Java Collections Interview Questions

1) What is framework in Java?

A framework is a popular and readymade architecture that contains a set of classes and interfaces.

2) What is the Collection framework in Java?

Collection Framework is a grouping of classes and interfaces that is used to store and manage the objects. It provides various classes like Vector, ArrayList, HashSet, Stack, etc. Java Collection framework can also be used for interfaces like Queue, Set, List, etc.

3). What is Collection?

In order to represent a group of individual objects as a single entity we use Collection

4). What are the Advantages of Collection?

1. Collection are growable in nature, we can increase or decrease the size dynamically.
2. Collection can hold both homogenous and heterogenous elements.
3. Every collection class is implemented using standard datastructure, so being a programmer to work with collection class we need not implement any logic already logic is a part of the class(API) so complexity of programming is reduced.

5). What is collections class?

It is an utility class which is a part of java.util package which consists a static method Through Which it supports the user to write less code and do more work.

eg:: Collections.sort(al);//it sorts the data in ascending order.
Collections.search(al);

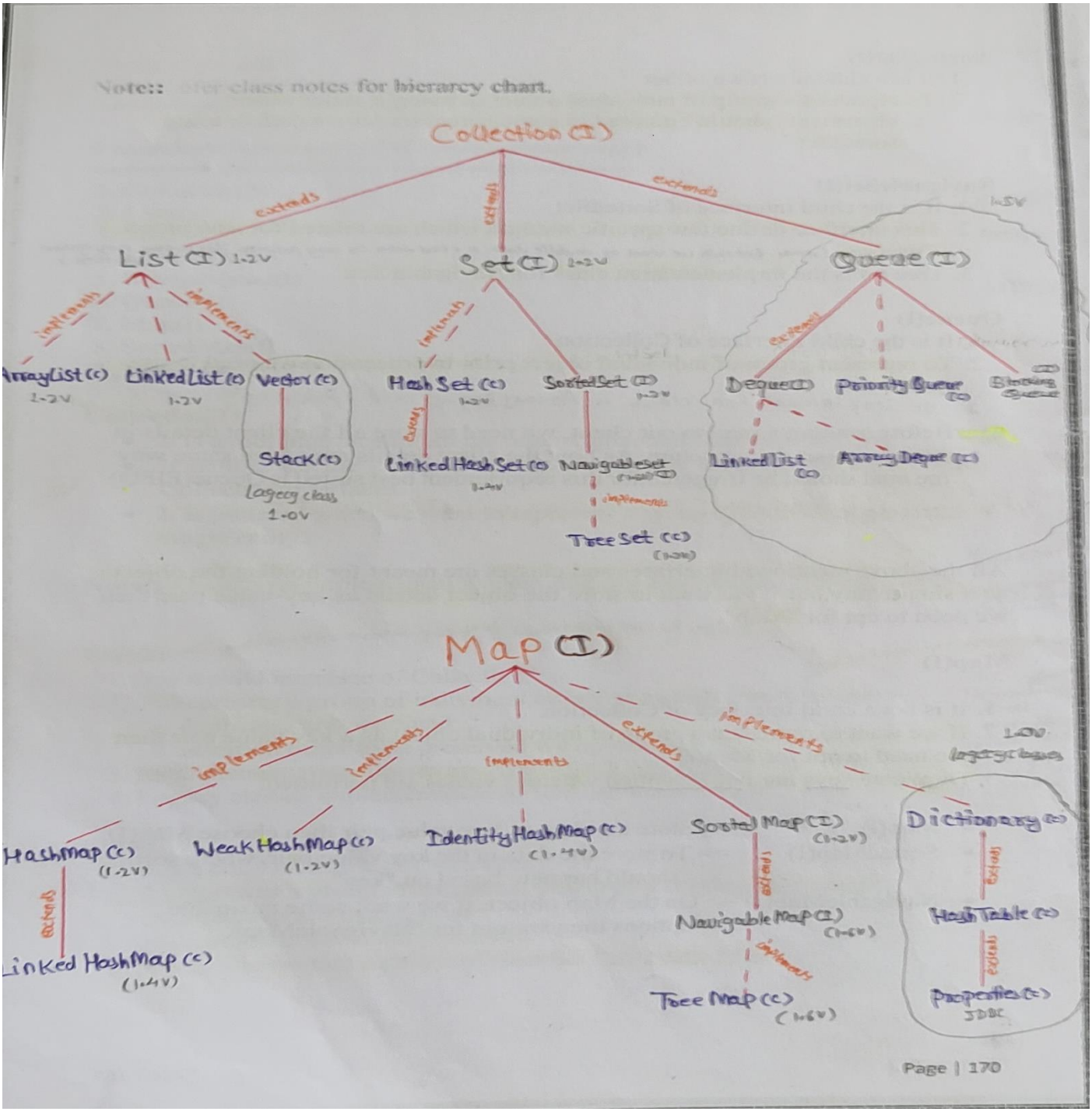
6). Difference b/w Collection and Collections?

Collection => It is an interface which is used to hold mulitple objects as a single entity

Collections => It is an utility class which is a part of java.util package, Through Which it supports the user to write less code and do more work.

eg:: Collections.sort(al);//it sorts the data in ascending order.

7). Explain the hierarcy of collection frame work?



8). List all the Interfaces of CollectionFramework?

1. Collection(I)
2. List(I)
3. Set(I)
4. SortedSet(I)
5. NavigableSet(I)
6. Queue(I)
7. Map(I)
8. SortedMap(I)
9. NavigableMap(I)

9). What is Collection(I) interface?

- In order to represent group of individual objects as a single entity we go for Collection(I).
- It is a root interface for all Collection framework.
- It defines the common methods which are required for all Collection Object.

Note:: There is not direct implementation class for Collection(I)

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 Collection
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

10). What is List(I) interface?

1. It is a child interface of Collection
2. To represent a group of individual object as a single entity where
   a. duplicates are permitted
   b. insertion order to be preserved we opt for List(I)
3. Its implementation classes are ArrayList,LinkedList.
4. Legacy classes implementation of List are Vector,Stack.

Commond methods

==========

a. **void add(int index,Object obj)**

b. **void addAll(int index,Collection c)**

c. **Object remove(int index)**

d. **Object get(int index)**

This method is used to get the elements from the respective index

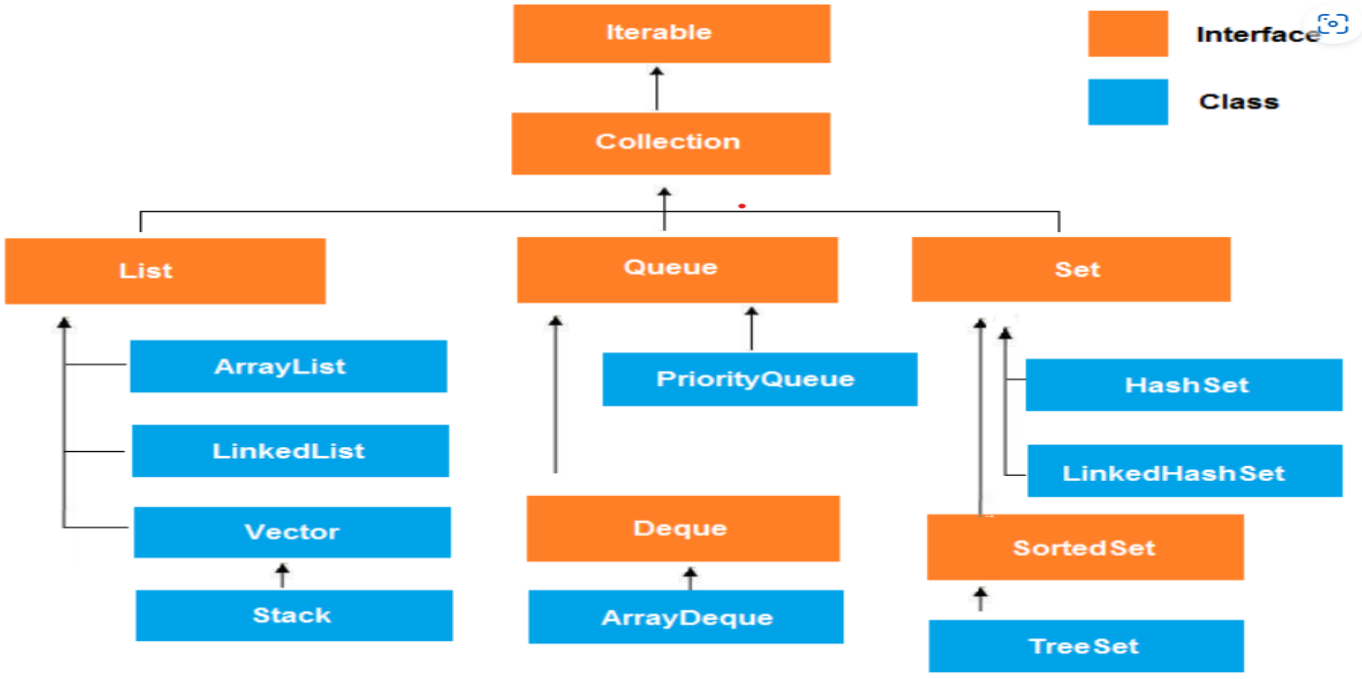
e. **Object set(int index,Object o)**

This method is used to replace a particular object at the respective index. if the element already existed in that index,it will be pushed to the right.

f. **int indexOf(Object obj)**

g. **int lastIndexOf(Object obj)**

i. **ListIterator listIterator()**



11). What is ArrayList?

Datastructure :Resizable Array/Growable Array
Heterogeneous :Yes it allows
Duplicates :yes it allows
insertion order :yes it preserves
null insertion :yes it allows

Different ways of Creating ArrayList

==========

a. **ArrayList al =new Arraylist();**
//Default capacity is 10
//Next capacity will be calculated based on :: (currentcapcity * 3/2) + 1
b. **ArrayList al =new ArrayList(int initalCapacity);**
c. **ArrayList al =new ArrayList(Collection c);**

eg#1.

```
import java.util.ArrayList;
public  class TestApp {
        public static void main(String[] args){
                ArrayList al = new ArrayList();
                al.add("A");
                al.add(10);
                al.add("A");
                al.add(null);

                System.out.println(al);//[A,10,A,null]

                al.remove(2);
                System.out.println(al); //[A,10,null]

                al.add(2,'M');
                System.out.println(al); //[A,10,M,null]

                al.add('M');
                System.out.println(al); //[A,10,M,null,M]
        }
}
```

Note:: Usally we use collection to hold the object and those objects we need to transmit over the network, if a particular object has to be transported over the n/w then those object should get the facility of "Serializalbe"

=> By default ArrayList class implements Serializable and Cloneable Interface.
=> To support RandomAccess of elements it also implements an interface called "RandomAccess".
=> ArrayList and Vector are the only implementation classes of "RandomAccess".

Advantages/Disadvantages of array list:-

=> ArrayList is best suited if the frequent operation is retrival
=> ArrayList is not good if the frequent operation is
a. inserting the element in b/w
b. removing the element based on the index
because it needs so many shifts which reduces the performances
=> To resolve this problem we need to use "LinkedList".

12) Explain the method to convert ArrayList to Array and Array to ArrayList.

```
List<String> fruitList = new ArrayList<>();
System.out.println("Converting ArrayList to Array" );
String[] item = fruitList.toArray(new String[fruitList.size()]);
for(String s : item){
   System.out.println(s);
}

System.out.println("Converting Array to ArrayList" );
List<String>l2 = new ArrayList<>();
l2 =  Arrays.asList(item);
System.out.println(l2);
```

12). Give the real example where we go for arraylist?

13). What is LinkedList?

DataStructure: doublylinkedlist
heterogenous : yes allowed
null : yes allowed
duplicates : yes allowed
insertionorder : preserved

- It implements Serializable,Cloneable interface but not RandomAccess.
- It is best suited when our frequent operation is "insertion/deletion in middle".
- It is not suited when our frequent operation is "retrieval".

Constructors of LinkedList

==========

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

Using linkedList making it to work like stack and queue is a common requirement,to support this
LinkedList class has given few methods as shown below

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

eg#1.

```
import java.util.LinkedList;
public  class TestApp {
        public static void main(String[] args){
                LinkedList ll=new LinkedList();
                ll.add("sachin");
                ll.add(10);
                ll.add(null);
                ll.add("sachin");
                System.out.println(ll);//[sachin,10,null,sachin]
```

```
            ll.set(0,"kohli");
            System.out.println(ll);//[kohli,10,null,sachin]

            ll.removeLast();
            System.out.println(ll);//[kohli,10,null]

            ll.addFirst("sachin");
            System.out.println(ll);//[sachin,kohli,10,null]
        }
}
```

Real time use of linkedList?

Real life example for:

1) Singly linked list

1. Human brain of a child(In order to remember something eg . poem he has to link it , if you will ask him the last line he will have to read from the first line)
2. Undo button of any application like Microsoft Word, Paint, etc:

2) Doubly linked list

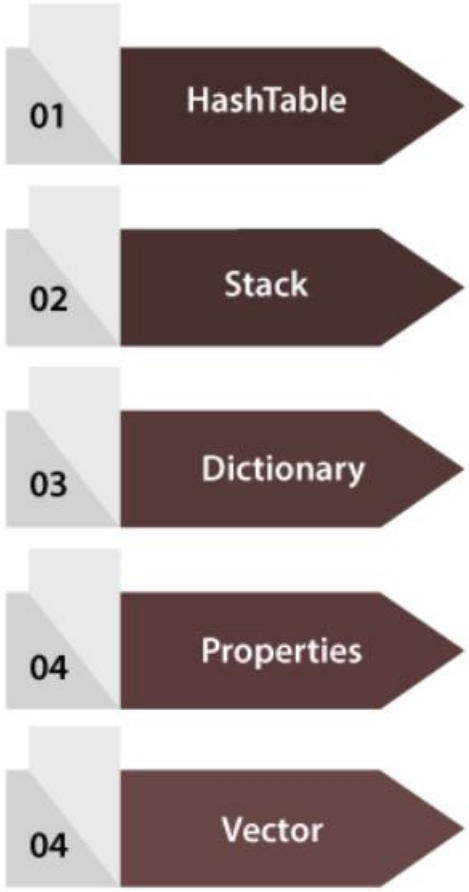
1. Train coaches are connected with the next and the previous ones.
2. Roller chain of bicycle(doubly circular linked list)
3. Browser's Next and Previous Button: a linked list of URLs
4. Microsoft Image Viewer's Next and Previous Button: a linked list of images
5. Undo and Redo button of Photoshop, a linked list of states.
6. Each tab in browser is an example of doubly linked list.

3) Circular linked list

1. Escalator
2. Multiple Player board game(luddo)

14). What is legacy class?

classes and interfaces that formed the collections framework in the older version of Java are known as **Legacy classes**.



15). What is vector ?

DataStructure : GrowableArray/Resizable Array
heterogenous : yes allowed
null : yes allowed
duplicates : yes allowed
insertionorder : preserved

- It implements Serializable,Cloneable interface and also RandomAccess.
- It is best suited when our frequent operation is "retrieval".
- It is not suited when our frequent operation is "insertion/deletion".
- Most of the methods of Vector is "synchronized" so it is "Thread safe".
- It is a legacy class(1.0V)

Since the class is a legacy class, the method names of those class is lengthy.

Add element

a. add(Object o) => Collection
b. add(int index,Object o) => List
c. addElement(Object o) => Vector

Remove element

a. remove(Object o) => Collection
b. removeElement(Object o) => Vector

---

a. remove(int index) => List
b. removeElementAt(int index) => Vector

---

a. clear() => Collection
b. removeAllElements() => Vector

Retrieve elements

a. Object get(int index) => List
b. Object elementAt(int index) => Vector
c. Object firstElement() => Vector
d. Object lastElement() => Vector

Constructors available in Vector

==========

1. **Vector v=new Vector();**
   a. Defalut capacity is 10
   b. if capacity is full, it grows in size with a capacity as (oldcapacity *2)
2. **Vector v=new Vector(int capacity);**
3. **Vector v=new Vector(int capacity,int incrementalcapacity)**
   To use the memory effectively,we take the help of incremental capacity.
   a. Vector will be created with given capacity
   b. if the capacity if full, then vector will not grow by double, it will grow based on incremental capacity.
4. **Vector v=new Vector(Collection c);**

Vector example

==========

```
import java.util.Vector;
public  class TestApp {
   public static void main(String[] args){
                Vector v= new Vector();
                System.out.println("Default capacity is :: " +v.capacity());//10

                for (int i=0;i<=9; i++){
                        v.addElement(i);
                }

                System.out.println("Capacity after adding elements  is :: " +v.capacity());//10

                v.addElement("sachin");
                System.out.println("Increased capacity is :: " +v.capacity());//20

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

16). What is Stack?

1. It is a child class of Vector
2. This class is specially designed to follow LIFO/FILO order

Constructor :: Stack s=new Stack()

Common operation associated with stack are push,pop,peek,empty,search

1. **Object push(Object o)** => Push the element into stack
2. **Object pop()** => remove the last element from the stack
3. **Object peek()** => returns the top element of the stack,without removal
4. **boolean empty()** => checks whether the stack is empty or not
5. **int search(Object o)** => It checks for the specified object and it returns the offset if found
   otherwise it returns -1.

eg#1.

```
import java.util.Stack;
public  class TestApp {
        public static void main(String[] args){
                Stack s=new Stack();//Default Capacity is :10
                s.push("A");
                s.push("B");
                s.push("C");
                System.out.println(s);//[A,B,C]

                s.pop();
                System.out.println(s);//[A,B]

                s.push("C");
                System.out.println(s);//[A,B,C]

                System.out.println(s.search("C"));//1(offset it returns)
                System.out.println(s.search("sachin"));//-1
        }
}
```

5) Distinguish between ArrayList and Vector in the Java collection framework.

ArrayList	Vector
ArrayList is cannot be synchronized.	Vector can be is synchronized.
It is not a legacy class.	It is a legacy class.
It can increase its size by 50% of the size of the array.	It can increase its size by doubling the size of the array.
ArrayList is not thread-safe.	Vector is a thread-safe.

Q.Differences between ArrayList and LinkedList?

ArrayList	LinkedList
It is the best choice if our frequent operation is retrieval	It is the best choice if our frequent Operation is insertion and deletion
ArrayList is the worst choice if our frequent operation is insertion or deletion	LinkedList is the worst choice if our frequent operation is retrieval operation
Underlying data structure for ArrayList is resizable or growable Array.	Underlying data structure is Double Linked List.
ArrayList implements RandomAccess interface	LinkedList doesn't implement RandomAccess interface

Time complexity :-

작업	메소드	ArrayList	LinkedList
add at last index	add()	O(1)	O(1)
add at given index	add(index, value)	O(N)	O(1)
remove by index	remove(index)	O(N)	O(1)
remove by value	remove(value)	O(N)	O(1)
get by index	get(index)	O(1)	O(N)
search by value	indexOf(value)	O(N)	O(N)

Q. ArrayList methods are non-synchronized,How to get synchronized version of ArrayList?

=> To get synchronized version of ArrayList, we take the help of Utility class called "Collections".

The methods to make collection thread safe are:

- Collections.synchronizedList(list);
- Collections.synchronizedMap(map);
- Collections.synchronizedSet(set);

Ex:-

ArrayList al = new ArrayList();//non-synchronized version
List l = Collections.synchronizedList(al);

al => nonsychrnoized version
l => synchronized version

Q. Explain Cursor in CollectionFramework?

In order to retrieve the object from the collection one by one we need to use Cursors

There are 3 cursors in Collection Framework

a. Enumeration(legacy cursor)
b. Iterator(universal cursor)
c. ListIterator

Enumeration

==========

1. It is a cursor which can be used on **legacy** classes
2. It can be used to get the collection object one by one.
3. It has 2 methods
   a. hasMoreElements=> checks whether still object is there are not if found returns true otherwise false.
   b. nextElement => returns the Object at the current position and change the cursor to the next object.

eg#1.

```
Vector v= new Vector();
for (int i=1;i<=10 ;i++ ){
        v.addElement(i);
}
System.out.println("Accessing through enumeration");

Enumeration e= v.elements();
while(e.hasMoreElements()){
        Integer i =(Integer)e.nextElement();
```

```
if (i%2==0)
  System.out.println(i);
}
```

Limitations

1. It is applicable only for Legacy classes.
2. Using enumeration we can perform only read operation, we can't perform remove operation.

To overcome this limitation we need to use a cursor called **"Iterator".**

Iterator

=======

- It is cursor which can be used to perform both read and remove operation.
- It is called as universal cursor becoz it can be used on any Collection object to retrieve the Object one by one.
- This iterator object can be obtained by using iterator() of Collection(I).

methods are

1. **hasNext()**=> checks whether still object is there are not if found returns true otherwise false.
2. **next()** => returns the Object at the current position and change the cursor to the next object.
3. **remove()** => removes the particular object where the cursor is pointing at.

```
ArrayList l=new ArrayList();
for (int i=1;i<=10 ;i++ ){
        l.add(i);
}
Iterator itr= l.iterator();
while (itr.hasNext()){
        Integer i =(Integer) itr.next();
        if (i%2==0)
                System.out.println(i);//2 4 6 8 10
        else
                itr.remove();
}
System.out.println(l);//[2,4,6,8,10]
```

Drawbacks w.r.t Iterator

==========

1. By using Enumeration and Iteration, we can iterate the elements only in forward direction we can't move in backward direction so we say Enumeration and Iterator has single direction cursor
2. using Iterator we can just perform read and remove operation oly, we can't perfom replace and adding of new object.

To resolve this problem we need to use **ListIterator.**

ListIterator

==========

- It is the child interface of Iterator
- By using ListIterator we can move in both the direction, forward and backward.
- Addition of read and remove operation we can also perform update and addition of new object to existing Collection.

Note::

ListIterator is the most powerful cursor, but it can be applied only on List object but not on other Collection objects.

How to get ListIterator Object?

public ListIterator listIterator()

Methods associated with ListIterator

ForwardDirection

```
boolean hasNext();
Object next();
int nextIndex();
```

==========

BackwardDirection

```
boolean hasPrevious();
Object previous();
int previousIndex();
```

==========

```
void remove();
void set(Object o);
void add(Object o);
```

eg#1.

```
LinkedList l=new LinkedList();
l.add("sachin");
l.add("kohli");
l.add("rahul");
l.add("dhoni");
l.add("saurav");
System.out.println(l); //[sachin,kohli,rahul,dhoni,saurav]

ListIterator litr= l.listIterator();
while (litr.hasNext()){
        String data= (String)litr.next();
        if (data.equals("dhoni"))
                litr.remove();
        if (data.equals("sachin"))
                litr.add("tendulkar");
        if (data.equals("rahul")){
                litr.set("dravid");
        }
}
System.out.println(l); //[sachin,tendulkar,kohli,dravid,saurav]
```

eg#2.

```
LinkedList l=new LinkedList();
l.add("sachin");
l.add("kohli");
l.add("rahul");
l.add("dhoni");
l.add("saurav");

System.out.println(l); //[sachin,kohli,rahul,dhoni,saurav]

ListIterator litr= l.listIterator(l.size()); //to point on last

while (litr.hasPrevious()){
        String data= (String)litr.previous();
        System.out.print(data+"\t");//saurav dhoni rahul kohli
                                    sachin
}
System.out.println();
```

Comparison b/w cursors :-

Enumeration	Iterator	ListIterator
can be applied only on legacy classes.	can be applied on all Collection objects.	can be applied only on List Object.
Movement is single direction.	**Movement is single direction.**	**Movement is Bi-direction.**
To get Object we use elements(). methods are 2:- **hasMoreElements(),** **nextElement()**	To get Object we use iterator(). **methods are 3:-** hasNext(), next(), remove()	To get Object we use listiterator(). **methods are 9:-** hasNext(),next(),nextIndex(), Remove(),hasPrevious(), previous(),previousIndex(), set(Object),add(Object)
operation allowed is only read.	**operation allowed is read and remove.**	**operation allowed is read ,remove,add,replace.**
Version : 1.0	Version : 1.2	Version : 1.2

**********Set**********

Set(I)

======

1. It is a child interface of Collection
2. If we want to represent a group of individual object as a single entity where
   a. duplicates are not allowed
   b. insertion order is not preserved then we need to go for Set
3. Set interface does not contain any new method rather it takes the help of Collection interface
   methods only(12 Methods).
4. its implementation class is HashSet, LinkedHashSet.

Q) Differentiate between List and Set.

List	Set
An ordered collection of elements	**An unordered collection of elements**
Preserves the insertion order	Doesn't preserves the insertion order
Duplicate values are allowed	**Duplicate values are not allowed**
Any number of null values can be stored	Only one null values can be stored
ListIterator can be used to traverse the List in any direction	**ListIterator cannot be used to traverse a Set**
Contains a legacy class called vector,stack	Doesn't contains any legacy class

HashSet

=======

- **Underlying DataStructure :** HashTable(internally uses hashing technique(hashcode) to keep the data)
- **heterogeneous elements : Yes**
- **insertion order preserved : No**
- **null insertion allowed : Yes**
- **duplicates allowed : No**
- **interface implementing : Serializable,Cloneable**
- **Best suited : since the data is stored based on hashing the searching operation is very fast.**

Constructor associated with HashSet

==========

1. **HashSet s=new HashSet();** //Default capacity => 16
   Default LoadFactor => 0.75
2. **HashSet s=new HashSet(int initialCapacity);**
   |=> user specified capacity
3. **HashSet s=new HashSet(int initialCapacity,int loadFactor);**
   |=> user specified capacity
   |=> user specified loadFactor
4. **HashSet s=new HashSet(Collection c);**

eg#1.

```
import java.util.HashSet;

public  class TestApp {
        public static void main(String[] args){
                        HashSet hs=new HashSet();
                        hs.add("B");
                        hs.add("C");
                        hs.add("D");
                        hs.add("Z");
                        hs.add(10);
                        hs.add(null);
                        System.out.println(hs.add("Z")); //false
                        System.out.println(hs); //[null,B,C,D,Z,10] random order
                                                    can't predict
        }
}
```

public boolean add(Object o)

|=> returns true, if object is added to HashTable internally otherwise it **returns**
false through which duplicates are avoided.

LinkedHashSet

==========

- It is the child class of HashSet
- It exhibits almost the same behaviour of HashSet.
- Insertion order preserved.

Note::

To build cacheBased application where duplicates are not allowed and insertion order is not important then we go for HashSet or LinkedHashSet.

Ex:-

```
LinkedHashSet hs=new LinkedHashSet();
                hs.add("B");
                hs.add("C");
                hs.add("D");
                hs.add("Z");
                hs.add(10);
                hs.add(null);
                System.out.println(hs.add(10));//false
                System.out.println(hs);//[B,C,D,Z,10,null]
```

Difference b/w HashSet and LinkedHashSet

==========

HashSet => Underlying datastructure is "HashTable".
LinkedHashSet => Underlying datastructure is "LinkedList + HashTable".

HashSet => Insertion order is not preserved and duplicates are not allowed
LinkedHashSet => Insertion order is preserved,duplicates are not allowed

HashSet => Introduced in 1.2V
LinkedHashSet => Introduced in 1.4V

SortedSet(I)

==========

1. It is the child interface of Set
2. If we want to represent a group of individual object as a single entity, where
   a.duplicates are not allowed
   b.insertion order is preserved based on some sorting order
   then we opt for SortedSet.
3. The default natural sorting order of Numbers is "Ascending order".
   The default natrual sorting order of String is "Alphabetical order".
4. The sorting can be made as Default natural sorting order or we can use "Customized Sorting order".
5. Its implementation class is TreeSet.

Methods associated with SortedSet

==========

1. **Object first()** => it returns the first element in the set
2. **Object last()** => it retursn the last element in the set
3. **SortedSet headSet(Object o)** => it returns the element < obj
4. **SortedSet tailSet(Object o)** => it returns the element > = obj
5. **SortedSet subset(Object o1,Object o2)** => it returns the element >=obj1 but less than obj2
7. **Compartor comparator** => If the elements are arranged in some default natural sorting order then it returns null, otherwise it returns the underlying sorted order mechanism value.

Ex::-

101,102,103,105,106,108,109

first() => 101
last () => 109
headSet(105) => 101,102,103
tailSet(105) => 105,106,108,109
subset(101,106) =>101,102,103,105
comparator() => null

TreeSet

==========

1. **Underlying datastructure :** is "balancedtree".
2. **insertion order preserved**: no(elements will stored based on some sorting order)
3. **duplicated allowed :** no
4. **heterogenous elements :** no(Sorting can't be done if it is heterogenous)
5. **null insertion possible :** no
6. **interface implementation :** Serializable and Cloneable, but not RandomAccess

Constructor associated with TreeSet

==========

1. TreeSet t=new TreeSet();

=> Elements added will be in default natrual sorting order

eg:: String => Alphabetical order

Number => Ascending order

2. TreeSet t=new TreeSet(Comparator cmp);

=> Elements will be added in TreeSet based on the Compartor written by the programmer.

3. TreeSet t=new TreeSet(Collection c);

4. TreeSet t=new TreeSet(SortedSet s);

eg#1.

```
TreeSet ts=new TreeSet();
ts.add("A");
ts.add("a");
ts.add("Z");
ts.add("B");
ts.add("L");
System.out.println(ts);//[A,B,L,Z,a]

ts.add(new Integer(10));//RE:ClassCastException

ts.add(null);//RE:NullPointerException
```

- If we try to add heterogenous elements then it would result in "ClassCastException".
- If we try to add null then it would result in "NullPointerException".

Note:

In the lower version below 1.8 in Treeset null as the first element was allowed, but in higer version even first element as null is also not allowed it would result in "NullPointerException".

eg#2.

```
import java.util.TreeSet;
public  class TestApp {
        public static void main(String[] args){
                        TreeSet ts=new TreeSet();
                        ts.add(new StringBuffer("A"));
                        ts.add(new StringBuffer("Z"));
                        ts.add(new StringBuffer("L"));
                        ts.add(new StringBuffer("B"));
                        System.out.println(ts);//RE:ClassCastException
        }
}
```

- When the elements are added into TreeSet based on natural sorting order the added object should be
  a. Homogenous
  b. It should implement Comparable interface
- If it fails to do jvm at the run time would throw an Exception called "ClassCastException".

Note::

String and all wrapper class has implemented an interface called "Comparable" so only for String and Number type object default natural sorting order is "Alphabetical and Ascending order".

Sorting in Collections :-

1. Comparable(I)=> default natural sorting order
2. Comparator(I)=> customized sorting order

Note:-
If we are not satisified with default natural sorting internally done by jvm, then we customize the sorting order as per our requirement,to do so we need to take the help of "Comparator".

Comparable

=> It is a part of java.lang package
=> It has only one method called int compareTo(obj)
=> It is purely meant for natural sorting order

Comparator

=>It is a part of java.util package
=>It has 2 methods
public int compare(Object obj1,Object obj2)
public boolean equals(Object obj)
=>It is purely meant for customizing the sorting order

Note:: while implementing an interface called Compartor, we can give body only for onemethod called compare(Object obj1,Object obj2), where as for equals() the body would come from Object class(inheritance at its best)

eg#1

```
class MyComparator implements Comparator{
        @Override
        public int compare(Object obj1,Object obj2){
                //body for compare method
        }
}
```

Note::

public int compartor(Object obj1,Object obj2)
|=> returns -ve,iff obj1 has to come before obj2
|=> returns +ve,iff obj1 has to come after obj2
|=> returns 0,iff obj1 and obj2 are equal

Difference b/w Comparable and Compartor interface?

Comparable	Comparator
• **meant for default natrual sorting order**	• **meant for custmoize sorting order**
• java.lang package	• java.util package
• **only one method :-** **int compareTo(Object obj)** **ex:- o1.compareTo(o2);**	• **2 method :-** **int compare(Object obj1,Object obj2)** **boolean equals(Object obj)**
• Predefined internal implementation classes are String,Wrapper class	• Predefined internal implementation classes are Collator and RuleBaseCollator.

Keypoints on implementation classes of Set

==========

HashSet	LinkedHashSet	TreeSet
underlying datastructure is HashTable.	**underlying datastructure is LinkedList+HashTable.**	**underlying datastructure is balancedtree.**
insertion order not preserved	insertion order is preserved	insertion order is not preserved
heterogenous objects are allowed.	**heterogenous objects are allowed**	**heterogenous objects are not allowed(default nature)**
duplicate objects not allowed.	duplicate objects not allowed.	duplicate objects not allowed.
Sorting order not applicable.	**Sorting order not applicable.**	**Sorting order applicable.**
null acceptance allowed.	null acceptance allowed.	null acceptance as first element allowed till jdk1.6. from 1.7V onward null acceptance is not allowed.

Queue(I)

========

1. It is a child interface of Collection
2. If we want to represent group of individual objects Prior to Processing then we should go for Queue.
3. From 1.5 version LinkedList implements Queue
4. Usually Queue follows FIFO order,but based on our needs we can implement our own Priorities.
   eg:PriorityQueue
5. LinkedListBased Implementation Queue also follows FirstInFirstOut Order only(FIFO).

eg: If our application wants to send some message,mail to the clients before sending it will collect in one datastructure that datastructure should support of keeping the data in same order the data is been sent so in the same order of storage the processing can happen.
In this scenario the best suited datastructrue is "Queue".

Methods

=======

boolean offer(Object obj)
=> To add an Object into the Queue

Object peek()
=> Returns the first element of the Queue
=> If Queue is empty it returns null

Object element()
=> Returns the first element of the Queue
=> If Queue is empty it throws an Exception called "NoSuchElementException".

Object poll()
=> It removes and returns the first element of the Queue
=> If Queue is empty it returns null

Object remove()
=> It removes and returns the first element of the Queue
=> If Queue is empty it returns null

PriorityQueue

==========

- **Underlying DataStructure**=> Queue
- **Insertion order** => Based on the priority the elements will be inserted
- **order** => default natrual sorting order
- **customization** => possible,but should implement Comparator.
- **Duplicates** => not allowed
- **Heterogenous elements** => if we are dependant on natural sorting order then not allowed
  if it is based on customisation then it can be allowed.
- **null elements** => not allowed.

Constructors

==========

PriorityQueue p=new PriorityQueue()
//Default Capacity => 11

PriorityQueue p=new PriorityQueue(int capacity)
PriorityQueue p=new PriorityQueue(int capacity, Comparator c)
PriorityQueue p=new PriorityQueue(SortedSet s)
PriorityQueue p=new PriorityQueue(Collection c)

eg#1.

```
PriorityQueue<Integer> p = new PriorityQueue<Integer>();
//System.out.println(p.poll());
//System.out.println(p.remove());

for (int i=0; i<=10;i++ ){
        p.offer(i);
}
System.out.println(p);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

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

Note:: Some operating System, wont give support for PriorityQueue.
To get the support we need to install batch files from the Service provider(Microsoft).

eg#2.

```
import java.util.PriorityQueue;
public  class TestApp {
        public static void main(String[] args)
        {
     PriorityQueue<String> q = new PriorityQueue<String>(15,new MyComparator());
                q.offer("Z");
                q.offer("A");
                q.offer("L");
                q.offer("X");
                System.out.println(q);//[Z, X, L, A]
        }
}
class MyComparator implements java.util.Comparator{

        @Override
        public int compare(Object obj1,Object obj2){

                String s1=(String)obj1;
                String s2=obj2.toString();

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

**********Map**********

Map

===

1. It is not a child interface of Collection
2. To hold the Object as key-value pair we need to use Map.
3. Key can't be duplicated, Value can be duplicated.
4. Key and value both are treated as Objects
5. Every key and value pair is treated as "Entry".

Common methods available in Map(I) for all the Map implementation class Object

==========

1. **Object put(Object Key,Object value)**
   => if the key already exists, then it would just add new value to that key and old value will be returned to the user.
2. **Object putAll(Map m)**
3. **Object get(Object key)**
4. **Object remove(Object key)**
5. **boolean containsKey(Object key)**
6. **boolean containsValue(Object value)**
7. **boolean isEmpty()**
8. **int size()**
9. **void clear()**

**Collection view methods of Map
10.**Set keySet()**
11.**Collection values()**
12.**Set entrySet()**

Entry
=====

- Each key-value pair is called one Entry.
- Without existence of Map Object,Entry object won't exists.
- Interface Entry should be a part of Map interface.

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

HashMap
=======

- **Underlying datastructure:** hashtable.
- **heterogenous elements :** yes.
- **duplicated allowed :** keys not allowed,but values can be duplicated.
- **insertion order :** not preserved because of Hashing technique(hashCode value of keys)
- **null insertion :** allowed for keys (only once),but for values (any no).

Constructor
==========

1. **HashMap hm=new HashMap();**
   //Default capacity => 16
   // load factor => 0.75
2. **HashMap hm=new HashMap(int initialCapacity);**
3. **HashMap hm=new HashMap(int initalCapacity,int fillratio);**
4. **HashMap hm=new HashMap(Map m);**

eg#1.

```
HashMap hm= new HashMap();
hm.put(10,"sachin");
hm.put(7,"dhoni");
hm.put(18,"kohli");
hm.put(19,"dravid");

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

System.out.println(hm.put(10,"messi"));//sachin
System.out.println(hm); //{18=kohli, 19=dravid, 7=dhoni,
                                    10=messi}

Set s=hm.keySet();
System.out.println(s); //[18,19,7,10]

Collection c = hm.values();
System.out.println(c); //[kohli,dravid,dhoni,messi]

Set s1=hm.entrySet();

System.out.println(s1);//[18=kohli,19=dravid,7=dhoni,10=sachin]

Iterator itr=s1.iterator();
while(itr.hasNext()){
        Map.Entry entry=(Map.Entry)itr.next();
        System.out.println(entry.getKey() +"----> "+
                                    entry.getValue());
        if (entry.getValue().equals("messi"))
                entry.setValue("sachin");
}

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

Difference b/w HashMap and Hashtable?

HashMap => methods are non-synchronized.
Hashtable=> methods are synchronized

HashMap => At a time multiple threads can act on a object, so it is not ThreadSafe.
Hashtable => At a time only one thread can act on a object, so it is ThreadSafe.

HashMap => performance is high.
Hashtable => performance is low.

HashMap => introduced in 1.2V not a legacy class.
Hashtable => introduced in 1.0v is a legacy class.

HashMap => null insertion as a key or value is permitted.
Hashtable => null insetion as a key or value is not permited,it would result in NullPointerException.

Note:: To get Synchronized version of HashMap, we use a method called sycnchronizedMap() using Collections class(utility class).

LinkedHashMap

==========

- **Underlying datastructure:** linkedlist + hashtable.
- **heterogenous elements :** yes.
- **duplicated allowed :** keys not allowed,but values can be duplicated.
- **insertion order :** not preserved because of Hashing technique(hashCode value of keys)
- **null insertion :** allowed for keys (only once),but for values (any no).

if we replace HashMap with LinkedHashMap then the output of the above pgm would be
{10=sachin, 7=dhoni, 18=kohli, 19=dravid}
sachin
{10=messi, 7=dhoni, 18=kohli, 19=dravid}

[10, 7, 18, 19]
[messi, dhoni, kohli, dravid]
[10=messi, 7=dhoni, 18=kohli, 19=dravid]
10----> messi
7----> dhoni
18----> kohli
19----> dravid
{10=sachin, 7=dhoni, 18=kohli, 19=dravid}

Note:: HashMap and LinkedHashMap best suited when we develop cache based application where duplicates should not be allowed,but focus only on the object and its insertion order.

IdentityHashMap(1.4v)

==========

It is almost similar to HashMap only but with only one difference.

a. HashMap uses equals() to compare the duplication of keys which is meant for content comparison.
b. IdentityHashMap uses == to check for the duplication of keys which is meant for reference comparison.

eg#1.

```
import java.util.HashMap;
public  class TestApp {
        public static void main(String[] args){
                        HashMap hm= new HashMap();

                        Integer i1=new Integer(10);
                        Integer i2=new Integer(10);

                        hm.put(i1,"sachin");
                        hm.put(i2,"messi");

                        System.out.println(hm); //{10=messi}

        }
}
```

If the above code is replace with IdentityHashMap then the output would be {10=sachin, 10=messi}.

WeakHashMap

==========

WeakHashMap is almost same as HashMap,but with the following difference

1. HashMap would dominate GarbageCollector, that is if Object does not have any reference still it is not eligible for garabage collection as it is associated with HashMap.
2. WeakHashMap can't dominate GarbageCollection, that is if Object does not have any reference then it is eligible for Garbage Collection even though it is associated with WeakHashMap.

eg#1.

```
import java.util.HashMap;
public  class TestApp {
        public static void main(String[] args)throws Exception{
                HashMap hm= new HashMap();
                Temp t= new Temp();
                hm.put(t,"nitin");
                System.out.println(hm);//{temp=nitin}

                t=null;
                System.gc();
                Thread.sleep(5000);//5sec
                System.out.println(hm);//{temp=nitin}
        }
}
class Temp{
        @Override
        public String toString(){
                return "temp";
        }
        @Override
        public void finalize(){
                System.out.println("garbage collector is cleaning the object");
        }
}
```

If we replace HashMap with WeakHashMap, then the output would be
{temp=nitin}
garbage collector is cleaning the object
{}

SortedMap

========

- It is a child interface of a Map.
- If we want to represent a group of key-value pair based on some sorting order then we need to use "SortedMap".

SortedMap defines the following specific methods

==========

1. **Object firstKey()**
2. **Object lastKey()**
3. **SortedMap headMap(Object key)**
4. **SortedMap tailMap(Object key)**
5. **SortedMap subMap(Object key1,Object key2)**
6. **Compartor compartor()**

TreeMap

=======

- **Underlying datastructure is :** red-black tree.
- **duplicate allowed** **:** keys not allowed,values are allowed.
- **insertion order** **:** not preserved,but it will stored based on some sorting order.
- **heterogenous objects** **:** w.r.t to keys not allowed,it would result in ClassCast Exception value Object can be heterogenous.
- **null values** **:** it would result in nullpointer exception after jdk1.7

  before jdk1.6 only as the first key element it was allowed.

Since in TreeMap,Sorting will happen based on keys we need to refer to

1. Comparable(I)=> default natural sorting order
2. Comparator(I)=> customized sorting order

Constructor in TreeMap

==========

1. **TreeMap t=new TreeMap();**
2. **TreeMap t=new TreeMap(Comparator c);**
3. **TreeMap t=new TreeMap(SortedMap s);**
4. **TreeMap t=new TreeMap(Map m);**

eg#1.

```
import java.util.TreeMap;
public  class TestApp {
        public static void main(String[] args){
                TreeMap tm=new TreeMap();
                tm.put("ZZZ",10);
                tm.put("AAA",100);
                tm.put("LLL",null);
                tm.put("CCC",10);
                tm.put("ZZZ",150);
                System.out.println(tm); //{AAA=100, CCC=10, LLL=null, ZZZ=150}
                tm.put(null,10); //RE:NullPointerException
                tm.put(10,10); //RE:ClassCastException
        }
}
```

eg#2.

```
import java.util.TreeMap;
import java.util.Comparator;
public  class TestApp {
        public static void main(String[] args){
                TreeMap tm=new TreeMap(new MyComparator());
                tm.put("ZZZ",10);
                tm.put("AAA",20);
                tm.put("LLL",40);
                tm.put("CCC",30);
                tm.put("XXX",150);
             System.out.println(tm);//{ZZZ=10,XXX=150,LLL=40,CCC=30,AAA=20}
        }
```

```
}
class MyComparator implements Comparator{

        @Override
        public int compare(Object obj1,Object obj2){
                String s1=(String)obj1;
                String s2=obj2.toString();
                return s2.compareTo(s1);

        }
}
```

Legacy classes

==========

Hashtable(1.0v)

- **Underlying datastructure is :** hashtable only.
- **duplicate allowed :** keys not allowed,values are allowed.
- **insertion order :** not preserved becoz it is based on hashcode of key Object.
- **heterogenous objects :** allowed for both keys and values
- **null values :** not allowed for both keys and values, it would result in NullPointerException.

Constructor

==========

1. **Hashtable hm=new Hashtable();**
   //Default capacity =>11
   //Fill ratio =>0.75
2. **Hashtable hm=new Hashtable(int initialCapacity);**
3. **Hashtable hm=new Hashtable(int capacity,int fillratio);**
4. **Hashtable hm=new Hashtable(Map m);**

eg#1.

```
import java.util.Hashtable;
public  class TestApp {
	public static void main(String[] args){
		Hashtable ht= new Hashtable();
		ht.put(new Temp(5),"A");
		ht.put(new Temp(2),"B");
		ht.put(new Temp(6),"C");
		ht.put(new Temp(15),"D");
		ht.put(new Temp(23),"E");
		ht.put(new Temp(16),"F");
		System.out.println(ht); //{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}
	}
}

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

	@Override
	public int hashCode(){
		return i;
	}

	@Override
	public String toString(){
		return i+"";
	}


}
```

Scenario2:

if hashcode() is changed to

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

then output would be {16=F, 15=D, 6=C, 23=E, 5=A, 2=B}

Scenario3:

if hashcode() is changed to

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

but hashtable is created using the following constructor

```
Hashtable hm = new Hashtable(25);
```

Properties

==========

1. It is a child class of Hashtable
2. In a application, if a data changes frequently then those variables should not be hardcoded because as the value changes to the varaible we need to
   a. recompile
   b. rebuild
   c. redeploy the application which would have huge impact at buisness level.
3. To avoid this we need to hardcode this values at the properties file and read those values from the properties file to the java application so recompilation and rebuilding won't happen it is just redeployment.

4. To work with properties file at the java end we need to use "Properties" Object.

Constructor

==========

Properties properties=new Properites()

Note:: While creating a properties the key and value would always be in String format only.

```
String url="";//java level
```

application.properties

```
url=jdbc:mysql:///abc
username=root
password=root123
```

Methods associated with Properties Object

==========

1. public String getProperty(String keyName)
2. public void setProperty(String keyName,String value)
3. public void load(InputStream is)
4. public void store(OutputStream os,String data)
5. public Enumeration propertyNames()

eg#1.

```
import java.util.Properties;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Enumeration;

public  class TestApp {
        public static void main(String[] args) throws Exception{
                //Step1: Create a properties object to hold Properties file
                Properties p=new Properties();

        //Step2: Create a FileInputStream to bring the properties file into Properties Object
                FileInputStream fis =new FileInputStream("application.properties");
                p.load(fis);


           System.out.println(p);//{url=jdbc:mysql:///abc,username=root,password=root123}

                System.out.println("The url      is "+p.getProperty("url"));
                System.out.println("The username  is "+p.getProperty("username"));
                System.out.println("The password  is "+p.getProperty("password"));


                Enumeration e=p.propertyNames();
                while(e.hasMoreElements()){
                                String data=(String)e.nextElement();
                                System.out.println(data);
                }

                FileOutputStream fos= new FileOutputStream("application.properties");
                p.store(fos,"Thanks for giving the db infromation....");

        }
}
```

application.properties

==========

#Thanks for giving the db infromation....
#Thu Feb 03 10:59:51 IST 2022
password=root123
url=jdbc:oracle:thin:@localhost:1521:XE
username=System

eg#1.

```
import java.util.Properties;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Enumeration;
import java.sql.Connection;
import java.sql.DriverManager;


public  class TestApp {
	public static void main(String[] args) throws Exception{
		//Step1: Create a properties object to hold Properties file
		Properties p=new Properties();

	//Step2: Create a FileInputStream to bring the properties file into Properties Object
		FileInputStream fis =new FileInputStream("application.properties");
		p.load(fis);

		String url=p.getProperty("url");
		String username=p.getProperty("username");
		String password=p.getProperty("password");

	Connection connection=DriverManager.getConnection(url,username,password);
		System.out.println("Connection Established to "+url);
	}
}
```

Just by changing the properties file,same code can be used to communicate with multiple databases
through which we can say java slogan "WORA".

Q. What is the need of Generics in collections?

a. To promote type safety.

b. To avoid TypeCasting at the run time.

GeneralSyntax

==========

basetype<parametertype> ref=new basetype<parametertype>();

eg::

ArrayList<String> al =new ArrayList<String>();

|=> In this list object we can keep only String type of Objects.

Conclusion

==========

1.

List<String> al =new ArrayList<String>();

Collection<String> al =new ArrayList<String>();

List<Object> al =new ArrayList<String>();//invalid

Polymporphism concept is applicable only for basetype it is not applicable for parameter type.

2.

List<int> al =new ArrayList<int>();//invalid use Wrapper class.

Generic type should always be of referenc type like classname/interface name it can't be of primitive type,if we take it would result in "CompileTimeError".

Usage of Generics in Development

==========

This Generics concpet is not only applicable for predefined API,it is also applicable for user defined classes also.

eg#1.

```
class Gen<T>{
        T obj;

        Gen(T obj){
            this.obj=obj;
        }

        public void show(){
       System.out.println("The type of object is :: "+obj.getClass().getName());
        }

        public T getObject(){
            return obj;
        }
}
public  class TestApp {
       public static void main(String[] args){

            Gen<String> g1= new Gen<String>("sachin");
            System.out.println(g1.getObject());
            g1.show();

            System.out.println();

            Gen<Integer> g2= new Gen<Integer>(10);
            System.out.println(g2.getObject());
            g2.show();

            System.out.println();

            Gen<Double> g3= new Gen<Double>(10.5);
            System.out.println(g3.getObject());
            g3.show();


       }
}
```

output
sachin
The type of object is :: java.lang.String

10
The type of object is :: java.lang.Integer

10.5
The type of object is :: java.lang.Double

Generic Type parameter:-

Note:: Type parameter can be applied at
a. Class level
b. Method level

Class level
==========

|=> Type parameter
class Test<T>{
//here T can be of any Type as choosen by the programmer
}

Method level
============

It is defined to control the argument values of 'T'
public<T> void m1(T t){}//valid
public<T extends Number> void m1(T t){}//valid
public<T extends Number&Runnable> void m1(T t){}//valid
public <T extends Number&Runnable&Comparable> void m1(T t){} //valid
public <T extends Number&String> void m1(T t){} //invalid: 2 classes at a time
public <T extends Runnable&Number> void m1(T t){}//invalid: first class and then interface.

ConcurrentCollection

What is the need of ConcurrentCollection?

a. Traditional Collection Object like ArrayList,LinkedList,HashSet,HashMap is accessed by Multiple Threads simultaneously and there may be a chance of "DataIncosistencyProblem".
Since they are accessed by multiple threads simultaneously they are not "ThreadSafe".

b. Already Existed Collection Objects like a Vector,Hashtable ,synchronizedList(),synchronizedSet(),synchronizedMap() are "ThreadSafe", but performance is very low as the locking mechanism is not good.

c. In case of Normal Collection like ArrayList,.... if one thread is iterating on Collection Object and if Other thread tries to update the same Collection Object it would result in and exception called "ConcurrentModification".

Becoz of the above mentioned problem,these collection objects are not suitable for "MultiThreadingEnvironment".To resolve this problem SUNMS introduced a new Set of Collection called "ConcurrentCollection" in JDK1.5.

Note:
ConcurrentCollection is part of java.util.concurrent.*.

Proof of getting ConcurrentModificationException

==========

```
import java.util.ArrayList;
import java.util.Iterator;

public  class TestApp extends Thread {

        static ArrayList<String> al=new ArrayList<String>();

        @Override
        public void run(){
                try{
                        Thread.sleep(2000);//2sec
                }
```

```
            catch (InterruptedException e){}
            System.out.println("Child thread is updating the list");
            al.add("nanith");
        }
        public static void main(String[] args) throws Exception{
            al.add("sachin");
            al.add("kohli");
            al.add("ABD");

            TestApp t=new TestApp();
            t.start();

            Iterator itr=al.iterator();
            while(itr.hasNext()){
                String data =(String)itr.next();
            System.out.println("Main thread is iterating the list : "+data);
                Thread.sleep(3000);
            }

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

Output

```
Main thread is iterating the list : sachin
Child thread is updating the list
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:911)
    at java.util.ArrayList$Itr.next(ArrayList.java:861)
    at TestApp.main(TestApp.java:28)
```

The limitation of Traditional Collection is overcomed in ConcurrentCollections

1. Concurrent collection is ThreadSafe.
2. Performance of ConurrentCollection is better than NormalCollection becoz of different locking mechansim.
3. In case of ConcurrentCollection simultaneously read and update operation can be performed so their is no "ConcurrentModificationException".

ConcurrentCollection classes

==========

1. ConcurrentHashMap
2. CopyOnWriteArrayList
3. CopyOnWriteArraySet

ConcurrentMap(I)

It is a child interface of Map(I).

It has 3 methods

==========

public Object putIfAbsent(Object key,Object value);

//put the entry into the Map,only if key is not present. if key already exists then it wont replace the value associated with new key.

Difference b/w HashMap and ConcurrentHashMap?

HashMap => Not Thread safe.
ConcurrentHashMap => Thread safe because of differnt locking mechanism(bucket level lock)

HashMap => one thread is iterating and at the same time modification is not possible it would result in "ConcurrentModificationException".
ConcurrentHashMap => one thread is iterating and at the same time modification is possibleit won't result in "ConcurrentModif icationException".

HashMap => Performance is low becoz of more waiting time.
ConcurrentHashMap => Performance is high becoz of less waiting time.

HashMap => Iterator would result in "FailFast".
ConcurrentHashMap => Iterator is of "FailSafe".

HashMap => 1.2v
ConcurrentHashMap => 1.5v

Difference b/w ConcurrentHashMap,synchronizedMap and Hashtable?

ConcurrentHashMap => Thread safety by getting lock at the bucket level
SynchronizedMap => Thread safety by appling lock at object level
Hashtable => Thread safety by appling lock at object level

ConcurrentHashMap =>Since lock is at bucket level so no ConcurrentModificationException.
SynchronizedMap =>Since lock is Object level so ConcurrentModificationException.
Hashtable =>Since lock is at Object level so ConcurrentModificationException.

ConcurrentHashMap =>Iterator is "FailSafe".
SynchronizedMap =>Iterator is "FailFast".
Hashtable =>Iterator is "FailFast".

ConcurrentHashMap => Read operation is performed without lock,only for update operation lock is required only at the bucket level (concurrency level).
SynchronizedMap => Every read and write operation required lock.
Hashtable => Every read and write operation required lock.

ConcurrentHashMap =>null is not allowed for both key and value.
SynchronizedMap =>null is allowed for both key and value.
Hashtable =>null is not allowed for both key and value.

ConcurrentHashMap =>JDK1.5
SynchronizedMap =>JDK1.2
Hashtable =>JDK1.0

Difference b/w ArrayList and CopyOnWriteArrayList?

ArrayList => It is not Thread Safe.
CopyOnWriteArrayList => It is Thread Safe.

ArrayList => If one Thread,tries to read and another Thread tries to modify the Object it would result in "ConcurrentModificationException".
CopyOnWriteArrayList => If one Thread,tries to read and another Thread tries to modify the Object then no "ConcurrentModificationException".

ArrayList => Iterator is FailFast
CopyOnWriteArrayList => Iterator is FailSafe.

ArrayList => Iterator can be used to perform remove operation.
CopyOnWriteArrayList => If we use remove operation it would result in "UnsupportedOperation".

ArrayList => JDK1.2V
CopyOnWriteArrayList => JDK1.5v

Difference b/w CopyOnWriteArrayList,synchronizedList and Vector?

CopyOnWriteArrayList => It is Thread safe,as every update operation is performed on seperate cloned copy object
synchronizedList => It is Thread safe, where at a time only one thread is allowed to operate.
Vector => It is Thread safe, where at a time only one thread is allowed to operate.

CopyOnWriteArrayList => One thread performing read operation, simultaneously other thread can perform update operation it won't result in "ConcurrentModificationException".
synchronizedList => One thread performing read operation,simultaneously other thread can't perform update operation it would result in "ConcurrentModificationException".
Vector => One thread performing read operation,simultaneously other thread can't perform update operation it would result in "ConcurrentModificationException".

CopyOnWriteArrayList => Iterator is FailSafe.
synchronizedList => Iterator is FailFast.
Vector => Iterator is FailFast.

CopyOnWriteArrayList => Iterator can't perform remove operation it would result in "UnsupportedOperationException".
synchronizedList => Iterator can perform remove operation.
Vector => Iterator can perform remove operation.

CopyOnWriteArrayList => JDK1.5V
synchronizedList => JDK1.2V
Vector => JDK1.0V

Difference b/w CopyOnWriteArraySet and synchronizedSet?

CopyOnWriteArraySet => Thread safety as every update operation is performed on cloned copy of Object.
synchronizedSet => Thread safety becoz at a time only one thread is allowed to operate on a Object.

CopyOnWriteArraySet => If one thread is iterating on a object,if other thread tries to change the structure of the Collection it won't result in "ConcurrentModification Exception".

synchronizedSet => If one thread is iterating on a object,if other thread tries to change the structure of the Collection it would result in "ConcurrentModification Exception".

CopyOnWriteArraySet => Iterator is FailSafe.
synchronizedSet => Iterator is FaileFast.

CopyOnWriteArraySet => Iterator can perform read only and update operation,if it tries to perform remove operation it would result in "UnSupportedOperationException".

synchronizedSet => Iterator can read and remove operation.
CopyOnWriteArraySet => JDK1.5V
synchronizedSet => JDK1.2V

Q) What is the difference between failfast and failsafe?

Failfast	Failsafe
It does not allow collection modification while iterating.	It allows collection modification while iterating.
It can throw ConcurrentModificationException	It can't throw any exception.
It uses the original collection to traverse the elements.	It uses an original collection copy to traverse the elements.
There is no requirement of extra memory.	There is a requirement of extra memory.

Difference b/w FailFast and FailSafe?

Property	FailFast	FailSafe
Does it throw ConcurrentModificationExcetpion	Yes	No
Cloned copy will be created	No	Yes
Memory problems	No	Yes
Example	ArrayList,Vector,...	CopyOnWriteArrayList, ConcurrentHashMap,...