Arrays	Collections
Arrays are fixed in size. Can not be increase at runtime as per need	Collections are growable in nature
Memory point of view – not recommended	Recommended for better memory utilisation compared to array
Performance point of view – recommended as faster	Not recommended
Only Homogenous data stores	Both homogenous + Heterogenous data
No readymade method support as no underlying DS, hence implementation code Written by developer	Here readymade method support available as each collection type has underlying DS implementation ,hence method directly used (no need to implement)
Arrays can hold both primitive and non-primitive datatypes(objects)	Collections can only hold non-primitive(objects) data i.e. Integer accepted but not int

WHAT IS COLLECTION - group of objects as a single entity

WHAT IS COLLECTION FRAMEWORK - The Java collections framework is **a set of classes and interfaces** that implement **commonly reusable collection data structures.**

In java	In C++
Collections	Container
Collection framework	STL → standard template library

Interface provides more information about specs than classes

9 Key interfaces important in collection framework.

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

Collection	Collections
It is a interface	It is a class
It represents group of individual objects as a single entity	It is a utility class defines several utility methods for collection object such as sorting and searching operations Eg. Collections.sort(al);

1) Collection(I) – 1.2version
   Root interface for collection framework

   If we want to represent group of individual objects as a single entity then go for collection

   This interface defined most common methods require for any collection object

   Note * - no concrete class implements this interface directly**

2) List(I) – 1.2version
   Child interface /Sub interface of Collection

   When should one go for List interface
   Group of individual objects as a single entity where

   - Duplicates are allowed and
   - Insertion order must be preserved

   Implementation classes – ArrayList LinkedList
   Vector and Stack (Legacy classes)

3)Set(I) – 1.2 version

Child interface of Collection(I)

When should one go for List interface
Group of individual objects as a single entity where

- Duplicates are NOT allowed and
- Insertion order NOT preserved

Implemented class – HashSet(1.2v) and LinkedHashSet(1.4v).

4) SortedSet - (1.2v)

Child interface of Set(I)

Used where all objects should be inserted according to some sorting order

5) Navigable set –(1.6v)

Child interface of SortedSet

It contains several methods for navigation purposes

Implementation class TreeSet.

List	**Set**
Duplicates allowed.	Dups not allowed
Insertion order preserved	Insertion order not preserved

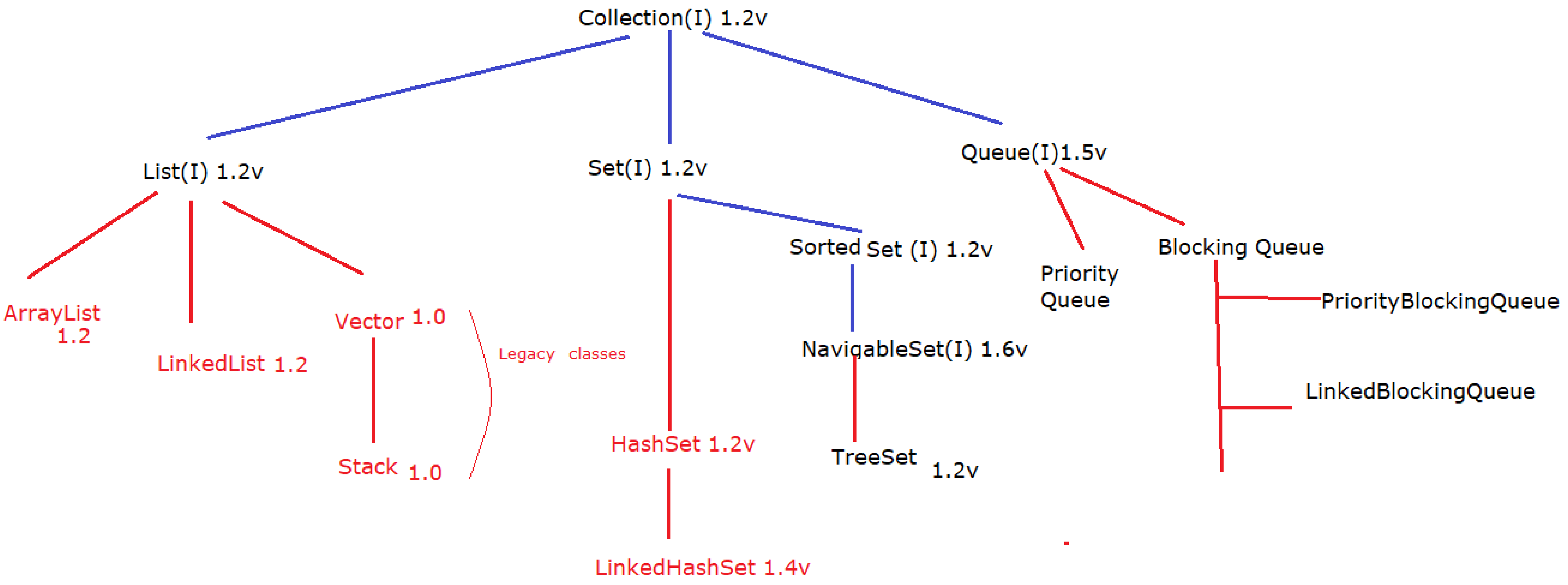
6) Queue (I) - 1.2v

Child Interface of Collection

FIFO

To represent group of individual objects *prior to processing* then we go for Queue for FIFO

But based on req. we can implement our priority order also



Map (I) – Map is **<u>NOT</u>** a child interface of collection

To represent object as a key-value pair then Map is used

Group of key-value pairs

Dups keys are NOT allowed but duplicate values are allowed

SortedMap(I) – child interface of Map

If we want to represent group of key value pairs according to some sorting order based on Key then we should go for sortedMap

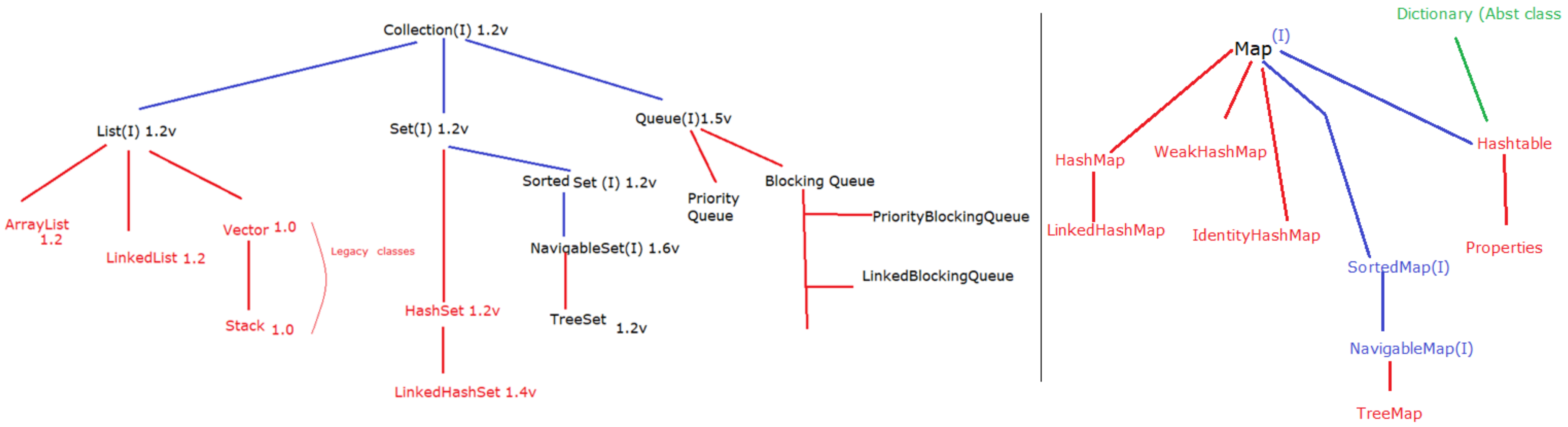
Sorting based on Key NOT based on value.

Navigable(I) – Child Interface of SortedMap

It defines several methods for Navigation purposes (1.6v)

Implementation class – TreeMap (1.2v)

Diagram to draw or able to visualise when asked what is Collection Framework.



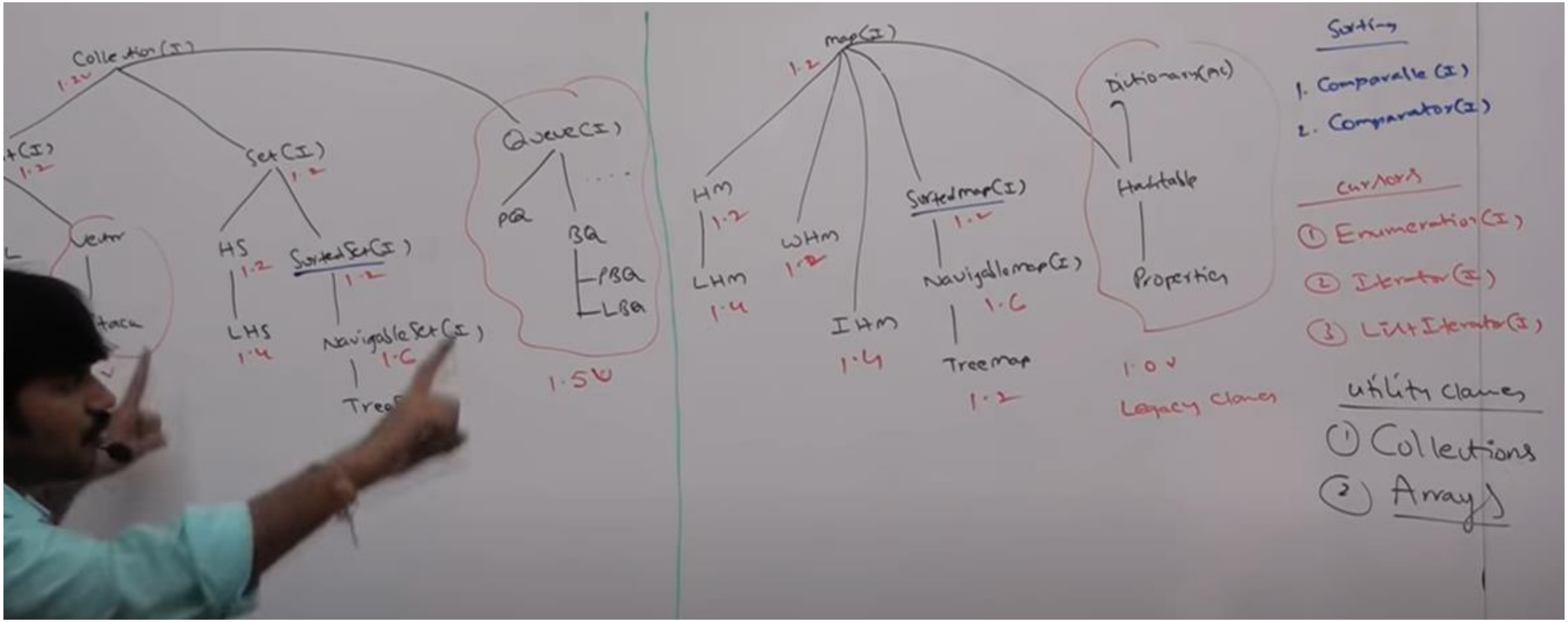
Sorting - Comparable
Comparators

Cursors - Enumeration
Iterators
ListIterators

Utility Classes - Collections
Arrays

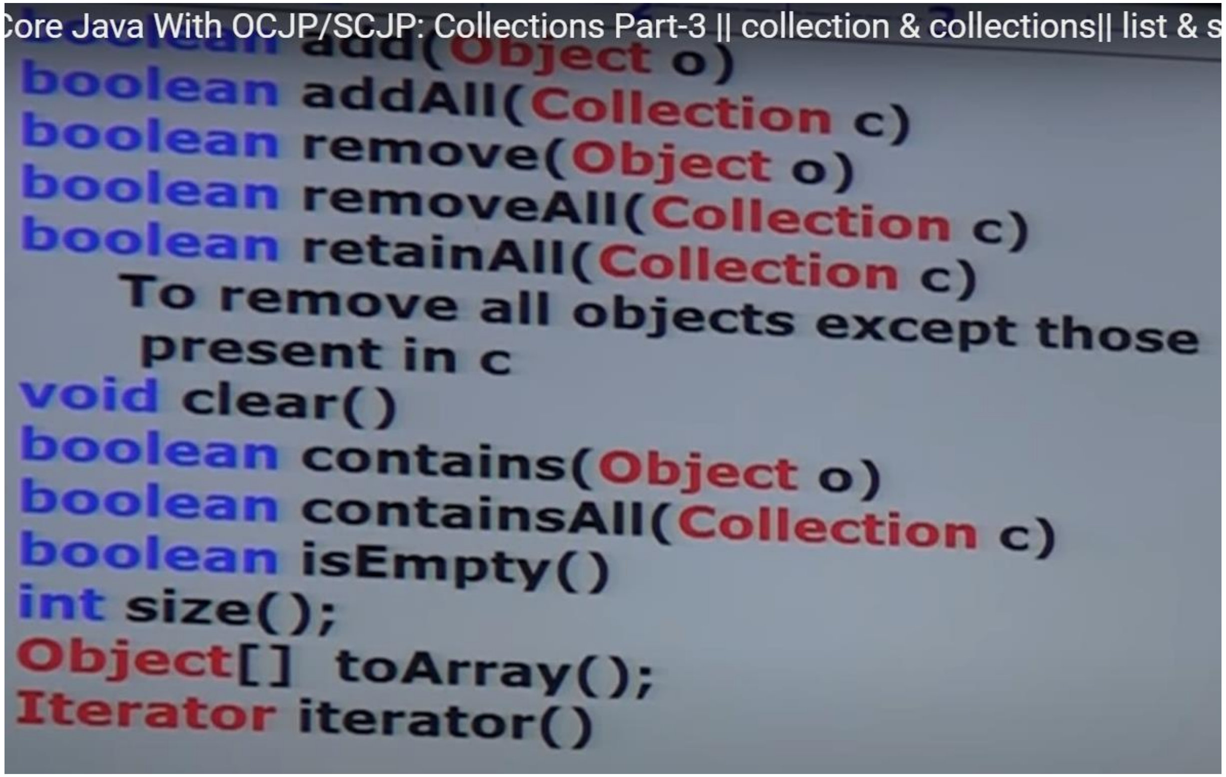
6 Legacy characters which introduced in
Vector(c)
Stack(c)
Dictionary(Abstract class)
Hashtable(c)
Properties(c)

Enumeration(I) (in cursors)



1.20.33 in Core Java With OCJP/SCJP: Collections Part-2 || 9key interfaces

Lecture 3 : Collection interface methods



Core Java With OCJP/SCJP: Collections Part-3 || collection & collections|| list & s

```
boolean add(Object o)
boolean addAll(Collection c)
boolean remove(Object o)
boolean removeAll(Collection c)
boolean retainAll(Collection c)
    To remove all objects except those
     present in c
void clear()
boolean contains(Object o)
boolean containsAll(Collection c)
boolean isEmpty()
int size();
Object[]  toArray();
Iterator iterator()
```

Contains,add,isEmpty , toArray , iterator , retainall method.

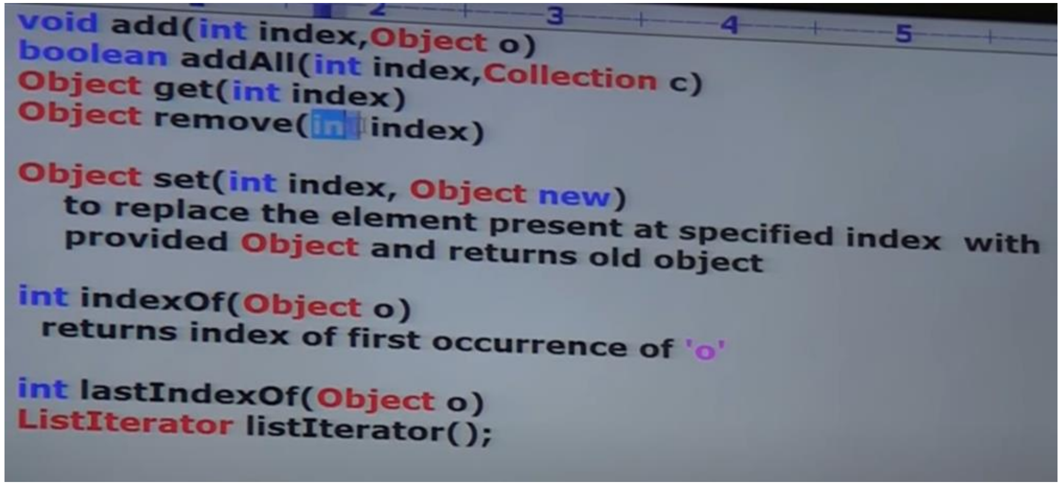
Collection has no implementation classes

List Interface :

To represent group of individual object as a single entity where duplicates allowed and insertion preserved.

Index has importance here for preserving insertion order and differentiate between duplicate entries

Methods in List (I)



```
void add(int index,Object o)
boolean addAll(int index,Collection c)
Object get(int index)
Object remove(int index)

Object set(int index, Object new)
    to replace the element present at specified index  with
    provided Object and returns old object

int indexOf(Object o)
  returns index of first occurrence of 'o'

int lastIndexOf(Object o)
ListIterator listIterator();
```

set method to replace the element present at specified index

add, get, remove, indexOf , lastIndexOf, addAll,removeAll etc.

\Implementation classes

ArrayList :

Resizable array or growable Array is underlying DS

Duplicate allowed, Insertion order preserved

Heterogenous object are allowed

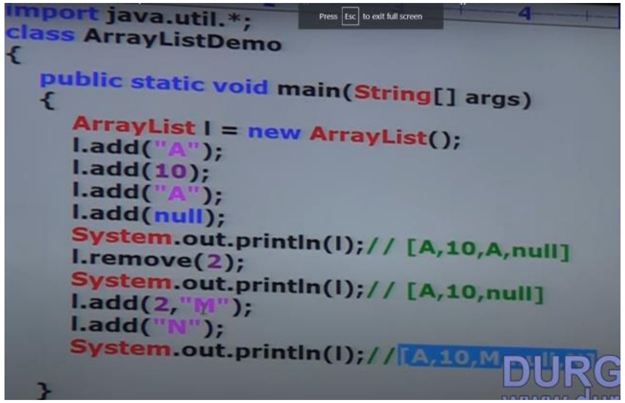
null insertion is possible

Constructors :

ArrayList l = new ArrayList();

ArrayList l = new ArrayList(int initialcapacity);

ArrayList l = new ArrayList(Collection c); //to convert LinkedList,treeset,Vector to AL.



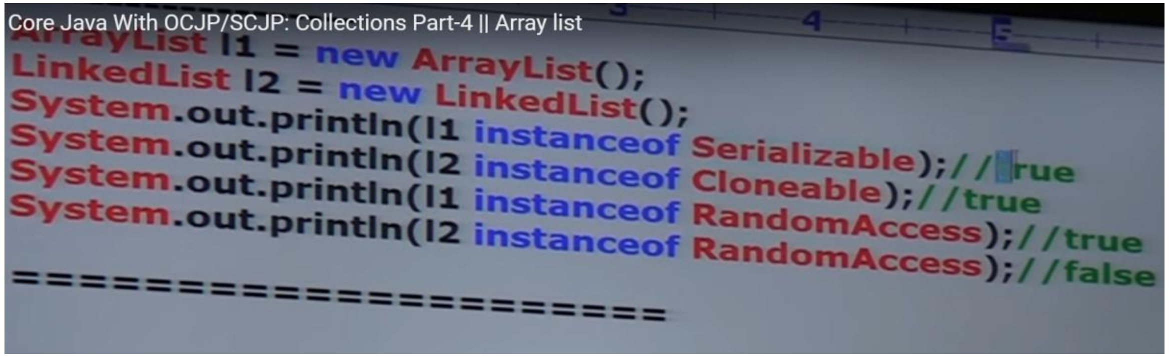
```
import java.util.*;
class ArrayListDemo
{
   public static void main(String[] args)
   {
      ArrayList l = new ArrayList();
      l.add("A");
      l.add(10);
      l.add("A");
      l.add(null);
      System.out.println(l);// [A,10,A,null]
      l.remove(2);
      System.out.println(l);// [A,10,null]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

ArrayList<String> l = new ArrayList<>(); *// to avoid warnings used generics with AL.*

l.remove(2) // removes from index 2

l.add(2,"M") // add M at index 2

Lecture 4 --- **ArrayList**



Core Java With OCJP/SCJP: Collections Part-4 || Array list

```
ArrayList l1 = new ArrayList();
LinkedList l2 = new LinkedList();
System.out.println(l1 instanceof Serializable);//true
System.out.println(l2 instanceof Cloneable);//true
System.out.println(l1 instanceof RandomAccess);//true
System.out.println(l2 instanceof RandomAccess);//false
=====================
```

Notes :

All Collection i/f implemented classes implements serializable and cloneable interface <u>to hold and transfer object AND creating clone of object respectively</u>

Only ArrayList and Vector implements RandomAccess I/f which is marker interface

Ensures constant time for searching/accessing element

ArrayList is BEST choice if our frequent operation is retrieving operation because of RandomAccess I/f

Default init capacity 10 next capacity 10*3/2 +1 → 10+6 =16 next → 25 ….etc.

AL is worst choice when insertion and deletion in the middle **Hence we use LinkedList for it but retrieval O(n) time for LL**

Differences between AL an vector

ArrayList	Vector
Non-Synchronised methods	Synchronised methods
Multiple threads allowed to operate on obj hence thread UNSAFE	Only single thread to operate on vector obj hence thrd SAFE
Relatively faster performance/ or high	Slow performance as thrds have to wait
Introduced in 1.2v Non legacy	Legacy class as introduced in 1.0 v

Interview – how to get synchronised version of arraylist object??

AL<T> l = new AL<>(); //non sync.

List<T> l1 = Collection.sychronisedList(l); // synchronized version of arraylist object by using synchronisedList method of collections class.

Public static List synchronisedList(List l)

Public static Set synchronisedSet(Set s) //similar methods for set and map

Public static Map synchroniseMap(Map m)

---------------------000--------------------------------------000---000-----------------------------

LinkedList :

- Underlying ds is doubly linked list
- Insertion order preserved , dups allowed
- Heterogeonous objects allowed
- Null insertion possible
- LL implements Serializable n Cloneable i/f but NOT RandomAcess
- If our freq operation is insertion and deletion LL best choice
- If our freq operation is retrieval then LL worst choice.

Constructors

LinkedList l = new LinkedList(); *//empty LL obj*

LinkedList l = new LinkedList(Collection c) *// LL obj with collection c*

Program ---Array or LinkedList based implementation of stack n queue

In LL – methods

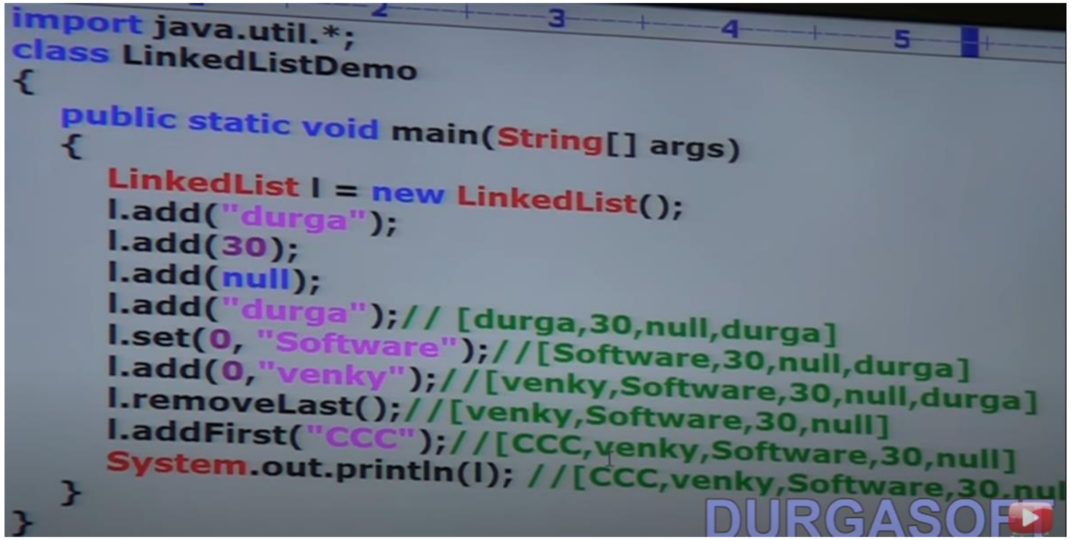
addFirst(Object o) ;   removeFirst();

addLast(Object o) ;   removeLast();

getFirst();

getLast();

```
psvm(String [] args){

        LinkedList l = new LinkedList();
        l.add("durga");
        l.add("bhavani");
        l.add("98");
        l.add("10.5 LPA ");[durga, bhavani ,98 ,10.5LPA]
        l.set(0,"soft"); //replace 0th elem durga with soft.
        l.add(0,"ware"); //add ware at 0th position NOT replace
        l.addFirst("hard");[hard ,ware , soft, bhavani ,98 ,10.5LPA]
        l.removelast();
        SOP(l) ;  // [hard ,ware , soft, bhavani ,98 ]
}
```



```
import java.util.*;
class LinkedListDemo
{
   public static void main(String[] args)
   {
      LinkedList l = new LinkedList();
      l.add("durga");
      l.add(30);
      l.add(null);
      l.add("durga");// [durga,30,null,durga]
      l.set(0, "Software");//[Software,30,null,durga]
      l.add(0,"venky");//[venky,Software,30,null,durga]
      l.removeLast();//[venky,Software,30,null]
      l.addFirst("CCC");//[CCC,venky,Software,30,null]
      System.out.println(l); //[CCC,venky,Software,30,nul
   }
}
```

ArrayList	LinkedList
Freq operation retrieval then preferred	Best choice for freq insertion and deletion in the middle
Worst for insertion /del in middle because internal shifiting operation performed	Worst choice for freq retrieval as O(n) time for nth retrieval
Consecutive mem location	Not consecutive mem loc.
RandomAccess marker I/f implemented	No randomaccess concept here

---------------------000-------------------------------------000---000------------------------

Note : in TreeSet and TreeMap heterogenous object are not allowed. As for sorting logic- object should be of same type for comparison.

Vector :

- Resizable /growable array
- Insertion order preserved and dups allowed
- Heterogenous objects allowed
- Null insertion possible
- Serializable Cloneable and RandomAccess
- Synchronised and hence thrd safe.

Constructors :

1. Vector v = new Vector(); // blank vector

Default capacity → 10 blocks

Double capacity once vector reaches max capacity

New max capacity = 2 current capacity*

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

to set increment in new capacity

3. Vector v = new Vector(int initialCapacity , int incrementalCapacity);

Vector v = new Vector(1000,10); //this is not in ArrayList

4. Vector v = new Vector(Collection c); //interconversion between collection objects

Methods :

Old versions legacy class → lengthy method names

addElement(Object o);

removeElement();

removeElementAt(int index);

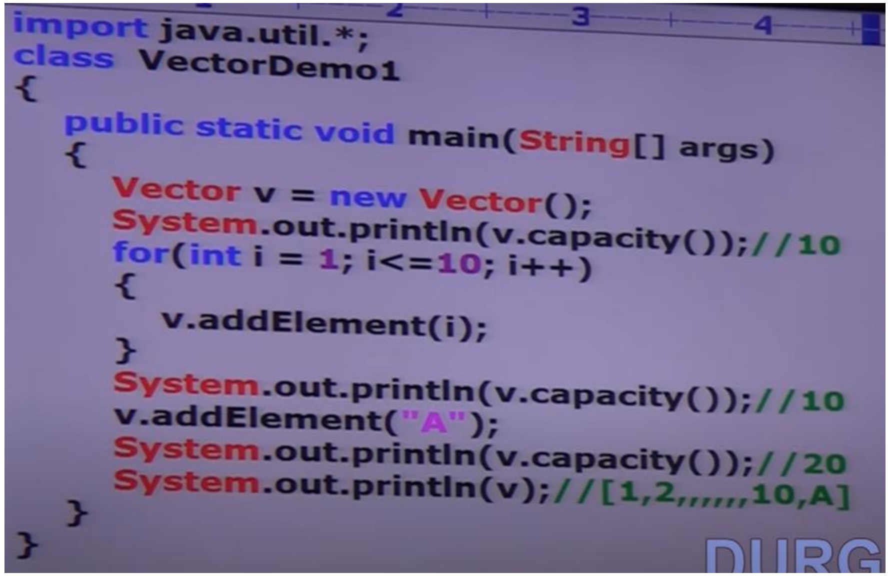
removeAllElements();

Default init capacity in AL 10 next capacity 10*3/2 +1 → 10+6 =16 next → 25 ….etc.

Default init capacity in V 10 next capacity 10*2 → 20 next → 30 ….etc.

Vector v = new Vector(10,5); → next capacity 15 next 20 ,25 so on….

v.capacity() ; to show current capacity …no such method In AL.



```
import java.util.*;
class VectorDemo1
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        System.out.println(v.capacity());//10
        for(int i = 1; i<=10; i++)
        {
            v.addElement(i);
        }
        System.out.println(v.capacity());//10
        v.addElement("A");
        System.out.println(v.capacity());//20
        System.out.println(v);//[1,2,,,,,,10,A]
    }
}
```

Stack extended by Vector

Methods :

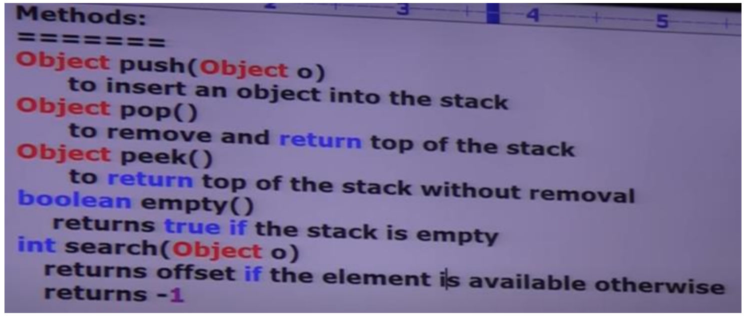
Void Push(Object o);

Object Pop();

Object Peek();

boolean empty();

int search(Object o); *returns offset if present otherwise -1 if absent*



```
Methods:
=======
Object push(Object o)
    to insert an object into the stack
Object pop()
    to remove and return top of the stack
Object peek()
    to return top of the stack without removal
boolean empty()
   returns true if the stack is empty
int search(Object o)
   returns offset if the element is available otherwise
   returns -1
```

```
psvm(String [] args){

        Stack s  = new Stack();
        s.push("X");
        s.push("Y");
        s.push("Z");
        s.push("a");
        sop(s); //[X,Y,Z,a]
        //if popped
        s.pop(); //[X,Y,Z] ----LastInFirstOut
        SOP(s.search("X")); //3
        SOP(s.search("M")); //-1
}
        offaset  | |Index {offeset n index different n ulta)
                1|Z|2
                2|Y|1
                3|X|0
```

Insertion order is preserved hence sequence same as input

Cursors :

Property	Enumeration	Iterator	ListIterator
Where we can apply?	Legacy class (Vector ,stack)	For any collection object	Only for List object
Is it legacy?	Yes	No	No
Movement?	Single Direction(forward)	Single Direction(forward)	Bidirectional
Allowed operation?	read	read ,remove	read, remove, add ,replace
How we can get?	By using elements method of vector class	By iterator method of Collection I/f	By ListIterator method of List I/f.

Methods?	2 methods
	hasMoreElements()
	nextElement()
		3 methods
		hasNext()
		next()
		remove()
			9 methods

To get object from the collection one by one at a time cursors introduced in collection framework

Three types of Cursors

Enumeration(legacy Interface)

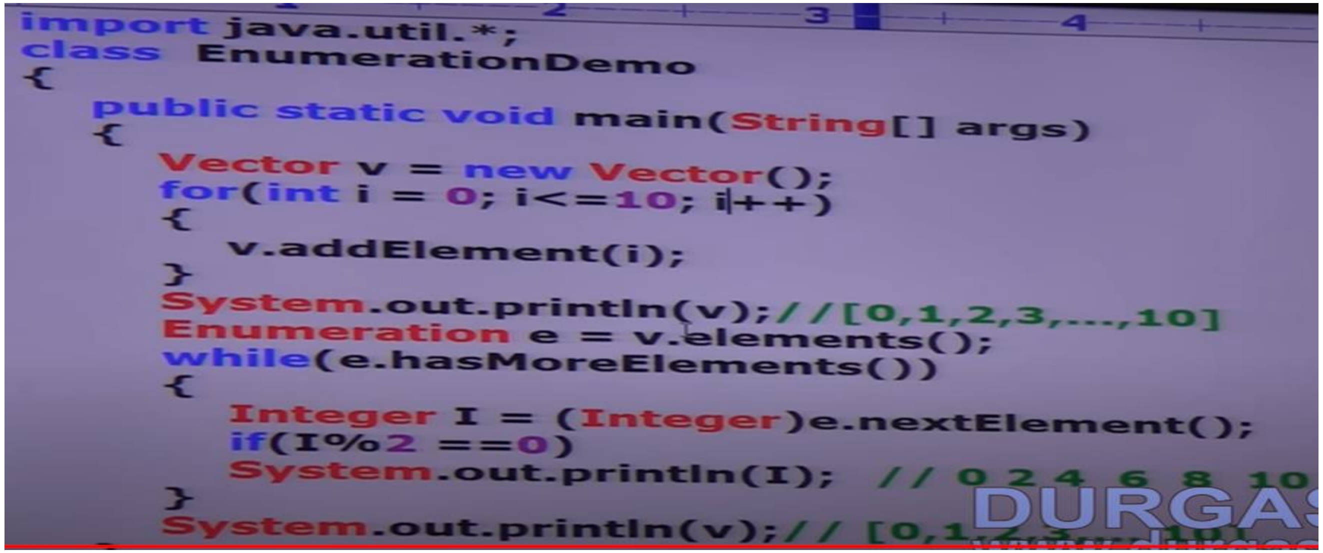
Iterator(I)

ListIterator(I)

Enumeration(legacy Interface) – used to get objects one by one from Legacy collection object (here vector object)

We can create enumeration object using elements() method of vector class

1. public Enumeration elements()
2. Enumeration e = v.elements(); here v is vector object
3. public boolean hasMoreElements(); public Object nextElement();



```
import java.util.*;
class  EnumerationDemo
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        for(int i = 0; i<=10; i++)
        {
            v.addElement(i);
        }
        System.out.println(v);//[0,1,2,3,...,10]
        Enumeration e = v.elements();
        while(e.hasMoreElements())
        {
            Integer I = (Integer)e.nextElement();
            if(I%2 ==0)
            System.out.println(I);  // 0 2 4 6 8 10
        }
        System.out.println(v);// [0,1,2,3,...10]
```

Limitations -

1. **only applicable for legacy classes such as Vector and Stack not universal cursor like Iterator**
2. **have only read access not remove operation .** In Iterator also have remove Capability.
   To overcome above limitation we go for Iterator

Iterator(I) →

- **we can apply it for any collection object Hence → Universal cursor**
- **By using iterator we can perform both read and remove operations.**

Public Iterator iterator(); //we create iterator object by using iterator method of collection interface

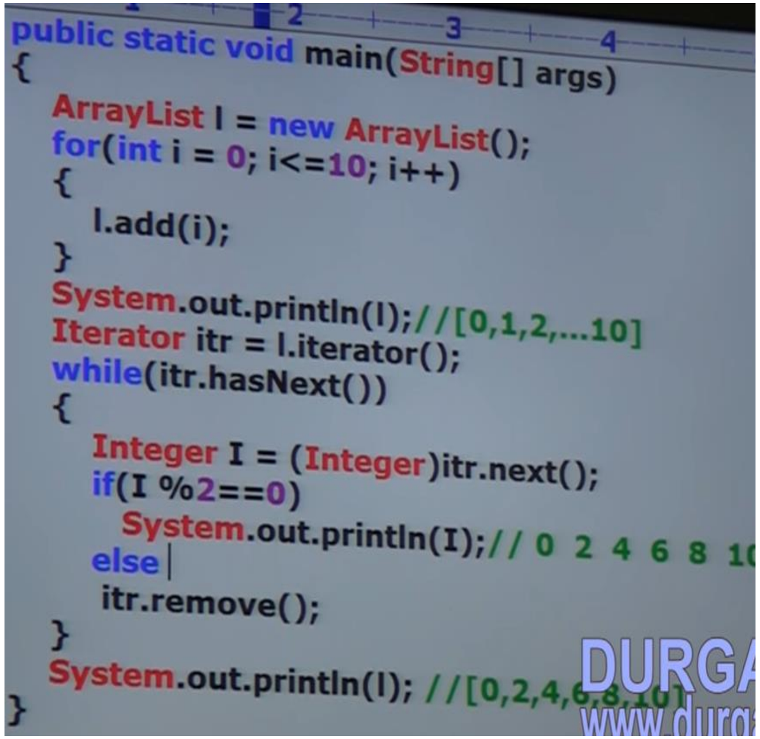
Iterator itr = c.iterator() ; where c is any collection obj.

METHODS :

- public boolean hasNext();

- public Object next();
- public void remove();

```
ArrayList l = new ArrayList();

for(int i = 0 ; i<=10 ; i++){
        l.add(i);
}
sop(i ); //[0 , 1 ....., 10]

//Iterator usage
Iterator itr = l.iterator();
while(itr.hasNext())
{
        Interger I = (Integer) itr.next();
        if(I%2 == 0)
        SOP(I); // 0 2 4 ...10
        else itr.remove;
}
SOP(i) // [0 , 2 , 4 ..., 10] odd numbers removed unlike enumeration
```



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

Limitations :

1. moves towards forward directions only i.e unidirectional cursor
2. we can perform only read and remove operations .not replace or add new object possible

For overcoming above limitation ListIterator can be used.

ListIterator(I) :

Child interface of iterator and hence all methods present in Iterator bydefault

Available to ListIterator

Iterator(I) ← ListIterator(I)

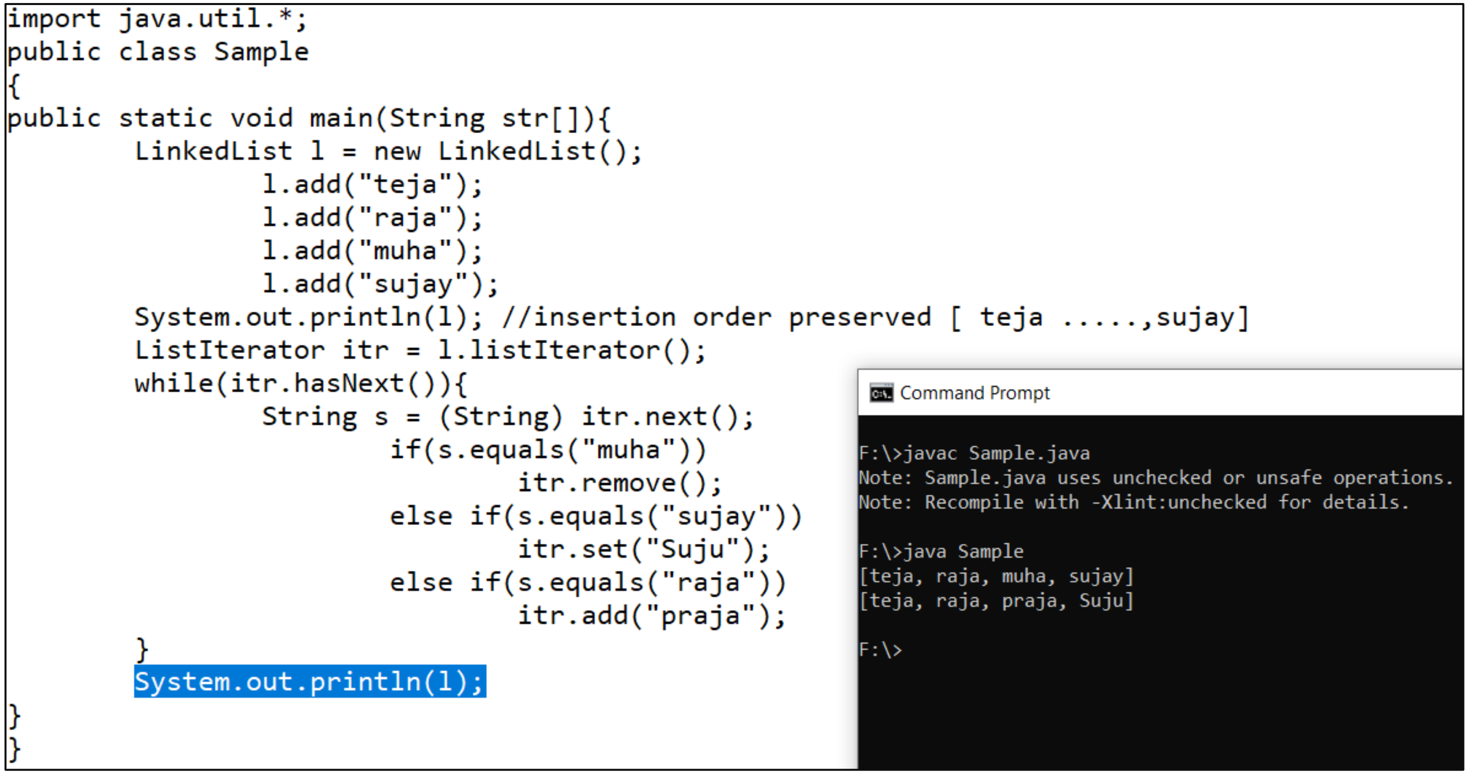
- Bidirectional cursor → forward + backward movement.
- Can perform replacement and addition of new objects in addition to read and remove operations
- **Most Powerful cursor but its limitation is it is applicable only for list objects**

Public ListIterator listIteractor();

ListIterator itr = l.listIteractor(); //l is any List object

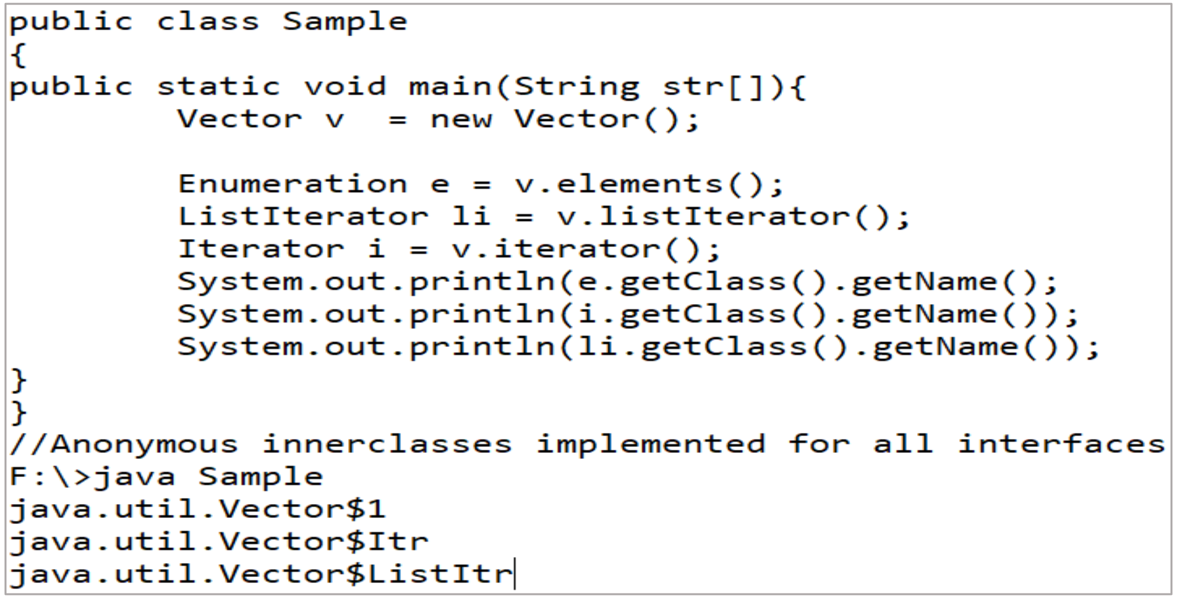
9 methods given for ListIterator

1. boolean hasNext()
2. Object next() providesd for forward operation
3. int nextIndex();
4. boolean hasPrevious();
5. Object previous(); provided for backward movement
6. int previousIndex();
7. remove();
8. add(Object o);
9. set(Object o);



```
import java.util.*;
public class Sample
{
public static void main(String str[]){
        LinkedList l = new LinkedList();
                l.add("teja");
                l.add("raja");
                l.add("muha");
                l.add("sujay");
        System.out.println(l); //insertion order preserved [ teja .....,sujay]
        ListIterator itr = l.listIterator();
        while(itr.hasNext()){
                String s = (String) itr.next();
                        if(s.equals("muha"))
                                itr.remove();
                        else if(s.equals("sujay"))
                                itr.set("Suju");
                        else if(s.equals("raja"))
                                itr.add("praja");
        }
        System.out.println(l);
}
}
```

```
F:\>javac Sample.java
Note: Sample.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

F:\>java Sample
[teja, raja, muha, sujay]
[teja, raja, praja, Suju]

F:\>
```



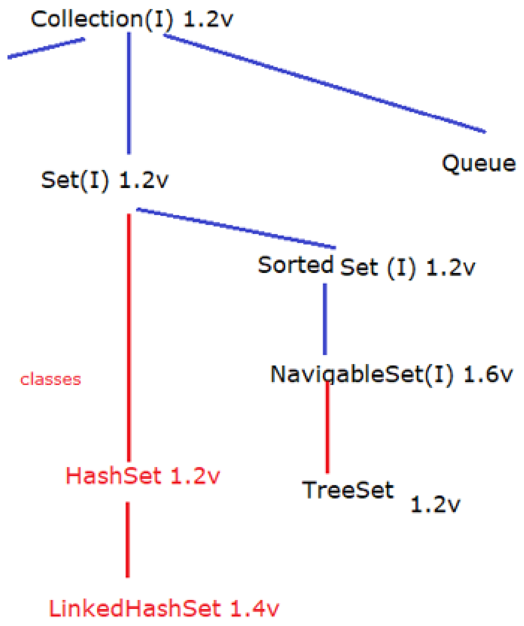
```
public class Sample
{
public static void main(String str[]){
        Vector v  = new Vector();

        Enumeration e = v.elements();
        ListIterator li = v.listIterator();
        Iterator i = v.iterator();
        System.out.println(e.getClass().getName();
        System.out.println(i.getClass().getName());
        System.out.println(li.getClass().getName());
}
}
//Anonymous innerclasses implemented for all interfaces
F:\>java Sample
java.util.Vector$1
java.util.Vector$Itr
java.util.Vector$ListItr
```

Lecture Part -7

Set Interface Tree →

Group of individual object as a single entity where **Dups not allowed and insertion order not preserved.**

12 methods already in Collection.



Set does not have any new methods and we only going to use Collection methods.

HashSet :

1. underlying DS is hashtable
2. Hashing related DS - *all objects inserted based on hashcode*
3. Dups not allowed , heterogenous objects allowed
4. Null insertion possible only once
5. Serializable Cloneable implemented but not RandomAccess
6. If frequent Searching operation required we go for hashset

In hashset dups not allowed but if we tried then we don't get any CE or run time errors

Add method simply returns false

HashSet hs = new HashSet();

SOP(hs.add("A")); // true

SOP(hs.add("A")); // false

Hashing related DS – Hashset, LinkedHashSet, HashMap, LHM, WeakHM, IdentityHM All having similar kind of ctors

Constructors :

HashSet hs = new HashSet();

// default initial capacity 16, default fill ratio / Load factor → 0.75

What is load factor or fill ratio ? → after loading/filling that much capacity increased capacity of hashset

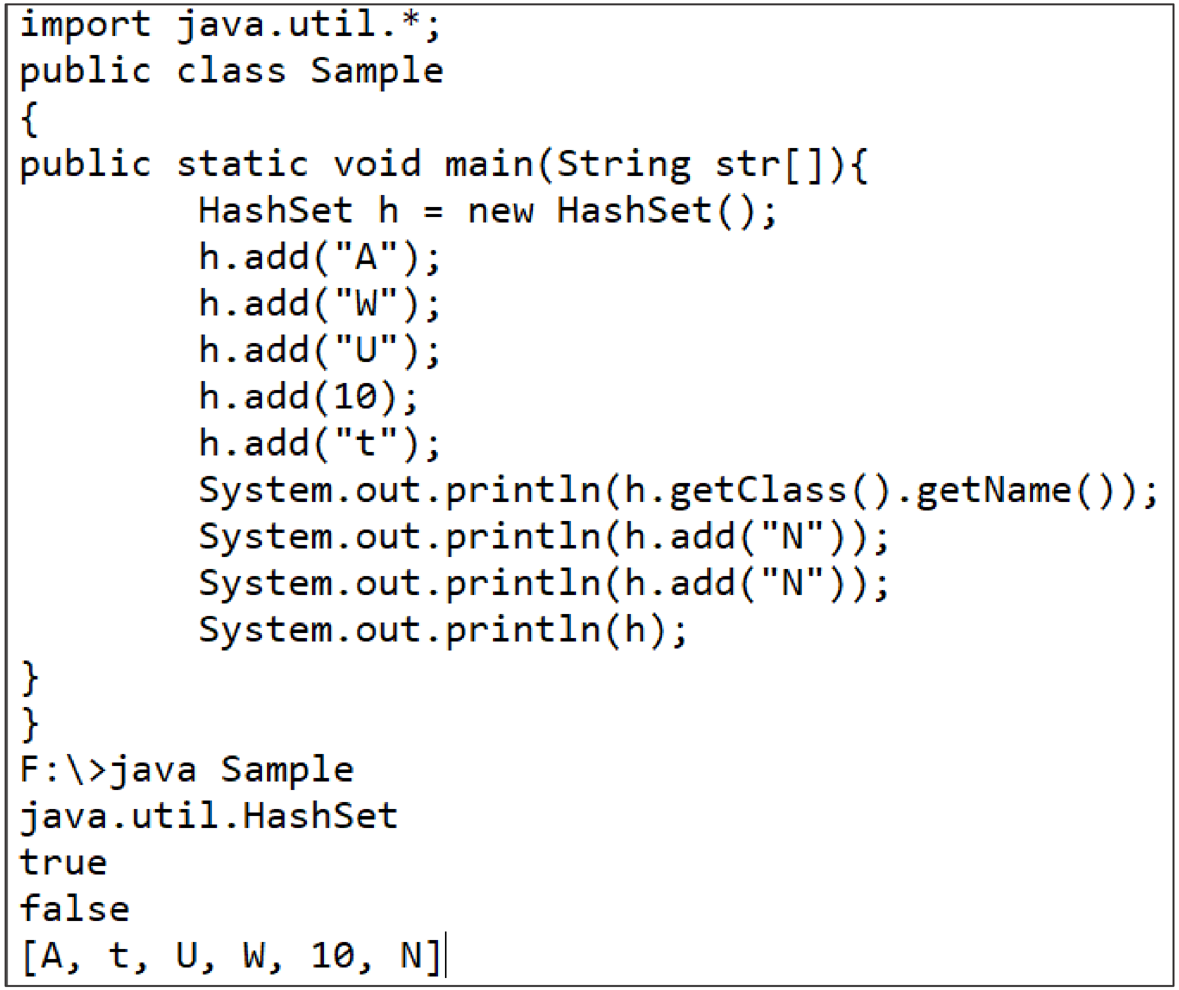
HashSet hs = new HashSet(int initialCapacity); //creates empty hs object with specified init capacity ,fill ratio def – 0.75

HashSet hs = new HashSet(int initialCapacity , float fillRatio);

HashSet hs = new HashSet(Collection c); //creates equivalent hs for the given collection. This Is for interconversion between collection object

Fill ratio or Load Factor : after loading how much ratio or factor new HS created….0.75 → after filling 75% of hashset new hashset object created

Example :



```
import java.util.*;
public class Sample
{
public static void main(String str[]){
        HashSet h = new HashSet();
        h.add("A");
        h.add("W");
        h.add("U");
        h.add(10);
        h.add("t");
        System.out.println(h.getClass().getName());
        System.out.println(h.add("N"));
        System.out.println(h.add("N"));
        System.out.println(h);
}
}
F:\>java Sample
java.util.HashSet
true
false
[A, t, U, W, 10, N]
```

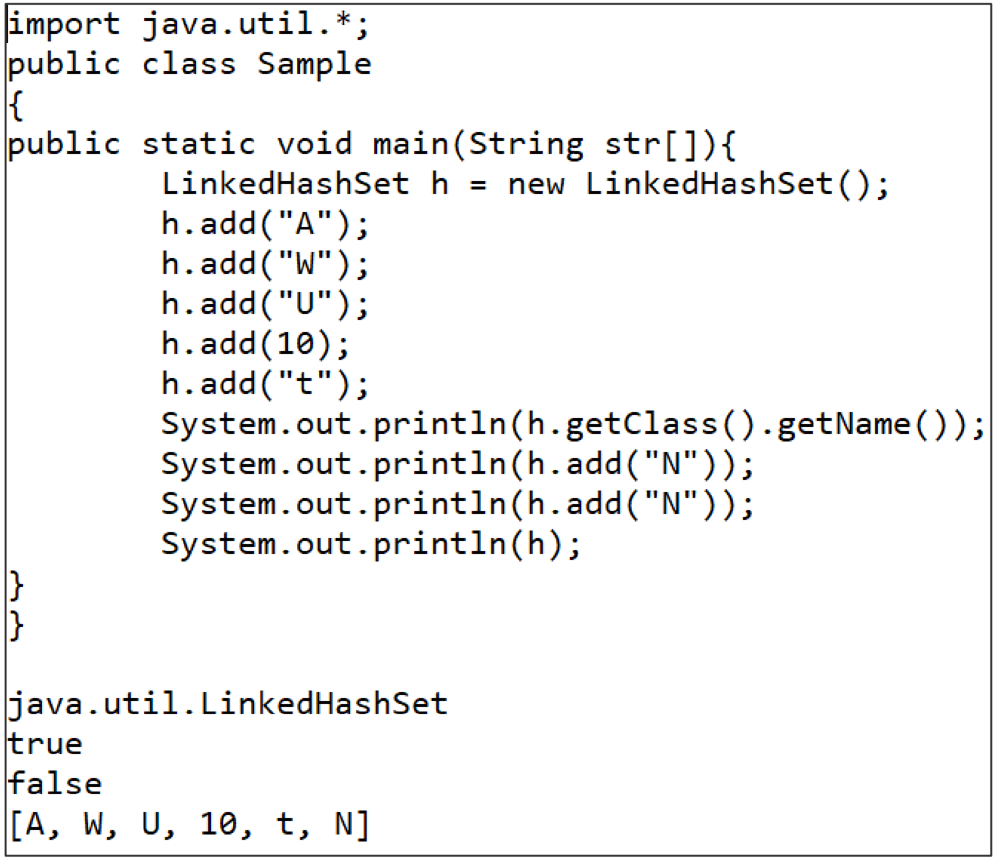
LinkedHashSet :

HashSet	LinkedHashSet
Underlying DS → Hashtable	Combination of LinkedList + Hashtable
Insertion order not preserved	Insertion order preserved
Introduced in 1.2 version	Introduced in 1.4 version

- Child class of HashSet
- exactly same as HashSet including ctors n methods. Except above differences

example :

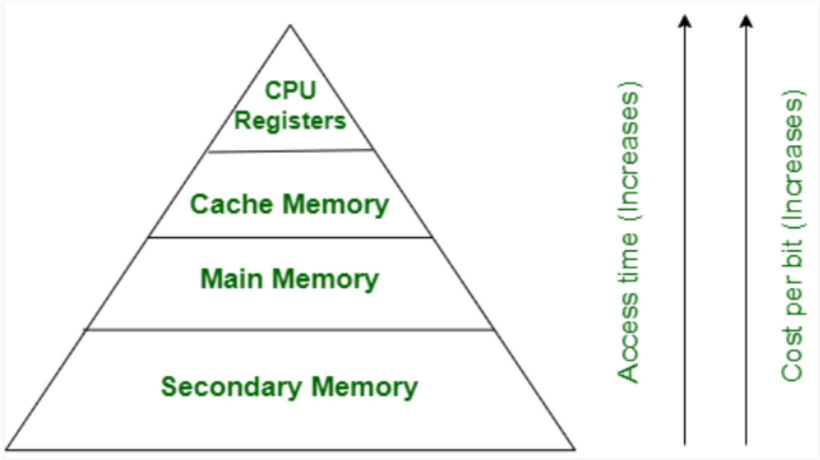
note to insertion order



```
import java.util.*;
public class Sample
{
public static void main(String str[]){
        LinkedHashSet h = new LinkedHashSet();
        h.add("A");
        h.add("W");
        h.add("U");
        h.add(10);
        h.add("t");
        System.out.println(h.getClass().getName());
        System.out.println(h.add("N"));
        System.out.println(h.add("N"));
        System.out.println(h);
}
}

java.util.LinkedHashSet
true
false
[A, W, U, 10, t, N]
```

Primary memory (registers ,registers,ram)➔ Primary memory is the computer memory that is directly accessible by CPU

They can be represented in an hierarchical form as:



In general, we use LinkedHashSet to developed cachebased applications where Dups not allowed and insertion order is preserved.

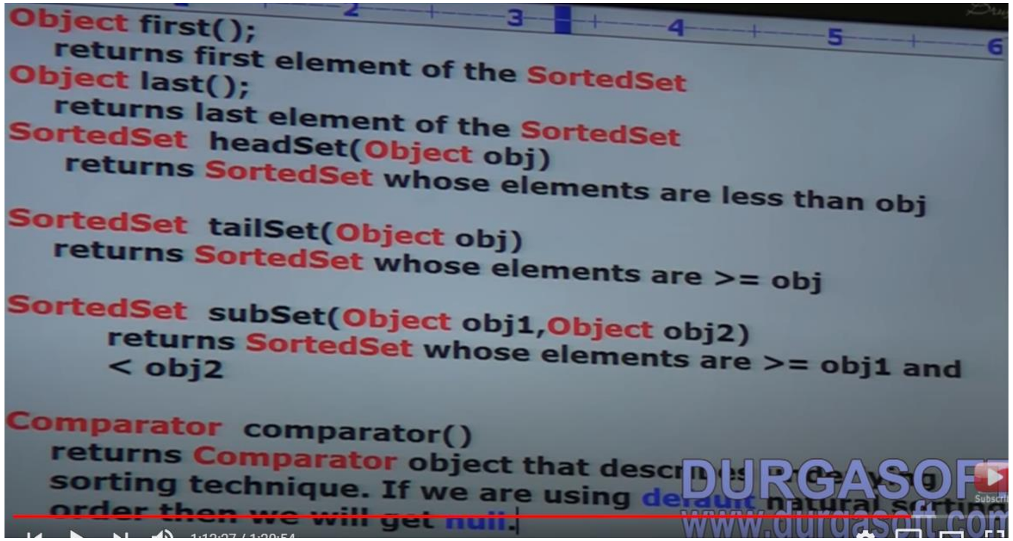
SortedSet –

1. child i/f of Set
2. if we want to represent group of individual objects according to some sorting order without dups then prefer sortedset.

Methods :

[100 101 104 106 110 115 120]

i. first() ➔ 100
ii. last() ➔ 120
iii. headSet(106) ➔ [100 , 101 ,104]
iv. tailSet(106) ➔ [110 , 115 , 120]
v. subSet(106,120) ➔ [110 ,115]
vi. comparator() ➔ natural sorting order



Object first();
returns first element of the SortedSet

Object last();
returns last element of the SortedSet

SortedSet headSet(Object obj)
returns SortedSet whose elements are less than obj

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

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

Comparator comparator()
returns Comparator object that describes underlying sorting technique. If we are using default natural sorting order then we will get null.

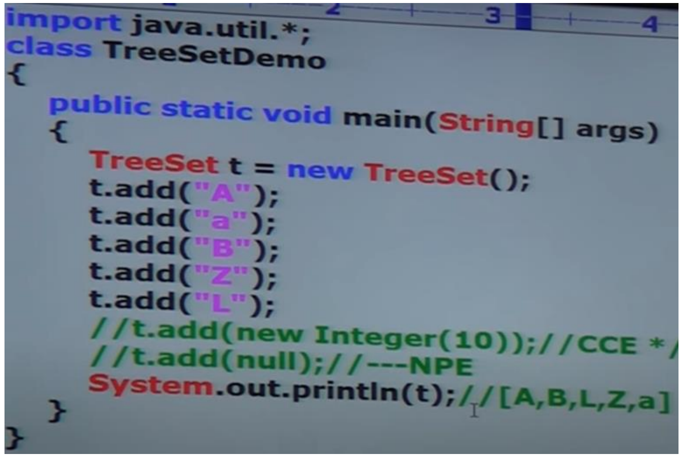
Lect part 8…

TreeSet –

- Underlying DS balanced tree
- Insertion order not preserved ,dups not allowed
- Heterogenous objects not allowed → ClassCastException
- Seriazable Cloneable
- Must implement Comparable (see StringBuffer example below) → otherwise ClassCastException

Constructors :

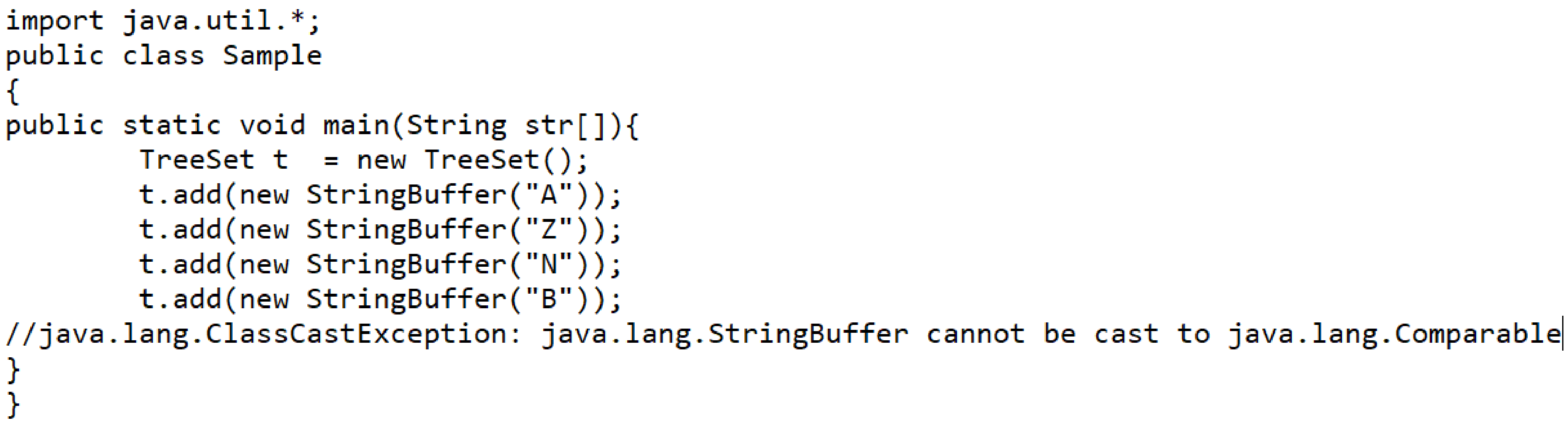
1. TreeSet t = new TreeSet() ; //default natural sorting order
2. TreeSet t = new TreeSet(Comparator c) ;//custom sorting order followed using comparator
3. TreeSet t = new TreeSet(Collection c);
4. TreeSet t = new TreeSet(SortedSet s) ;



```
import java.util.*;
class TreeSetDemo
{
   public static void main(String[] args)
   {
      TreeSet t = new TreeSet();
      t.add("A");
      t.add("a");
      t.add("B");
      t.add("Z");
      t.add("L");
      //t.add(new Integer(10));//CCE */
      //t.add(null);//---NPE
      System.out.println(t);//[A,B,L,Z,a]
   }
}
```

null acceptance :

- *for non empty TreeSet if we try to insert null then NPE*
- *Until 1.6 version → if first element in empty TreeSet then no NPE , acceptable BUT <u>if add second non null element then NPE for second element.</u>*
- *From 1.7 onwards first element null also gives NPE.*



```
import java.util.*;
public class Sample
{
public static void main(String str[]){
        TreeSet t  = new TreeSet();
        t.add(new StringBuffer("A"));
        t.add(new StringBuffer("Z"));
        t.add(new StringBuffer("N"));
        t.add(new StringBuffer("B"));
//java.lang.ClassCastException: java.lang.StringBuffer cannot be cast to java.lang.Comparable
}
}
```

String implements comparable but stringbuffer not

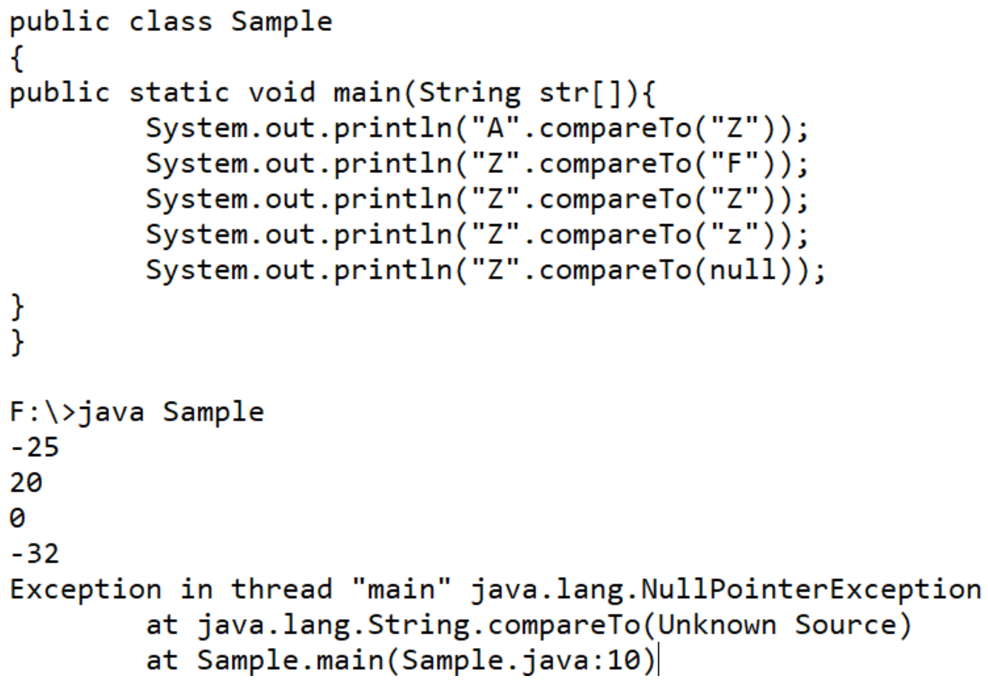
Comparable (I)

- **Present in java.lang package .**
- **It contains only one method compareTo**

public int compareTo(Object o)
obj1.compareTo(obj2)

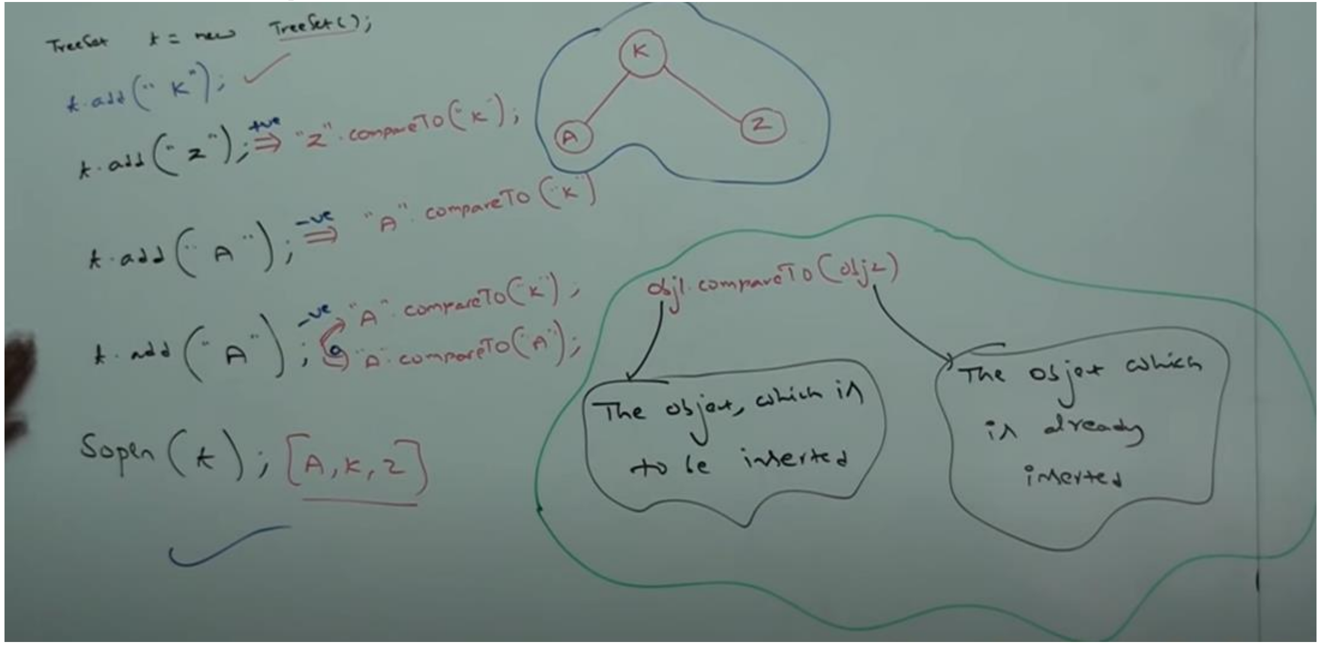
- **returns -ve iff obj1 has to come before obj2**
- **returns +ve iff obj1 has to come after obj2**
- **returns 0 iff obj1 equal to obj2**

Sop(**"A".compareTo("Z")); // -ve**
Sop(**"X".compareTo("B")); // +ve**
Sop(**"A".compareTo("A")); // 0**



```
public class Sample
{
public static void main(String str[]){
        System.out.println("A".compareTo("Z"));
        System.out.println("Z".compareTo("F"));
        System.out.println("Z".compareTo("Z"));
        System.out.println("Z".compareTo("z"));
        System.out.println("Z".compareTo(null));
}
}

F:\>java Sample
-25
20
0
-32
Exception in thread "main" java.lang.NullPointerException
        at java.lang.String.compareTo(Unknown Source)
        at Sample.main(Sample.java:10)
```

If we are depending on default natural sorting order then while adding objects into the TreeSet
JVM will call compareTo method


TreeSet t = new TreeSet();
t.add("K"); ✓
t.add("Z"); +ve ⇒ "Z".compareTo("K");
t.add("A"); -ve ⇒ "A".compareTo("K")
t.add("A"); -ve "A".compareTo("K"); 0 "A".compareTo("A");
Sopln(t); [A,K,Z]
K
A
Z
obj1.compareTo(obj2)
The object, which is to be inserted
The object which is already inserted

- If sorting which we desired is not available then we can take option of customised sorting i.e → use of Comparator Interface' compare method

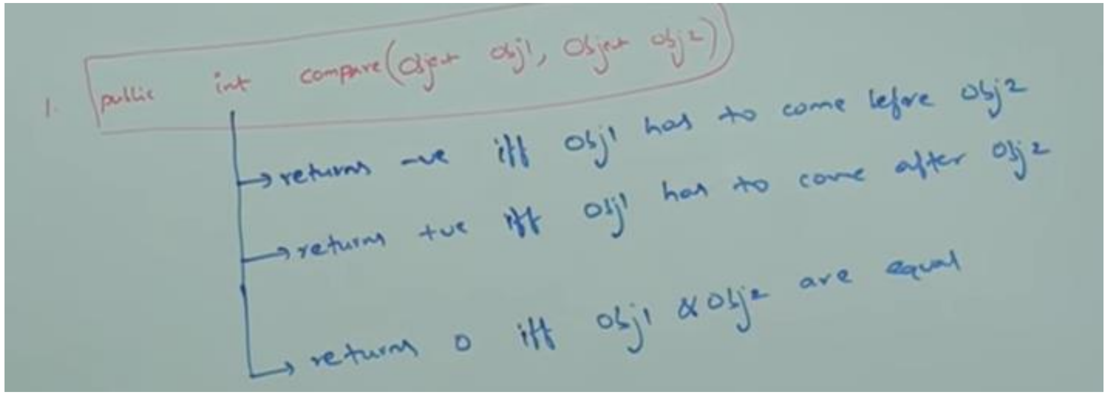
Comparable is meant for natural sorting order

Comparator is meant for customised sorting order

Comparator (I) →

- **Comparator present in java.util. package (but comparable present in java.lang ---note).**
- **It defines two methods → compare and equals;**

i. public int compare(Object o1 ,Object o2)



1. public int compare(Object obj1, Object obj2)

→ returns -ve iff obj1 has to come before obj2

→ returns +ve iff obj1 has to come after obj2

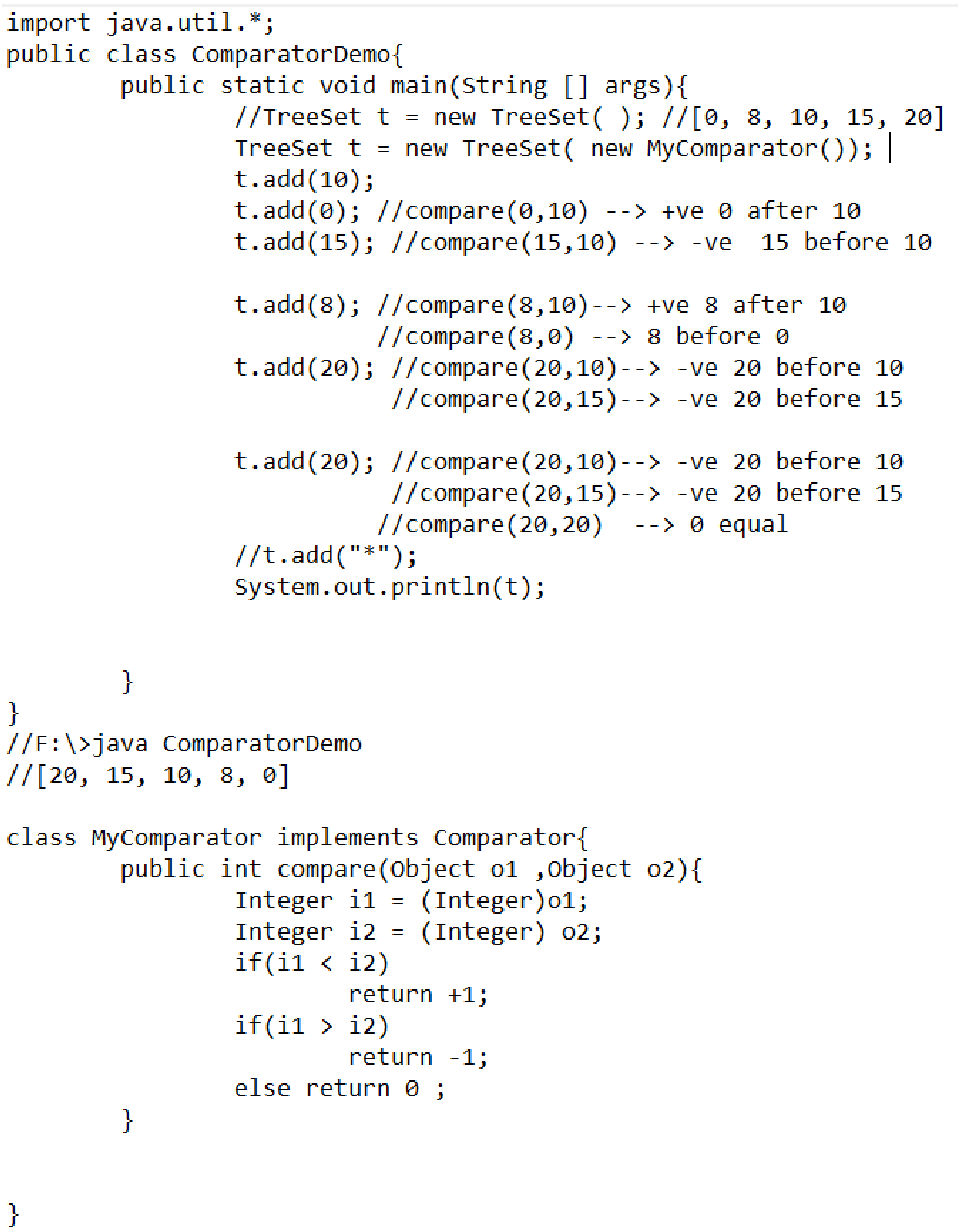
→ returns 0 iff obj1 & obj2 are equal

ii. public boolean equals(Object o);

NOTE : no need to provide implementation of equals method as equals method implementation already provided by Object class which is already superclass of calling object.

Only compare method implemented whenever we want custom sorting

Write a Program to insert integer into the TreeSet where the sorting order is desc order

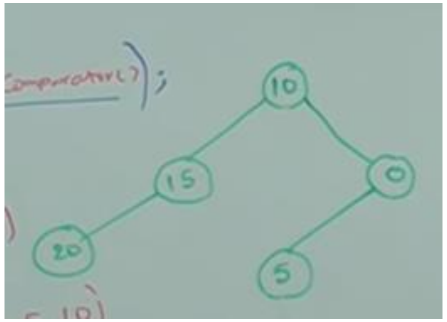


```
import java.util.*;
public class ComparatorDemo{
        public static void main(String [] args){
                //TreeSet t = new TreeSet( ); //[0, 8, 10, 15, 20]
                TreeSet t = new TreeSet( new MyComparator());
                t.add(10);
                t.add(0); //compare(0,10) --> +ve 0 after 10
                t.add(15); //compare(15,10) --> -ve  15 before 10

                t.add(8); //compare(8,10)--> +ve 8 after 10
                          //compare(8,0) --> 8 before 0
                t.add(20); //compare(20,10)--> -ve 20 before 10
                           //compare(20,15)--> -ve 20 before 15

                t.add(20); //compare(20,10)--> -ve 20 before 10
                           //compare(20,15)--> -ve 20 before 15
                          //compare(20,20)  --> 0 equal
                //t.add("*");
                System.out.println(t);


        }
}
//F:\>java ComparatorDemo
//[20, 15, 10, 8, 0]

class MyComparator implements Comparator{
        public int compare(Object o1 ,Object o2){
                Integer i1 = (Integer)o1;
                Integer i2 = (Integer) o2;
                if(i1 < i2)
                        return +1;
                if(i1 > i2)
                        return -1;
                else return 0 ;
        }


}
```

NOTES:

- **At line 1 if we don’t pass comparator obj then internally compareTo method called which is meant for default natural sorting order**
  In this case output is ascending
- **At line 1 if we pass comparator obj then internally compare method called which is meant for customised sorting order In this case output is ascending**

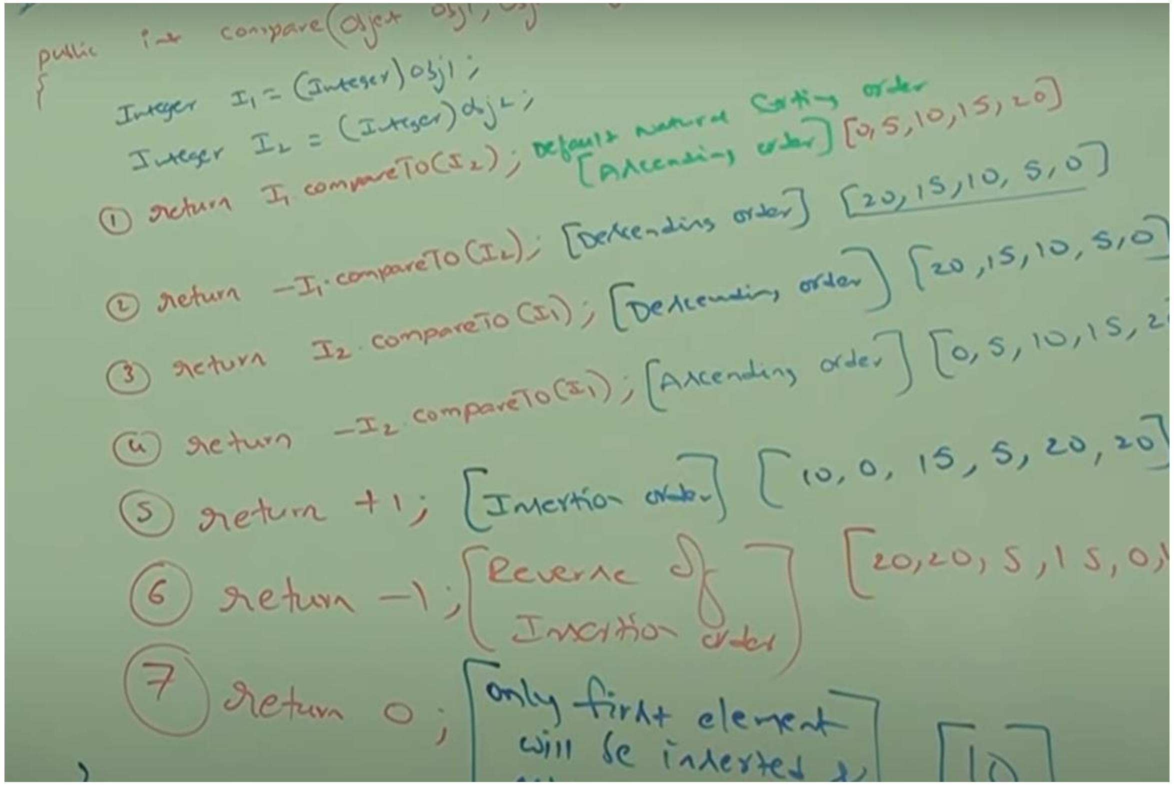
In-order traversal



Various possible implementation of compare methods **##IMPORTANT**

i. **return i1.compareTo(i2); //asc order**
ii. **return -(i1.compareTo(i2)); //desc**
iii. **return i2.compareTo(i1); //desc**
iv. **return +111 ; //follows insertion order   [10, 0, 15, 8, 20, 20]**
v. **retuen -123; //reverse of insertion order   [20, 20, 8, 15, 0, 10]**
vi. **return 0 ; // return first ele [10] as** remaining all considered as dups by jvm although not actually

```
class MyComparator implements Comparator{
        public int compare(Object o1 ,Object o2){
                Integer i1 = (Integer)o1;
                Integer i2 = (Integer) o2;
                return i2.compareTo(i1);
        }


}

//[20,15,10 ,8, 0]
```



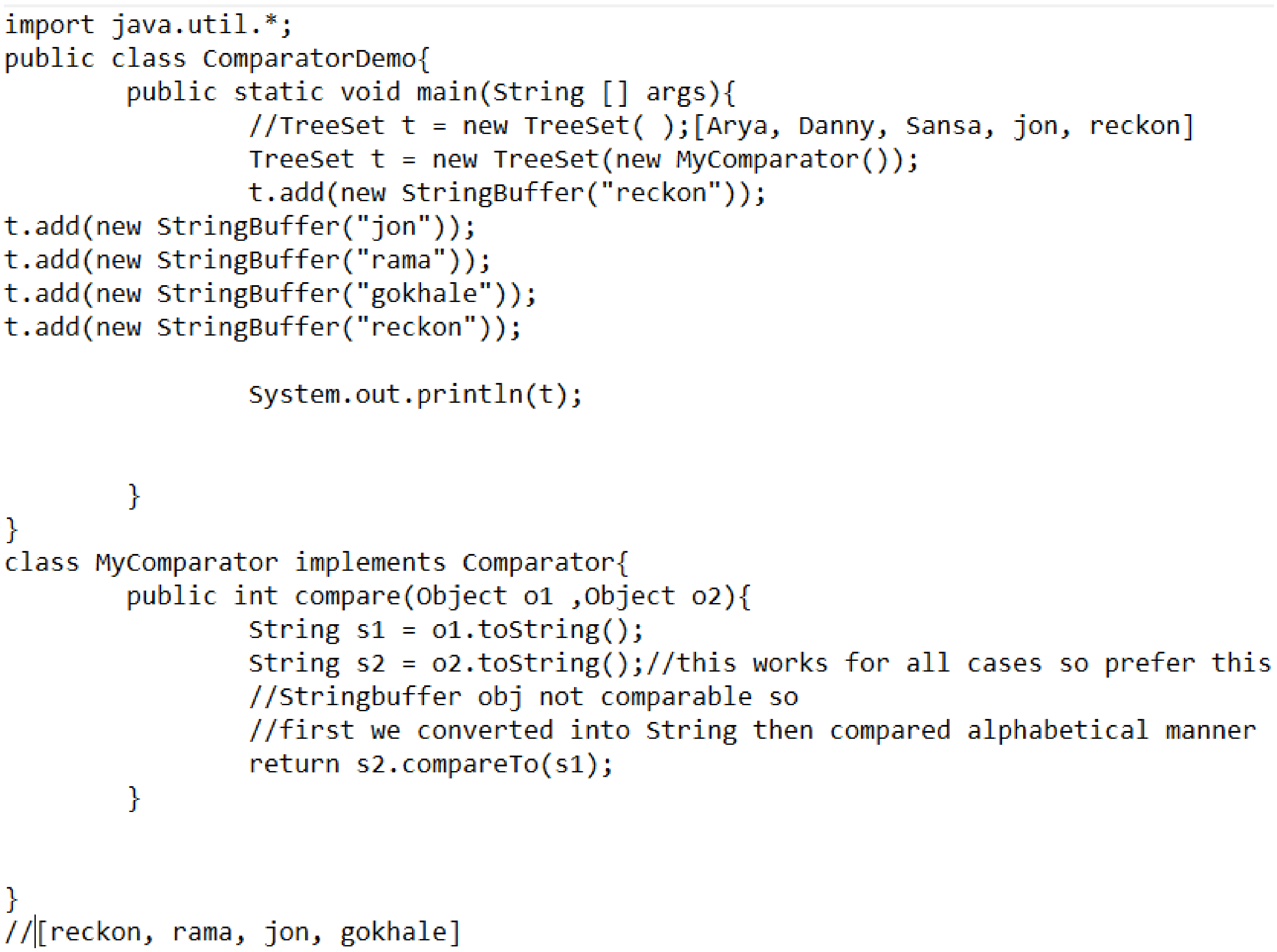
```
public int compare(Object obj1, obj
{
    Integer I1 = (Integer)obj1;
    Integer I2 = (Integer)obj2;
  ① return I1.compareTo(I2); Default natural sorting order
                             [Ascending order] [0,5,10,15,20]
  ② return -I1.compareTo(I2); [Descending order] [20,15,10, 5,0]
  ③ return I2.compareTo(I1); [Descending order] [20 ,15,10, 5,0
  ④ return -I2.compareTo(I1); [Ascending order] [0, 5, 10,15, 2
  ⑤ return +1; [Insertion order] [10, 0, 15, 5, 20, 20]
  ⑥ return -1; [Reverse of
               Insertion order] [20,20, 5 ,15, 0,
  ⑦ return 0; [only first element
              will be inserted &   [10
}
```

Write a Program to insert String into the TreeSet where the sorting order is desc order

```
import java.util.*;
public class ComparatorDemo{
        public static void main(String [] args){
                //TreeSet t = new TreeSet( );[Arya, Danny, Sansa, jon, reckon]
                TreeSet t = new TreeSet(new MyComparator());
                t.add("reckon");
                t.add("jon");
                t.add("Arya");
                t.add("Sansa");
                t.add("Danny");
                System.out.println(t);


        }
}
class MyComparator implements Comparator{
        public int compare(Object o1 ,Object o2){
                String s1 = (String) o1 ; // if args is string typecasting works
                String s2 = o2.toString();//this works for all cases so prefer this
                return s2.compareTo(s1);
        }


}
```

```
Output :
F:\>java ComparatorDemo
[reckon, jon, Sansa, Danny, Arya]
```

Write program from StringBuffer object where sorting in alphabetically ordered.



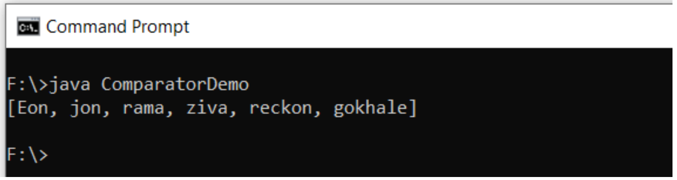
```
import java.util.*;
public class ComparatorDemo{
        public static void main(String [] args){
                //TreeSet t = new TreeSet( );[Arya, Danny, Sansa, jon, reckon]
                TreeSet t = new TreeSet(new MyComparator());
                t.add(new StringBuffer("reckon"));
t.add(new StringBuffer("jon"));
t.add(new StringBuffer("rama"));
t.add(new StringBuffer("gokhale"));
t.add(new StringBuffer("reckon"));

                System.out.println(t);


        }
}
class MyComparator implements Comparator{
        public int compare(Object o1 ,Object o2){
                String s1 = o1.toString();
                String s2 = o2.toString();//this works for all cases so prefer this
                //Stringbuffer obj not comparable so
                //first we converted into String then compared alphabetical manner
                return s2.compareTo(s1);
        }


}
//[reckon, rama, jon, gokhale]
```

- **If we are depend on default natural sorting order object should be homogenous n comparable otherwise CCE runtime exception**
- **If we are defining our own sorting by comparator then objects need not be comparable or homogenous ..that is we can add heterogenous non comparable object also….**

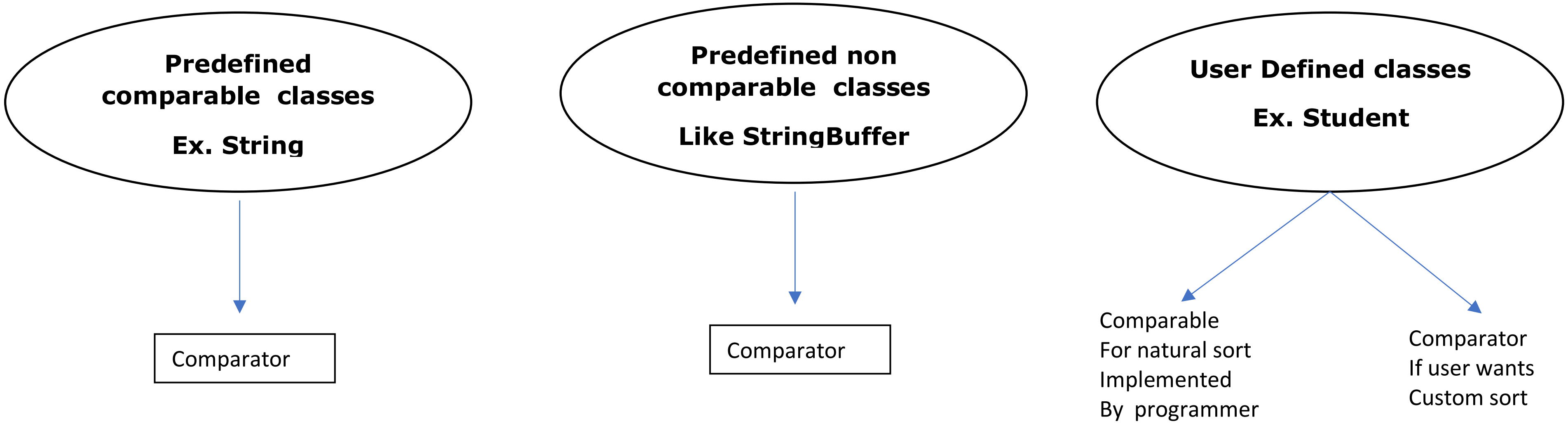
Write program where sorting order is increasing length order.

If two object have same length then consider alphabetical order

```
//sorting based on length of the string n if same len then alphabetically....
import java.util.*;
public class ComparatorDemo{
        public static void main(String [] args){
                //TreeSet t = new TreeSet( );[Arya, Danny, Sansa, jon, reckon]
                TreeSet t = new TreeSet(new MyComparator());
                t.add(new StringBuffer("reckon"));
                t.add("jon");
                t.add(new StringBuffer("Eon"));
                t.add(new StringBuffer("rama"));
                t.add("gokhale");
                t.add("ziva");
                System.out.println(t);

        }
}
class MyComparator implements Comparator{
        public int compare(Object o1 ,Object o2){
                String s1 = o1.toString();
                String s2 = o2.toString();//this works for all cases so prefer this
                int l1 =s1.length();
                int l2 =s2.length();
                return l1<l2?-1:(l1>l2)?9:s1.compareTo(s2);
        }
}
F:\>java ComparatorDemo
[Eon, jon, rama, ziva, reckon, gokhale]
```



```
F:\>java ComparatorDemo
[Eon, jon, rama, ziva, reckon, gokhale]

F:\>
```



Predefined comparable classes

Ex. String

Comparator

Predefined non comparable classes

Like StringBuffer

Comparator

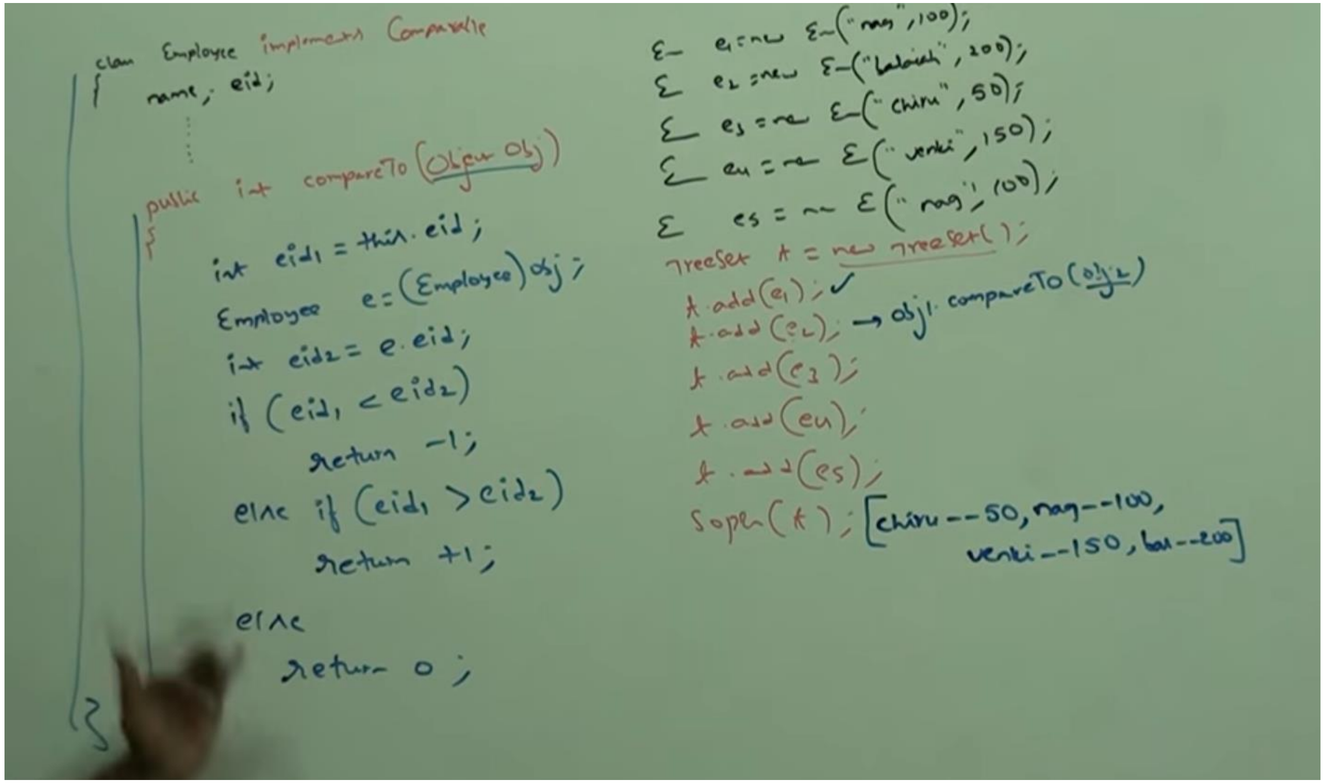
User Defined classes

Ex. Student

Comparable
For natural sort
Implemented
By programmer

Comparator
If user wants
Custom sort

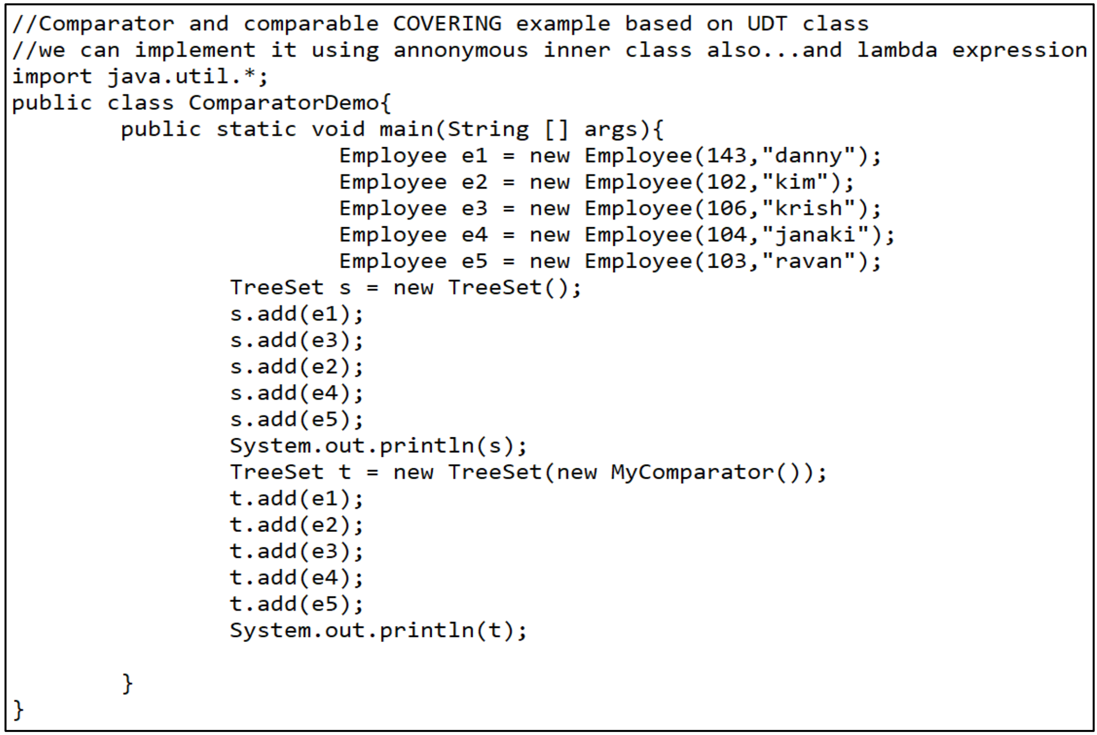
- The person who is writing the class is responsible to define def natural sorting by impl comparable interface
- If person who is using our class not satisfied with our natural sorting order then go with comparator for custom sort.



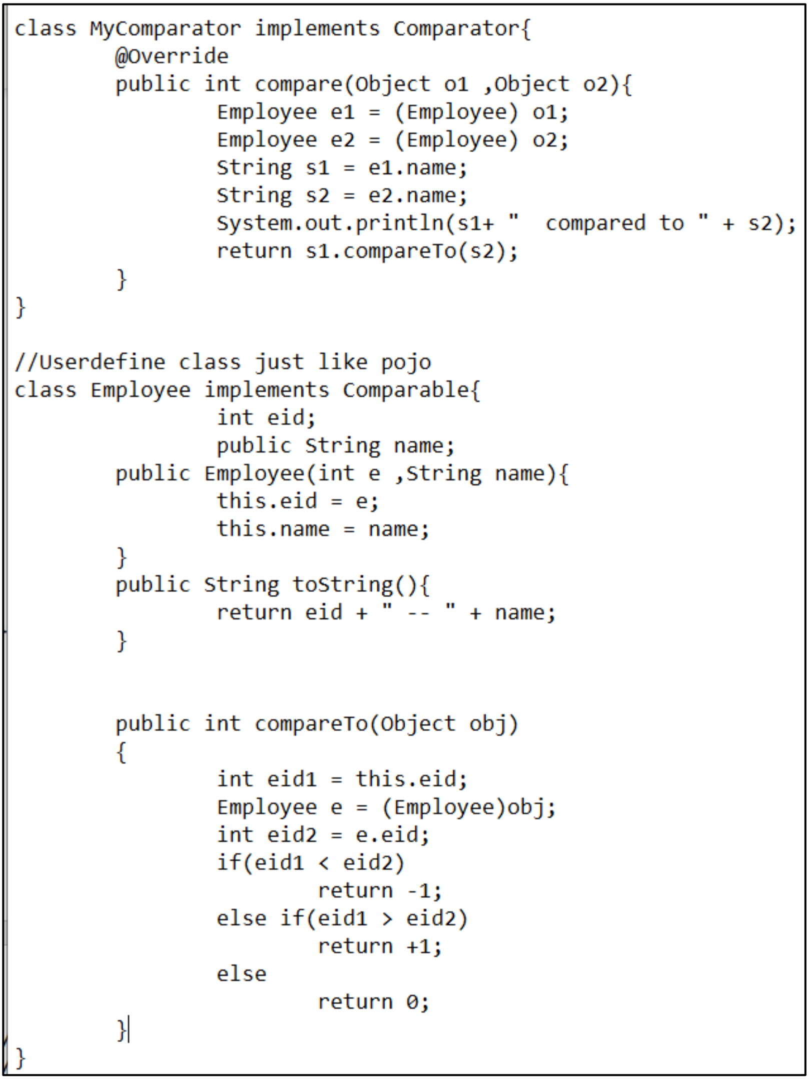
```
class Employee implements Comparable
{
    name; eid;
       :
       :
    public int compareTo(Object obj)
    {
        int eid1 = this.eid;
        Employee e=(Employee)obj;
        int eid2= e.eid;
        if (eid1 < eid2)
             return -1;
        else if (eid1 >eid2)
             return +1;

        else
           return 0;
}
```

```
E-  e1=new E-("nag",100);
E   e2=new E-("balaiah",200);
E   e3=new E-("chiru",50);
E   e4=new E("venki",150);
E   e5=new E("nag",100);
TreeSet t = new TreeSet();
  t.add(e1); ✓
  t.add(e2); → obj1.compareTo(obj2)
  t.add(e3);
  t.add(e4);
  t.add(e5);
  Sopln(t); [chiru--50,nag--100,
                venki--150,bal--200]
```

Core Java With OCJP/SCJP: Collections Part-10 || string object in Treeset Example



```
//Comparator and comparable COVERING example based on UDT class
//we can implement it using annonymous inner class also...and lambda expression
import java.util.*;
public class ComparatorDemo{
        public static void main(String [] args){
                        Employee e1 = new Employee(143,"danny");
                        Employee e2 = new Employee(102,"kim");
                        Employee e3 = new Employee(106,"krish");
                        Employee e4 = new Employee(104,"janaki");
                        Employee e5 = new Employee(103,"ravan");
                TreeSet s = new TreeSet();
                s.add(e1);
                s.add(e3);
                s.add(e2);
                s.add(e4);
                s.add(e5);
                System.out.println(s);
                TreeSet t = new TreeSet(new MyComparator());
                t.add(e1);
                t.add(e2);
                t.add(e3);
                t.add(e4);
                t.add(e5);
                System.out.println(t);

        }
}
```



```
class MyComparator implements Comparator{
        @Override
        public int compare(Object o1 ,Object o2){
                Employee e1 = (Employee) o1;
                Employee e2 = (Employee) o2;
                String s1 = e1.name;
                String s2 = e2.name;
                System.out.println(s1+ "  compared to " + s2);
                return s1.compareTo(s2);
        }
}

//Userdefine class just like pojo
class Employee implements Comparable{
                int eid;
                public String name;
        public Employee(int e ,String name){
                this.eid = e;
                this.name = name;
        }
        public String toString(){
                return eid + " -- " + name;
        }


        public int compareTo(Object obj)
        {
                int eid1 = this.eid;
                Employee e = (Employee)obj;
                int eid2 = e.eid;
                if(eid1 < eid2)
                        return -1;
                else if(eid1 > eid2)
                        return +1;
                else
                        return 0;
        }
}
```

[102 -- kim, 103 -- ravan, 104 -- janaki, 106 -- krish, 143 -- danny] → natural sort with eid

[143 -- danny, 104 -- janaki, 102 -- kim, 106 -- krish, 103 -- ravan] → custom sort based on emp name

Interface	Comparable	Comparator
Present in	Java.lang package	Java.util package
Meant for	Default natural sorting order	Customised Sorting order
Defines methods	Only one method → compareTo	Two methods → compare , equals
Implementation	Implemented by all wrapper classes and String	Only implemented classes of comparator are Collator and RuleBasedCollater →GUI based app class

Concluding SET part in collection tree:

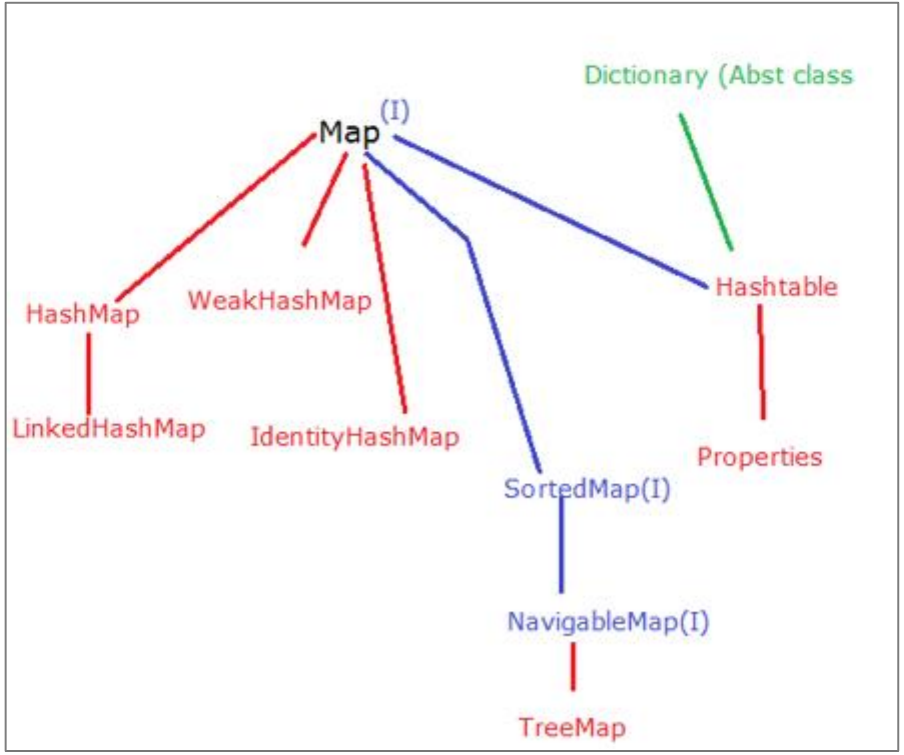
Property	HashSet	LinkedHashSet	TreeSet
Underlying DS	Hashtable	Linkedlist + Hashtable	Balanced tree
Duplicate objects	Not allowed	Not allowed	Not allowed
Insertion order	Not preserved	Preserved	Not preserved
Sorting order	NA	NA	Applicable
Heterogenous order	Allowed	Allowed	Not allowed
Null acceptance	Allowed	Allowed	For empty treeset as first n last element

--00000000000000000000000000---

MAP :

- **not** child interface of collection
- group of objects as "Key-value" pairs then go for map
- dups value can be entertained but If the key is already present then old value wil be replaced with new value

Key	value
101	"ram"
102	"tom"
103	"ram"

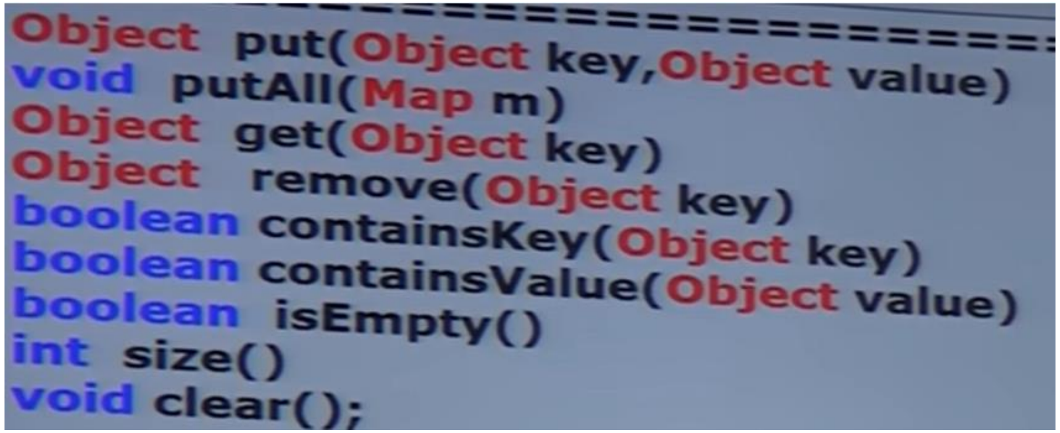


Methods ::

1. Object put(Object key, Object "value");
   To add new k,v pair in the map
   If the key is already present then old value wil be replaced with new value

Eg. m.put(101,”teja”);
    m.put(102,”jon”);
    m.put(102,”ketan”);//jon replaced by ketan.

2. m.putAll(Map m);
3. m.get(key); -- associated value returned
4. m.remove(key); -- removes entry associated with specified key.
5. m.containsKey(key); -- ret boolean
6. m.containsValue(value); -- ret boolean
7. isEmpty(); -- ret boolean
8. m.size();
9. m.clear();



```
Object  put(Object key,Object value)
void  putAll(Map m)
Object  get(Object key)
Object   remove(Object key)
boolean containsKey(Object key)
boolean containsValue(Object value)
boolean  isEmpty()
int  size()
void clear();
```

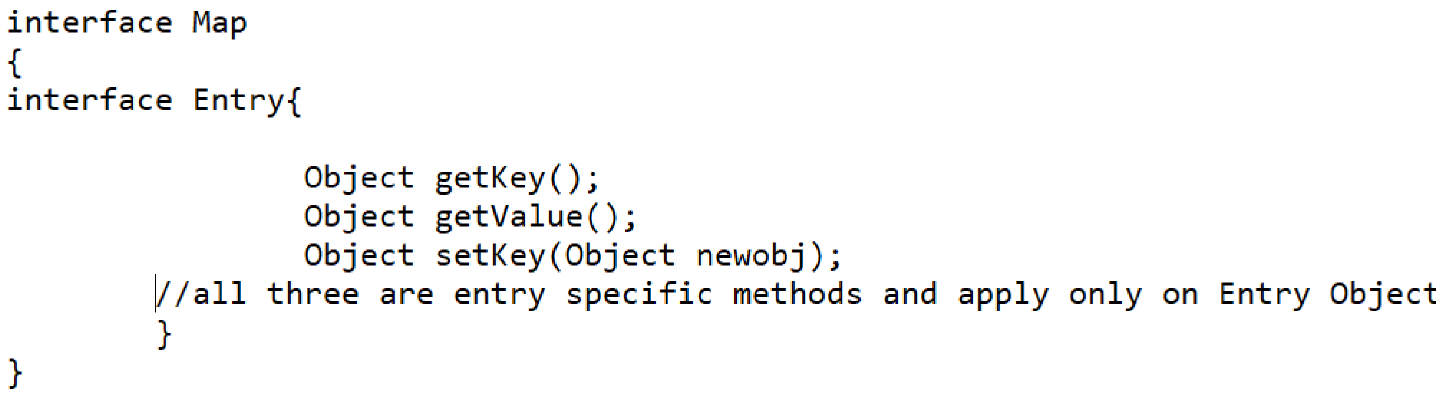
Some more important methods ::

10.Set   keySet () → dups not allowed hence set ret type
11.Collection values() → dups possible n order is not important
12.Set   entrySet() → set of Entry ret.(k,v)

Collection views of maps - as ret type is collection related

Entry(I) –

- each key-value pair is called entry hence map is collection of entry objects.
- Without map obj existence no chance of Entry obj
- Hence Entry i/f defined inside map interface



```
interface Map
{
interface Entry{

                Object getKey();
                Object getValue();
                Object setKey(Object newobj);
        //all three are entry specific methods and apply only on Entry Object
        }
}
```

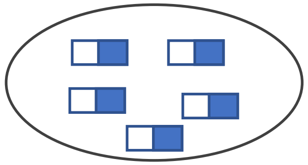
HashMap:

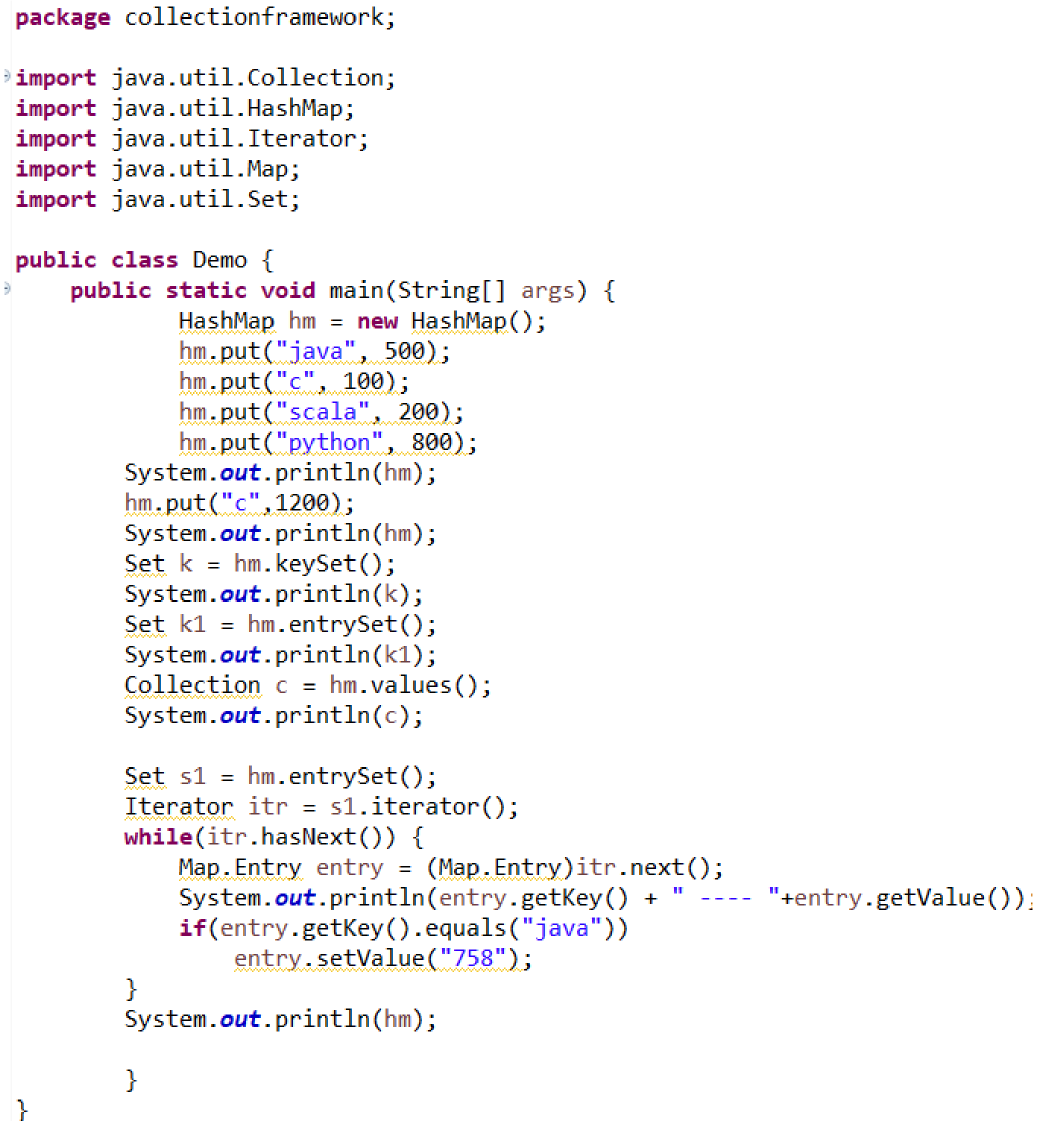
- Underlying DS Hashtable
- Hashcode of keys not values
- Dups keys not allowed ,values can be duplicated.
- Heterogeneous objects are allowed for both the key n value
- Null is allowed for key only once, null allowed for values any number of times
- Implements serializable n cloneable interface but not RandomAccess.
- Best choice if frequent operation is search operation.

Constructors:

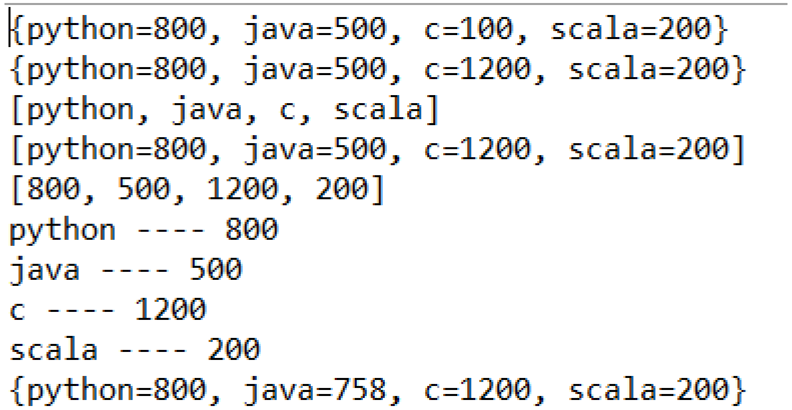
```
1. HashMap m = new HashMap() ; //init cap : 16 , fill ratio/load factor : 0.75
2. HashMap m = new HashMap(int initialcapacity) ; //fill ratio/load factor : 0.75
3. HashMap m = new HashMap(int initialcapacity,float fillratio) ;
4. HashMap m = new HashMap(Map m);
```

Get these entry objects one by one….





```
package collectionframework;

import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class Demo {
    public static void main(String[] args) {
            HashMap hm = new HashMap();
            hm.put("java", 500);
            hm.put("c", 100);
            hm.put("scala", 200);
            hm.put("python", 800);
        System.out.println(hm);
        hm.put("c",1200);
        System.out.println(hm);
        Set k = hm.keySet();
        System.out.println(k);
        Set k1 = hm.entrySet();
        System.out.println(k1);
        Collection c = hm.values();
        System.out.println(c);

        Set s1 = hm.entrySet();
        Iterator itr = s1.iterator();
        while(itr.hasNext()) {
            Map.Entry entry = (Map.Entry)itr.next();
            System.out.println(entry.getKey() + " ---- "+entry.getValue());
            if(entry.getKey().equals("java"))
                entry.setValue("758");
        }
        System.out.println(hm);

        }
}
```



```
{python=800, java=500, c=100, scala=200}
{python=800, java=500, c=1200, scala=200}
[python, java, c, scala]
[python=800, java=500, c=1200, scala=200]
[800, 500, 1200, 200]
python ---- 800
java ---- 500
c ---- 1200
scala ---- 200
{python=800, java=758, c=1200, scala=200}
```

Important for Collection interview

Hashmap	Hashtable
Every method Not synchonized	Every method is Synchronised
Many threads allowed to operate on hashmap hence not thread safe	**Only one thread operate at a time hence threadsafe**
Performance fast	**Performance slow as other threads have to wait**
Null key n value applicable to insert	**Null key or null value not applicable else NullPointerException**
Not LEGACY(1.2v)	**LEGACY 1.0 version**

How to get synchronised version of hashmap object?

```
HashMap m = new HashMap()

Map m1 =          Collections.synchronizedMap(m)
```

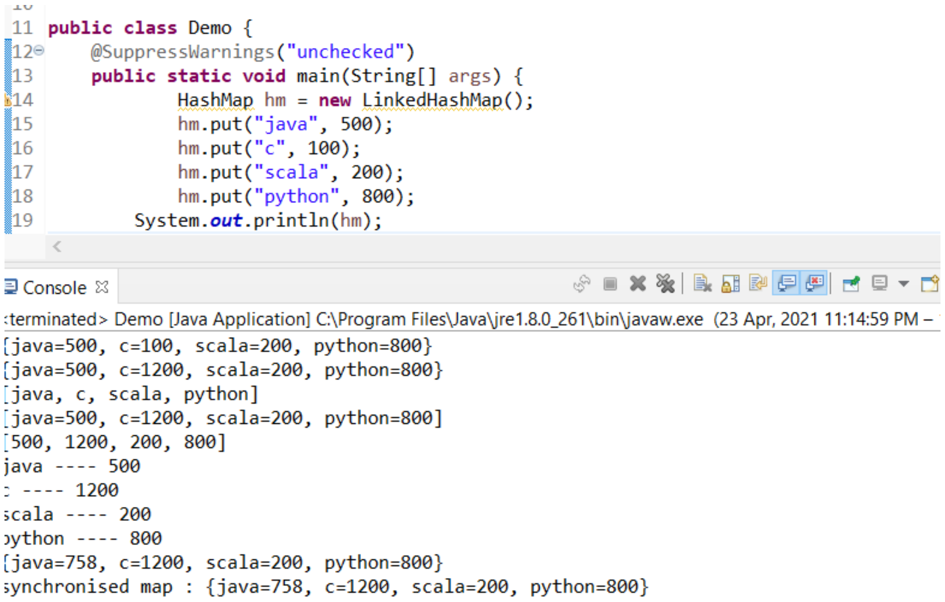
LinkedHashMap : //similar to LinkedHashSet **used for cache-based application**

Child class of hashMap

Constructors also same and most methods

HashMap	LHS
Hashtable is underlying ds	LL + Hashtable
Insertion order Is not preservered and based on hashcode of keys	Insertion order is preserved
Introduced in 1.2 v	1.4v

If above program is replaced with linked hashmap then insertion order preserved….



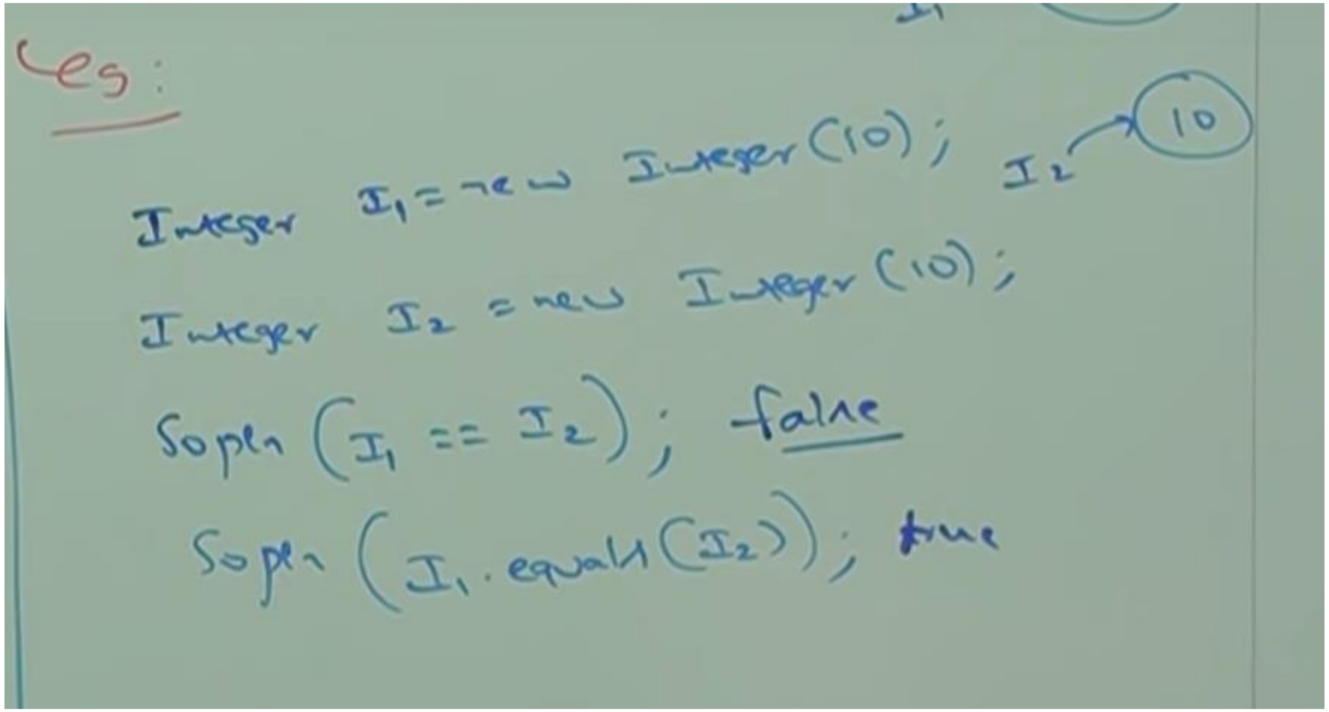
```
public class Demo {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
            HashMap hm = new LinkedHashMap();
            hm.put("java", 500);
            hm.put("c", 100);
            hm.put("scala", 200);
            hm.put("python", 800);
        System.out.println(hm);
```

```
<terminated> Demo [Java Application] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe  (23 Apr, 2021 11:14:59 PM –
{java=500, c=100, scala=200, python=800}
{java=500, c=1200, scala=200, python=800}
[java, c, scala, python]
[java=500, c=1200, scala=200, python=800]
[500, 1200, 200, 800]
java ---- 500
c ---- 1200
scala ---- 200
python ---- 800
{java=758, c=1200, scala=200, python=800}
synchronised map : {java=758, c=1200, scala=200, python=800}
```

IdentityHashMap

== vs .equal difference

== is for reference or address comparison whereras .equals method meant for content comparison.



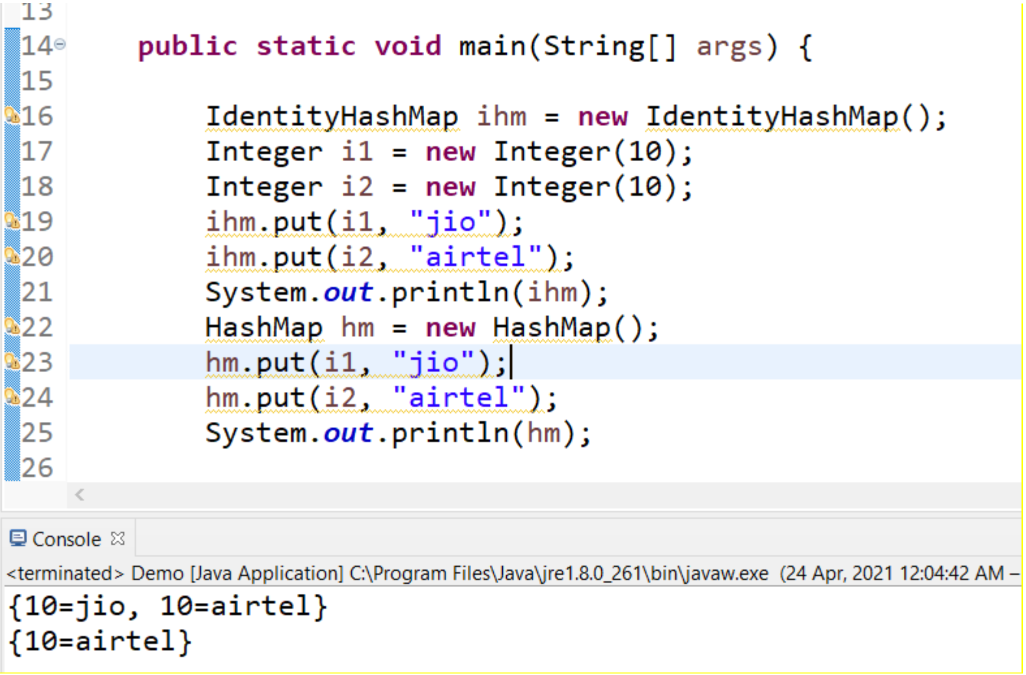
eg:

```
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);
Sopln (I1 == I2); false
Sopln (I1.equals(I2)); true
```

Exactly similar to HM except following difference

In hashmap jvm uses .equals method to find duplicate keys …which is meant for content comparison

But in case of identity hm jvm will use == operator to identify dup keys…which is meant for reference comparison or address comparison



```java
    public static void main(String[] args) {

        IdentityHashMap ihm = new IdentityHashMap();
        Integer i1 = new Integer(10);
        Integer i2 = new Integer(10);
        ihm.put(i1, "jio");
        ihm.put(i2, "airtel");
        System.out.println(ihm);
        HashMap hm = new HashMap();
        hm.put(i1, "jio");
        hm.put(i2, "airtel");
        System.out.println(hm);

```

```
<terminated> Demo [Java Application] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe  (24 Apr, 2021 12:04:42 AM –
{10=jio, 10=airtel}
{10=airtel}
```

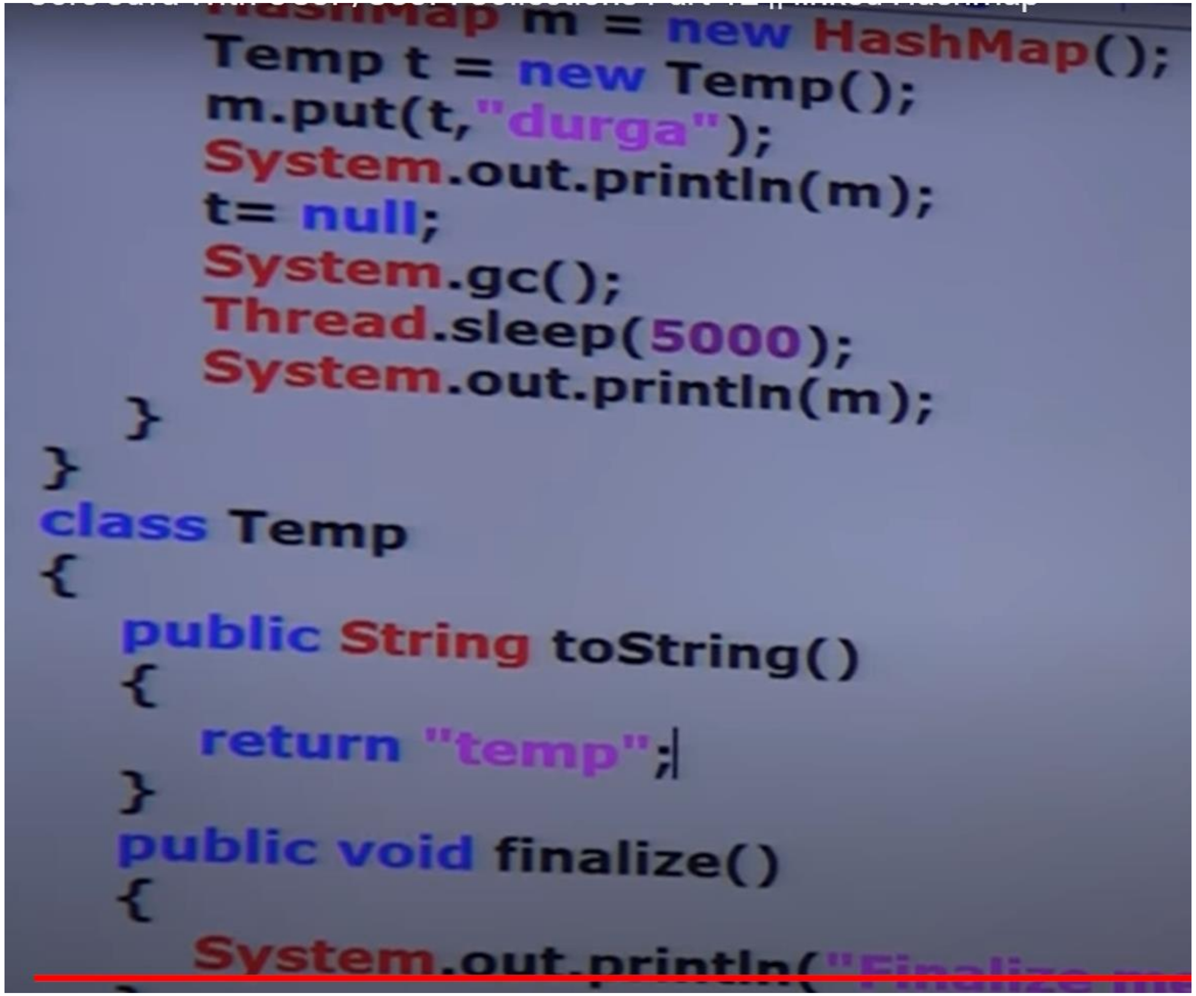
WeakHashMap → same as HS except below diff

In case of hashmap even though object doesn’t have any ref it is not eligible for gc if it is associated with hashmap

That is hm dominates gc.

But WeakHashMap is weak by name here gc dominates whm → any memory used by dereferenced object get freed by gc even if

Associated with WeakHashMap…



```
        HashMap m = new HashMap();
        Temp t = new Temp();
        m.put(t,"durga");
        System.out.println(m);
        t= null;
        System.gc();
        Thread.sleep(5000);
        System.out.println(m);
    }
}
class Temp
{
    public String toString()
    {
        return "temp";
    }
    public void finalize()
    {
        System.out.println("Finalize me
```

HASHTABLE ::

- Underlying ds for java hashtable → hashtable
- Insertion order not preserved n it is based on hashcode of keys
- Dups key not allowed ,values allowed
- Heterogenous obj allowed for both k v
- Null not allowed for both k v otherwise NPException
- Serializable and cloneable
- Every method Synchornized hence thread safe
- Choice if freq searching operation.

Hashtable init cap = 11 not 16 as Hm and hs

Constructors same as hm only name changes