrayList => performance is high becoz threads are not allowed to wait.

Vector => performance is relatively low becoz thread are required to wait.

ArrayList => It is not a legacy class.

Vector => 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);
```

eg#1.

```
import java.util.*;
import java.io.*;
public class Test
{
    public static void main(String[] args)
    {
        //List : index plays a role[internally Array]
        ArrayList al = new ArrayList();
        LinkedList ll = new LinkedList();
        System.out.println("Arraylist implements Serializable :: "+(al instanceof Serializable));
        System.out.println("Linkedlist implements Serializable ::"+(ll instanceof Serializable));

        System.out.println();

        System.out.println("Arraylist implements Cloneable :: "+(al instanceof Cloneable));
        System.out.println("Linkedlist implements Serializable ::"+(ll instanceof Cloneable));

        System.out.println();

        System.out.println("Arraylist implements RandomAccess :: "+(al instanceof RandomAccess));
        System.out.println("Linkedlist implements RandomeAccess ::"+(ll instanceof RandomAccess));
    }
}
```

Output

Arraylist implements Serializable :: true

Linkedlist implements Serializable :: true

Arraylist implements Cloneable :: true

Linkedlist implements Serializable :: true

Arraylist implements RandomAccess :: true

Linkedlist implements RandomeAccess:: false

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".

   LinkedList l=new LinkedList();

   l.add(a);

   l.add(10);

   l.add(z);

   l.add(2,'a');

   l.remove(3);

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);

eg#1.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        //Underlying datastructure :: doubly linked list
        LinkedList ll = new LinkedList();
        ll.add("pwskills");
        ll.add(30);
        ll.add("pwskills");
        ll.add(null);
        System.out.println(ll);
        System.out.println();
        ll.add(0,"nitin");
        System.out.println(ll);
        ll.set(0,"naveen");
        System.out.println(ll);
        ll.addFirst("dhoni");
        System.out.println(ll);
        ll.addLast("kohli");
        System.out.println(ll);
    }
}
```

Output

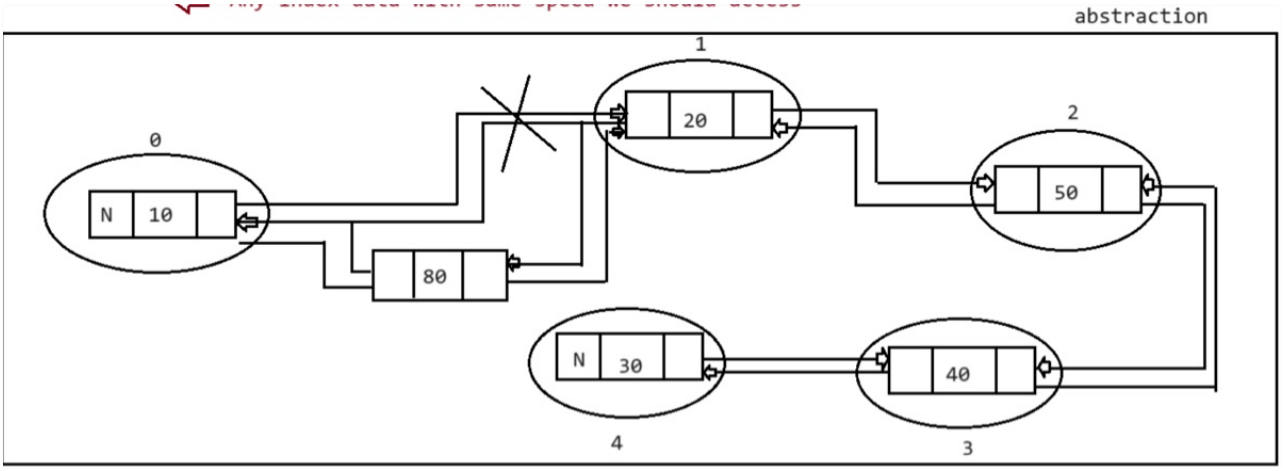
[pwskills, 30, pwskills, null]

[nitin, pwskills, 30, pwskills, null]

[naveen, pwskills, 30, pwskills, null]

[dhoni, naveen, pwskills, 30, pwskills, null]

[dhoni, naveen, pwskills, 30, pwskills, null, kohli]



Above image is an example of abstraction because while using linked list we are not doing all the above operations done to form the doubly lindked list in the image.

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:**
   add(Object o) -> Collection
   add(int index, Object o) -> List
   addElement(Object o) -> Vector
2. **To Remove Elements:**
   remove(Object o) -> Collection

removeElement(Object o) -> Vector

remove(int index) -> List

removeElementAt(int index) -> Vector

clear() -> Collection

removeAllElements() -> Vector

3. **To Retrive Elements:**

   Object get(int index) -> List

   Object elementAt(int index) -> Vector

   Object firstElement() -> Vector

   Object lastElement() -> Vector

4. **Some Other Methods:**

   int size()

   int capacity()

   Enumeration element()

eg#1.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        //Underlying datastructure :: Growable Array
        Vector v = new Vector();
        System.out.println(v.capacity());//10
        for (int i =1;i<=10 ;i++ )
        {
            v.addElement(i);
        }
        System.out.println(v);
        System.out.println(v.capacity());
        v.addElement("sachin");
        v.addElement(null);
        System.out.println(v.capacity());
        v.removeElementAt(3);
        System.out.println(v);
        System.out.println();
        //Working with Cursor
        Enumeration e= v.elements();
        while (e.hasMoreElements())
```

```
        {
            Object o =e.nextElement();
            System.out.println(o);
        }
    }
}
```

Output

10

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

10

20

[1, 2, 3, 5, 6, 7, 8, 9, 10, sachin, null]

1

2

3

5

6

7

8

9

10

sachin

null

interfaces

1. Collection
2. List
3. Set

Implementation :: HashSet,LinkedHaset

4. SortedSet
5. NavigableSet

Implementation :: TreeSet

When to use Set(I) implementation class Objects?

Ans. Duplicates are not allowed

Insertion Order is not preserved

Note: To iterate any type of Collection object,the Collection Object should be "Iterable". For Collection Interface the Parent interface is "Iterable".

Note: Map(I) is not a part of Collection.

6. Map(<K,V>)

Implementation :: HashMap,WeakHashMap,IdentityHashMap
Dictionary,Hashtable,Properties

7. SortedMap
8. NavigableMap

Implementation :: TreeMap

9. Queue

Implementation class : BlockingQueue,PriorityBlockingQueue,...

When to use Queue(I) implementation class Objects?

Ans. If we want to represent group of individual objects prior to processing then we should go for Queue.

Name the cursors available for iterating the Collection Objects

a. Enumeration(I)
b. Iterator(I)
c. ListIterator(I)

Name the interfaces available for Sorting the Objects

a. Comparable(I)
b. Comparator(I)

List(I)

=> Duplicates are allowed

=> Insertion Order is preserved

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

Methods available to immitate Stack and Queue

a. addFirst(Obj)
b. addLast(Obj)
c. getFirst()
d. getLast()

e. removeFirst()

f. removeLast()

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);
}
```

eg#1.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        Stack s =new Stack();
        System.out.println(s.empty());
        s.push("A");
        s.push("B");
        s.push("C");
        System.out.println(s);//[A,B,C]
        System.out.println("Pop element is :: "+s.pop());
        System.out.println(s.empty());
        System.out.println(s.search("Z"));//-1
        System.out.println(s.search("A"));//3
    }
}
```

Output

true

[A, B, C]

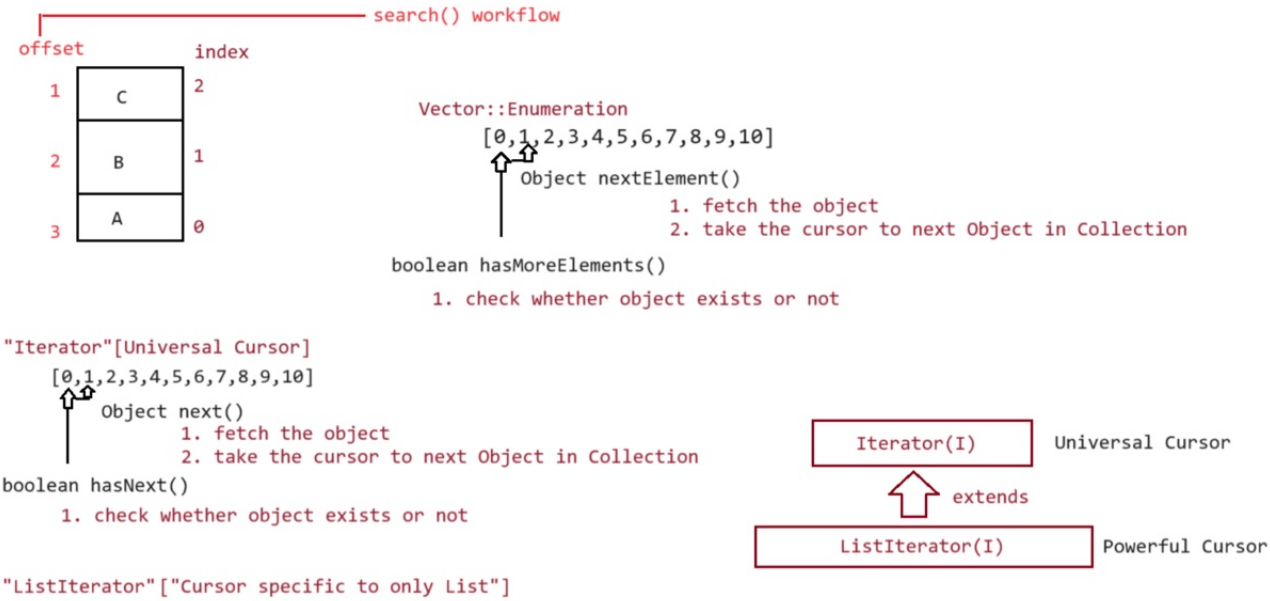
Pop element is :: C

false

-1

2

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();

eg#1.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        //Legacy Object
        Vector v = new Vector();
        for (int i =0;i<=10 ;i++ )
        {
            v.add(i);
        }
        System.out.println(v);//[0,1,2,3,4,5,6,7,8,9,10]
        System.out.println("Iterating through Cursors");
        //Legacy Cursor
        Enumeration e = v.elements();
        while (e.hasMoreElements())
        {
            Integer obj =(Integer)(e.nextElement());
            if (obj%2==0)
                System.out.println(obj);//[0,2,4,6,8,10]
        }
        System.out.println(v);//[0,1,2,3,4,5,6,7,8,9,10]
    }
}
```

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();
```

Eg: Iterator itr = c.iterator(); //c Means any Collection Object.

Methods:

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

eg#2.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        for (int i =0;i<=10 ;i++ )
        {
            al.add(i);
        }
        System.out.println(al);//[0,1,2,3,4,5,6,7,8,9,10]
        System.out.println();
        //Getting iterator obj
        Iterator itr = al.iterator();
        while (itr.hasNext())
        {
            Integer i = (Integer)(itr.next());
            if (i%2==0)
            {
                //Perform read operation
                System.out.println(i);//[0,2,4,6,8,10]
            }
            else
            {
                //Perform delete operation
                itr.remove();
            }
        }
        System.out.println(al);//[0,2,4,6,8,10]
    }
}
```

Output

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

0

2

4

6

8

10

[0, 2, 4, 6, 8, 10]

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 butNotBiDirection.

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

To Overcome these Limitations we should go for ListIterator.

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();

Eg: ListIteratorltr = l.listIterator(); //l is Any List Object

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.

public boolean hasNext()

public Object next()

public int nextIndex()

public boolean hasPrevious()

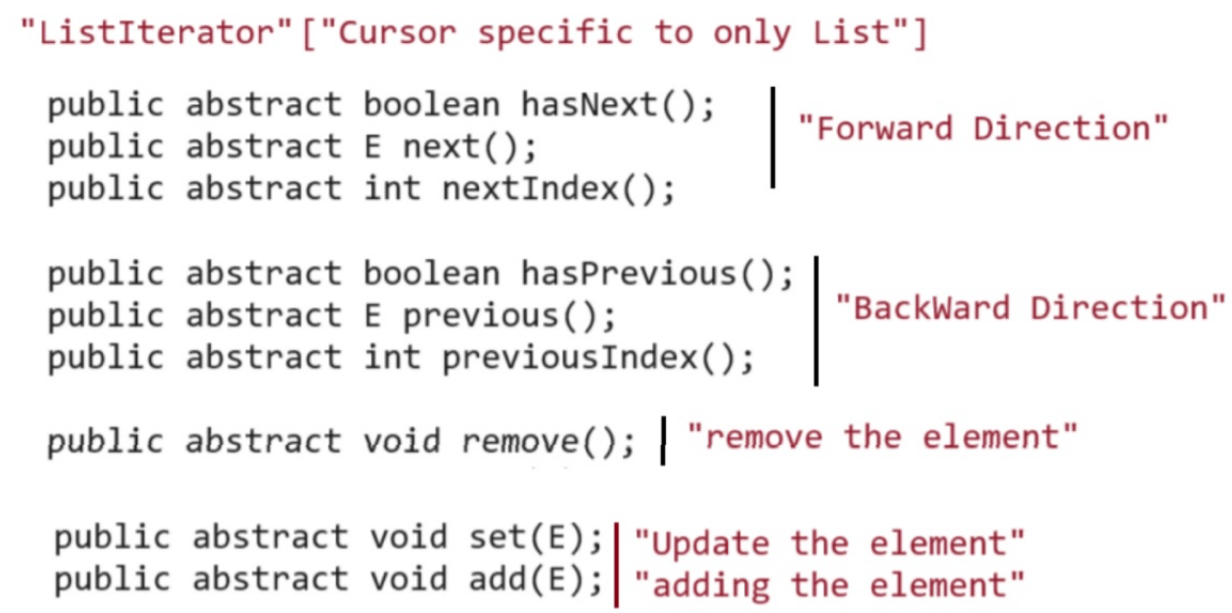
public Object previous()

public int previousIndex()

public void remove()

public void set(Object new)

public void add(Object new)



eg#1.

```
import java.util.*;
public class Test
{
    public static void main(String[] args)
    {
        LinkedList ll = new LinkedList();
        ll.add("sachin");
        ll.add("saurav");
        ll.add("dhoni");
        ll.add("kohli");
        ll.add("dravid");
        System.out.println(ll);//["sachin","saurav","dhoni","kohli","dravid"]
```

```
        //Getting ListIterator:: Powerful cursor
        ListIterator litr = ll.listIterator();
        while(litr.hasNext())
        {
            //Performing read operation
            String name = (String)(litr.next());
            if (name.equals("dravid"))
            {
                //peforming remove operation
                litr.remove();
            }
            if (name.equals("sachin"))
            {
                //performing update operation
                litr.set("sachintendulkar");
            }
            if (name.equals("dhoni"))
            {
                //performing add operation
                litr.add("sakshi");
            }
        }
        System.out.println(ll);
        //[sachintendulkar, saurav, dhoni, sakshi, kohli]
    }
}
```

Output

[sachin, saurav, dhoni, kohli, dravid]

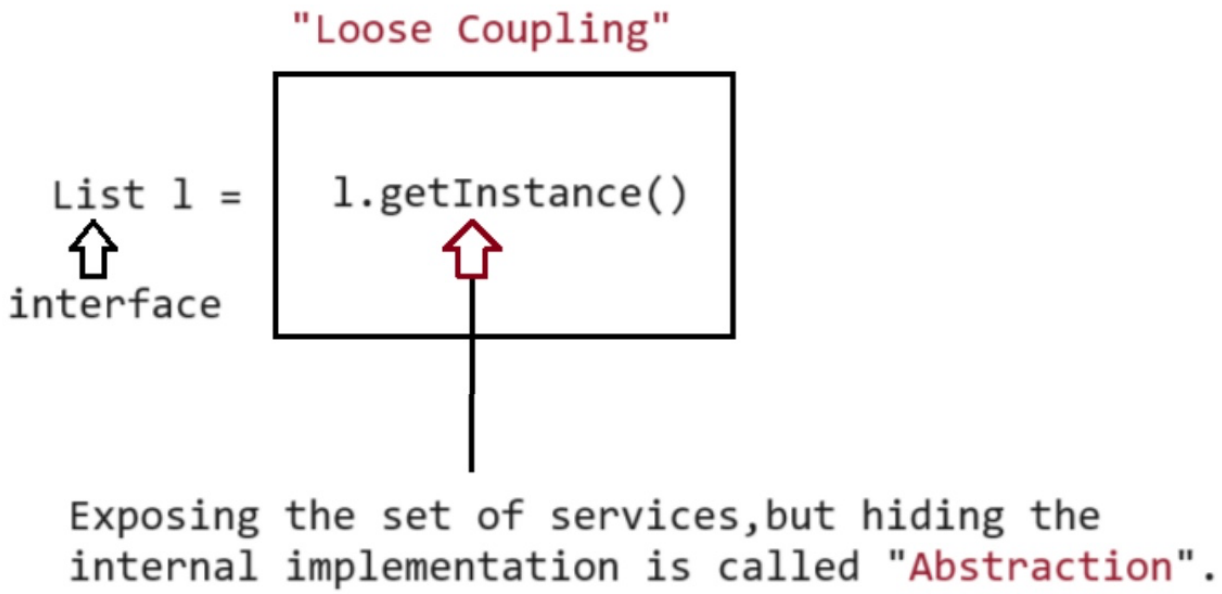
[sachintendulkar, saurav, dhoni, sakshi, kohli]

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.

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

```
        LinkedList ll = new LinkedList();
        ll.add("sachin");
        ll.add("saurav");
        ll.add("dhoni");
        ll.add("kohli");
        ll.add("dravid");
        System.out.println(ll);//["sachin","saurav","dhoni","kohli","dravid"]
        //foreach loop to iterate the Object
        for(Object name : ll)
        {
            System.out.println(name);//sachin saurav dhoni kohli dravid
        }
        //Iterators to iterate the Object
        while(itr.hasNext()){
            System.out.println(itr.next());//sachin saurav dhoni kohli dravid
        }
    }
}
```

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.

```
import java.util.*;
public class Test
{
   public static void main(String[] args)
   {
       //Underlying datastructure is Hashtable
       //JDK1.2 Version
      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
      //JDK1.4 Version
      LinkedHashSet lhs = new LinkedHashSet();
      lhs.add("A");
      lhs.add("B");
      lhs.add("Z");
      lhs.add("C");
      lhs.add(10);
      System.out.println(lhs);
   }
}
```

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 => underlying datastructure is "HasTable"

LinkedHashSet => underlying datastructure is combination of "Hashtable + "linkedlist" .

HashSet => Duplicates are not allowed and insertion order is not preserved

LinkedHashSet => Duplicates are not allowed,but insertion order is preserved.

HashSet => 1.2V

LinkedHashSet => 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

100 101 102 103 104 105 106 107

firstElement() => 100

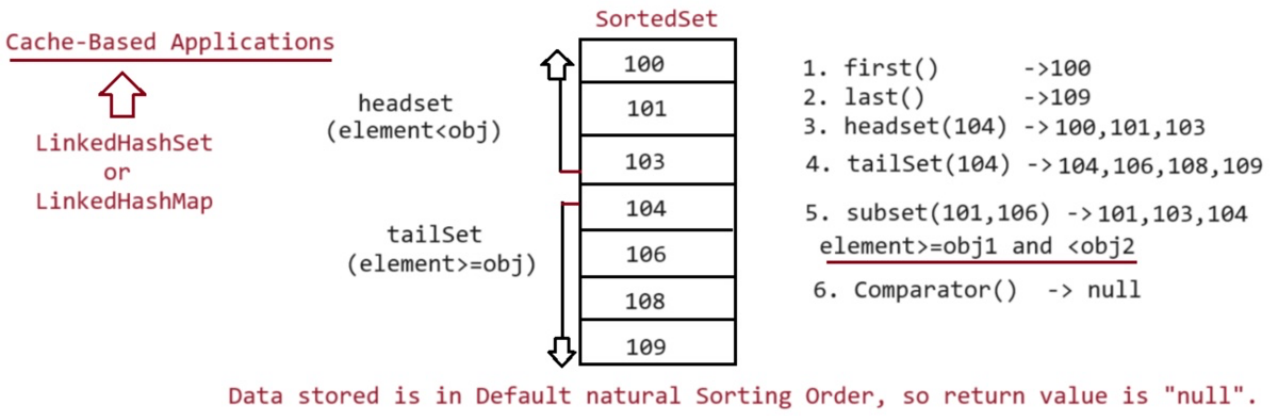
lastEleemnt() => 107

headSet(104) => 100 101 102 103

tailSet(105) => 105,106,107

subset(101,104) => 101,102,103

compartor() => null**(because it is already sorted by default)**



When to use Queue(I) implementation class Objects?

Ans. if we want to represent group of individual objects prior to processing then we should go for Queue.

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.

Name the interfaces available for Sorting the Objects

a. Comparable(I) :: meant for Default Natural Sorting Order.

b. Comparator(I) :: meant for Customized Sorting Order.

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.

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]

eg#1.

System.out.println("A".compareTo("Z")); **//A should come before Z so -ve**

System.out.println("Z".compareTo("K")); **//Z should come after K so +ve**

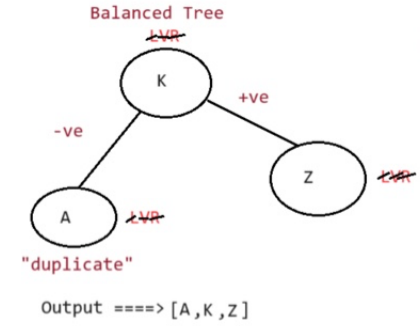
System.out.println("A".compareTo("A")); **//Both are equal zero**

System.out.println("A".compareTo(null)); **//NullPointerException**

eg#2.

```
import java.util.TreeSet;
public class TestApp{
    public static void main(String... args){
        TreeSet ts= new TreeSet();
        ts.add("K");
        ts.add("Z");  //internally "Z".compareTo("K") +ve
        ts.add("A");  //internally "A".compareTo("K") -ve
        ts.add("A");  //internally "A".compareTo("K") -ve
                      //internally "A".compareTo("A") 0
        System.out.println(ts);//[A K Z]
    }
}
```

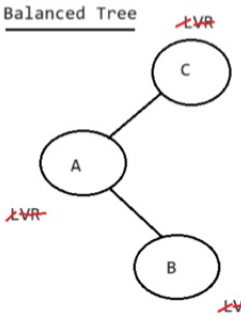
```
TreeSet ts = new TreeSet();

ts.add("K");//root node in a tree

ts.add("Z"); "Z".compareTo("K") +ve
ts.add("A"); "A".compareTo("K"); -ve

ts.add("A"); "A".compareTo("K"); -ve
             "A".compareTo("A") 0

System.out.println(ts);
```



InOrder traversal
L =>Left
V =>Vertex(print the node)
R => Right

Output ====> [A,K,Z]

```
TreeSet ts = new TreeSet();

ts.add("C");//root node in a tree

ts.add("A");// "A".compareTo("C"); -ve
ts.add("B");// "B".compareTo("C"); -ve
            // "B".compareTo("A"); +ve
```



Output => [A,B,C]

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.

Comparable

=> compareTo()

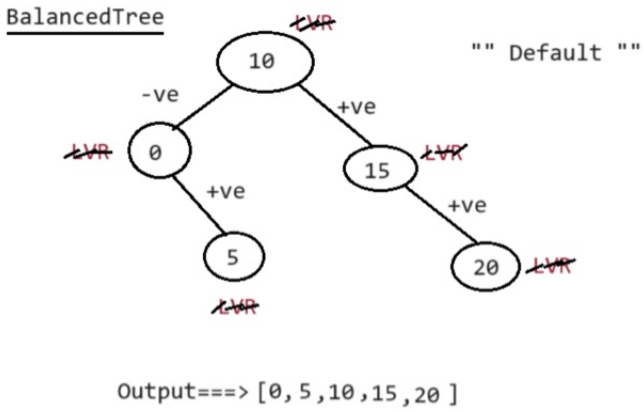
It is meant for default natural sorting order.

Comparator

=> compare()

It is meant for customized sorting order.

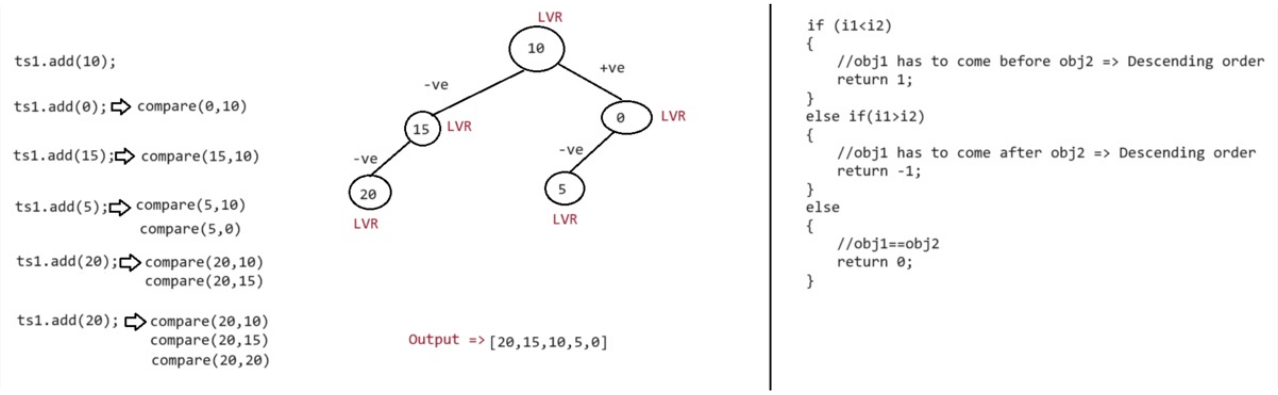
```
ts1.add(10);

ts1.add(0);  ⇨ 0.compareTo(10)
ts1.add(15); ⇨ 15.compareTo(10)
ts1.add(5);  ⇨ 5.compareTo(10)
               5.compareTo(0)
ts1.add(20); ⇨ 20.compareTo(10)
               20.compareTo(15)
ts1.add(20); ⇨ 20.compareTo(10)
               20.compareTo(15)
               20.compareTo(20)
```



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]
    }
}
```



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
    }
}
```

=> Insert String object into treeset, perform sorting in reverse of Alphabetical Order.

eg#1.

```
import java.util.*;
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        String s1=(String)obj1 ;
        String s2=obj2.toString();
        return -s1.compareTo(s2);
    }
}
public class Test
{
    public static void main(String[] args)
    {
        //TreeSet[Balanced Tree] -> Comparable :: DNS
        //public int compareTo(Object obj)
        TreeSet ts1 =new TreeSet();
        ts1.add("sachin");
        ts1.add("saurav");
        ts1.add("dhoni");
        ts1.add("kohli");
        ts1.add("yuvi");
        System.out.println(ts1);//[dhoni,kohli,sachin,saurav,yuvi]

        System.out.println();

        //TreeSet[Balanced Tree] -> Comparator :: CSO
        //public abstract int compare(Object obj1,Object obj2);
        //public abstract boolean equals(java.lang.Object);
        TreeSet ts2 =new TreeSet(new MyComparator());
        ts2.add("sachin");
        ts2.add("saurav");
        ts2.add("dhoni");
        ts2.add("kohli");
        ts2.add("yuvi");
        System.out.println(ts2);//[yuvi, saurav, sachin, kohli, dhoni]
    }
```

```
}
```

=> Insert StringBuffer object into treeset, perform sorting in Alphabetical Order.

eg#1.

```
import java.util.*;
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        String s1=obj1.toString(); //we cant do (String)obj1 as there is no relationship between string and stringbuffer
        String s2=obj2.toString();
        return -s1.compareTo(s2);
    }
}
public class Test
{
    public static void main(String[] args)
    {
        //TreeSet[Balanced Tree] -> Comparable :: DNS
        //public int compareTo(Object obj)
        TreeSet ts1 =new TreeSet();
        ts1.add(new StringBuffer("sachin"));
        ts1.add(new StringBuffer("saurav"));
        ts1.add(new StringBuffer("dhoni"));
        ts1.add(new StringBuffer("kohli"));
        ts1.add(new StringBuffer("yuvi"));
        System.out.println(ts1);//[dhoni,kohli,sachin,saurav,yuvi]

        System.out.println();

        //TreeSet[Balanced Tree] -> Comparator :: CSO
        //public abstract int compare(Object obj1,Object obj2);
        //public abstract boolean equals(java.lang.Object);
        TreeSet ts2 =new TreeSet(new MyComparator());
        ts2.add(new StringBuffer("sachin"));
        ts2.add(new StringBuffer("saurav"));
        ts2.add(new StringBuffer("dhoni"));
```

```
        ts2.add(new StringBuffer("kohli"));
        ts2.add(new StringBuffer("yuvi"));
        System.out.println(ts2);//[yuvi, saurav, sachin, kohli, dhoni]
    }
}
```

Write a java program to insert the String and StringBuffer object into TreeSet where sorting order is in increasing length order. if 2 objects have same length then consider their Alphabetical order

sample::

```
ts.add(new StringBuffer("A"));
ts.add(new StringBuffer("ABC"));
ts.add(new StringBuffer("AA"));
ts.add("XX");
ts.add("ABCE");
ts.add("A");
```

eg#1.

```
import java.util.*;
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        String s1=obj1.toString();
        String s2=obj2.toString();
        int i1= s1.length();
        int i2= s2.length();
        //increasing length order
        if (i1<i2)
        {
            //obj1 should come before obj2
            return -1;
        }
        else if(i1>i2)
        {
            //obj1 should come after obj2
            return +1;
        }
```

```
            else
            {
                //same length :: Alphabetical Order
                return s1.compareTo(s2);
            }
        }
}
public class Test
{
    public static void main(String[] args)
    {
        //TreeSet[Balanced Tree] -> Comparator :: CSO
        //public abstract int compare(Object obj1,Object obj2);
        //public abstract boolean equals(java.lang.Object);
        TreeSet ts1 =new TreeSet(new MyComparator());
        ts1.add(new StringBuffer("A"));
        ts1.add(new StringBuffer("ABC"));
        ts1.add(new StringBuffer("AA"));
        ts1.add("XX");
        ts1.add("ABCE");
        ts1.add("A");
        System.out.println(ts1);//
    }
}
```

Output

[A, AA, XX, ABC, ABCE]

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

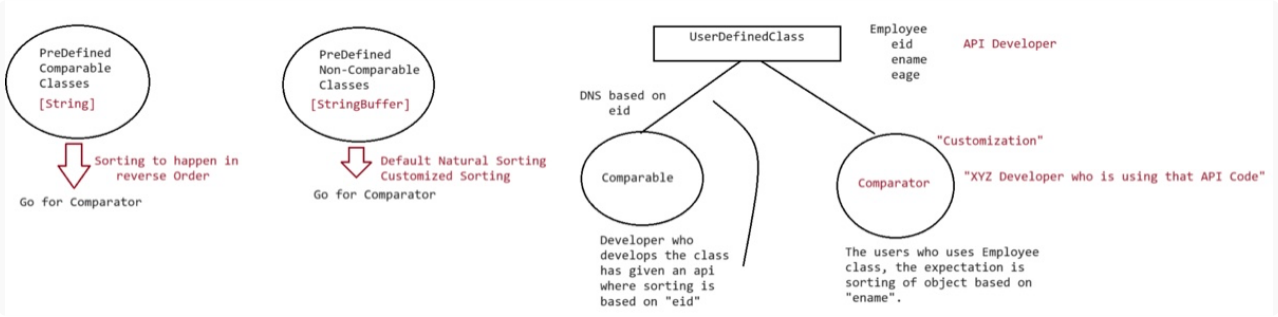
=> Default natural sorting order not available so go for Comparator only always

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.



Write a Program to Insert Employee Objects into the TreeSet where DNSO is Based on Ascending Order of EmployeeId and Customized Sorting Order is Based on Alphabetical Order of Names.

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

eg#1.

```
import java.util.*;
class Employee implements Comparable
{
    int eid;
    String ename;
    //Parameterized Constructor
    Employee(int eid,String ename)
    {
        this.eid = eid;
        this.ename = ename;
    }
    @Override
    public String toString()
```

```
    {
        return "{ "  +eid +   "----> "   + ename+  "}";
    }
    @Override
    public int compareTo(Object obj1)

        //logic for Sorting based on eid
        int id1 = this.eid;
        Employee emp1 = (Employee)obj1;
        int id2 = emp1.eid;
        if (id1<id2)

            return -1;

        else if(id1>id2)

            return +1;

        else

            return 0;


}
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        //logic for Sorting based on ename
        Employee e1 = (Employee) obj1;
        Employee e2 = (Employee) obj2;
        String s1 = e1.ename;
        String s2 = e2.ename;
        //DefaultNatural Sorting Order based on Alphabetical Order
        return s1.compareTo(s2);
    }
}
```

```
public class Test
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee(10,"sachin");
        Employee e2 = new Employee(9,"lara");
        Employee e3 = new Employee(14,"ponting");
        Employee e4 = new Employee(7,"dhoni");
        Employee e5 = new Employee(18,"kohli");
        TreeSet ts1 = new TreeSet();
        ts1.add(e1);
        ts1.add(e2);    //e2.compareTo(e1)
        ts1.add(e3);
        ts1.add(e4);
        ts1.add(e5);
        System.out.println("Default Natural Sorting Order:: Based on ID");
        System.out.println(ts1);

        System.out.println();

        TreeSet ts2 = new TreeSet(new MyComparator());
        ts2.add(e1);
        ts2.add(e2);
        ts2.add(e3);
        ts2.add(e4);
        ts2.add(e5);
        System.out.println("Customized Sorting Order:: Based on NAME");
        System.out.println(ts2);
    }
}
```

Output

Default Natural Sorting Order:: Based on ID

[{ 7----> dhoni}, { 9----> lara}, { 10----> sachin}, { 14----> ponting}, { 18---→ kohli}]

Customized Sorting Order:: Based on NAME

[{ 7----> dhoni}, { 18----> kohli}, { 9----> lara}, { 14----> ponting}, { 10---→ sachin}]

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)

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.

Difference b/w Iterator and Iterable

The target element in for-each loop should be Iterable object/array/Collection.

```
for(datatype item:target){
    .....
    .....
}
```

=> An object is set to be iterable iff corresponding class implements java.lang.Iterable interface.

=> Iterable interface introduced in 1.5 version and it's contains only one method iterator().

Syntax : public Iterator iterator();

Note: Every array class and Collection interface already implements Iterable interface.

Difference between Iterable and Iterator:

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.

interfaces

1. Collection
2. List
3. Set

Implementation class :: HashSet,LinkedHashSet

4. SortedSet
5. NavigableSet[1.6v]

Implementation class :: TreeSet

When to use Set(I) implementation class Objects?

Ans. Duplicates are not allowed

Insertion Order is not preserved

Note: To iterate any type of Collection object,the Collection Object should be "Iterable".

For Collection Interface the Parent interface is "Iterable",so Iterator interface can be used for all the Objects.

Note: Map(I) is not a part of Collection.

6. Map(<K,V>)

Implementation :: HashMap,WeakHashMap,IdentityHashMap

Dictionary,Hashtable,Properties

7. SortedMap
8. NavigableMap[1.6V]

Implementation :: TreeMap

9. Queue[1.5V]

Implementation class : BlockingQueue,PriorityBlockingQueue,...

When to use Queue(I) implementation class Objects?

Ans. if we want to represent group of individual objects prior to processing then we should go for Queue.

When to go for List(I) implementation class Objects?

Ans. duplciates are allowed, insertion order is preserved

List :: ArrayList, LinkedList

Legacy Classes :: Vector,Stack

When to go for Set(I) implementation class Objects?

Ans. duplicates are not allowed, insertion order is not preserved

Set :: HashSet, LinkedHashSet

SortedSet TreeSet[By default Data Allowed are Homogenous,Comparable, otherwise :: ClassCastException]

NavigableSet [null not allowed if we try to keep :: NullPointerException]

Name the cursors available for iterating the Collection Objects

a. Enumeration(I) => Legacy Cursor.[Vector, Stack]
=> we can perform only read operation in forward direction.

b. Iterator(I) => Universal Cursor as it can be iterated on every collection Object.
we can perform read in forward direction and remove operation.

c. ListIterator(I) => It can be used only on List(I) implementation class Object.
we can perform read,update,delete and also we can iterate in forward and backward direction.

Name the interfaces available for Sorting the Objects

a. Comparable(I) meant for Default Natural Sorting Order.
b. Comparator(I) meant for Customized Sorting Order.

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**".

+diagram left → pdf not uploaded(4feb)

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.**

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
        System.out.println(ht);
    }
}
```

Output

HASHMAP***

{1=rahul, 18=virat, 7=dhoni, 10=sachin, 45=rohith}

[1, 18, 7, 10, 45]

[rahul, virat, dhoni, sachin, rohith]

[1=rahul, 18=virat, 7=dhoni, 10=sachin, 45=rohith]

key is :: 1

Value is :: rahul

key is :: 18

Value is :: virat

key is :: 7

Value is :: dhoni

key is :: 10

Value is :: sachin

key is :: 45

Value is :: rohith

{1=rahul, 18=virat, 7=dhoni, 10=sachinrameshtendulkar, 45=rohith}

{null=null, 1=rahul, 18=virat, 7=dhoni, 10=sachinrameshtendulkar, 45=rohith}

HASHTABLE***

Exception in thread "main" java.lang.NullPointerException

at java.util.Hashtable.put(Hashtable.java:460)

at Test.main(Test.java:59)

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**

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

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.)**

LinkedHASHMAP***

{10=sachin, 7=dhoni, 18=virat, 1=rahul, 45=rohith}

[10, 7, 18, 1, 45]

[sachin, dhoni, virat, rahul, rohith]

[10=sachin, 7=dhoni, 18=virat, 1=rahul, 45=rohith]

key is :: 10

Value is :: sachin

key is :: 7

Value is :: dhoni

key is :: 18

Value is :: virat

key is :: 1

Value is :: rahul

key is :: 45

Value is :: rohith

{10=sachinrameshtendulkar, 7=dhoni, 18=virat, 1=rahul, 45=rohith}

{10=sachinrameshtendulkar, 7=dhoni, 18=virat, 1=rahul, 45=rohith, null=null}

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...

{}

+diagram left → pdf not uploaded(4feb)

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

eg#2.

```
import java.util.*;
public class Test
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("HashMap*");
        HashMap hm = new HashMap();
        Temp t = new Temp();
        hm.put(t , "sachin");
        //Making object Garbage
        t =null;
        //Explicitly Calling Garbage Collector
        System.out.println("Signal to Garbage Collector...");
        System.gc();
        Thread.sleep(5000);
        System.out.println(hm);  //{Temp=sachin}
    }
}
class Temp
{
    @Override
```

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

Output

HashMap*

Signal to Garbage Collector...

{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)

eg#1.

```
import java.util.*;
class MyComparator implements Comparator
{
    @Override
    public int compare(Object obj1,Object obj2)
    {
        Integer i1 = (Integer)obj1;
        Integer i2 = (Integer)obj2;
        return -i1.compareTo(i2);
    }
}
public class Test
{
    public static void main(String[] args)
    {
        //Underlying DS :: RedBlackTree
        //Comparable :: DNS(ascending order)
        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}

        System.out.println();

        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}
```

```
    }
}
```

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);

+diagram left → pdf not uploaded(4feb)

```
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:

+diagram left → pdf not uploaded(4feb) → very imp diagram includes streams as well

=> 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.

NOTE: Queue is covered in next section(Main Part - 7) + Concurrent Collection