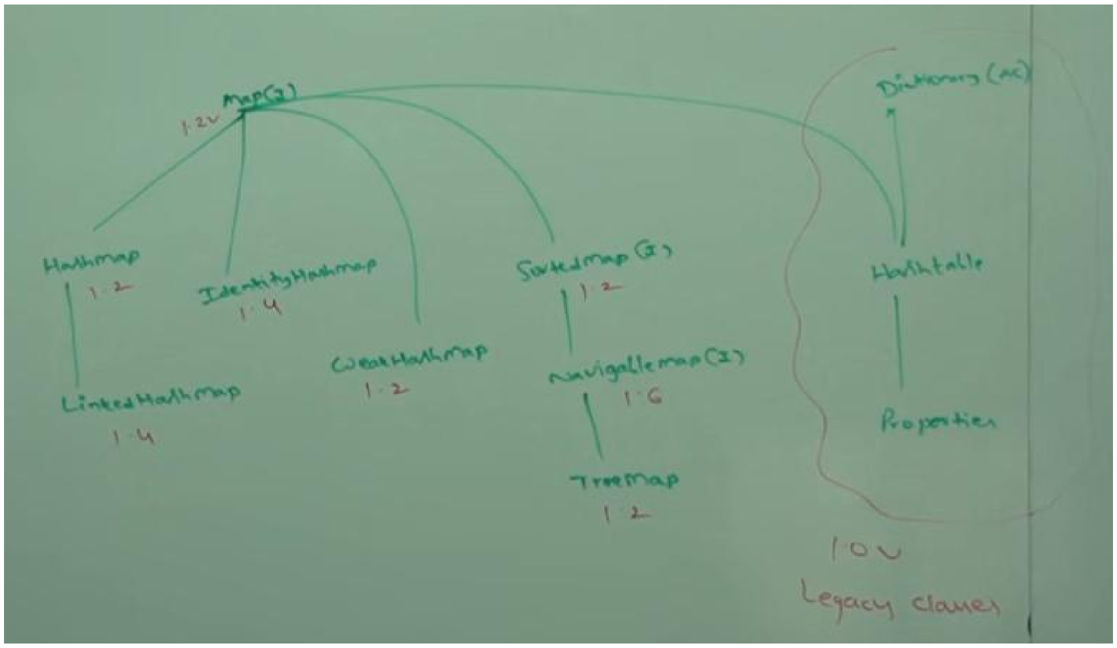
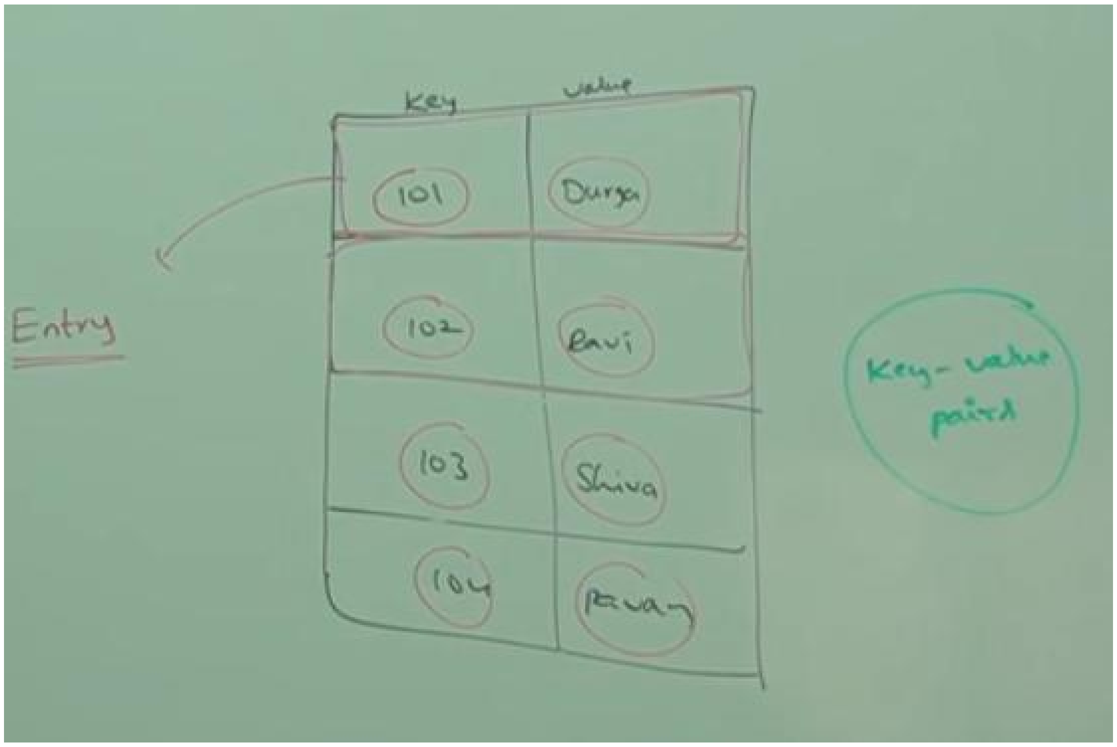


Map is not child interface of collection.

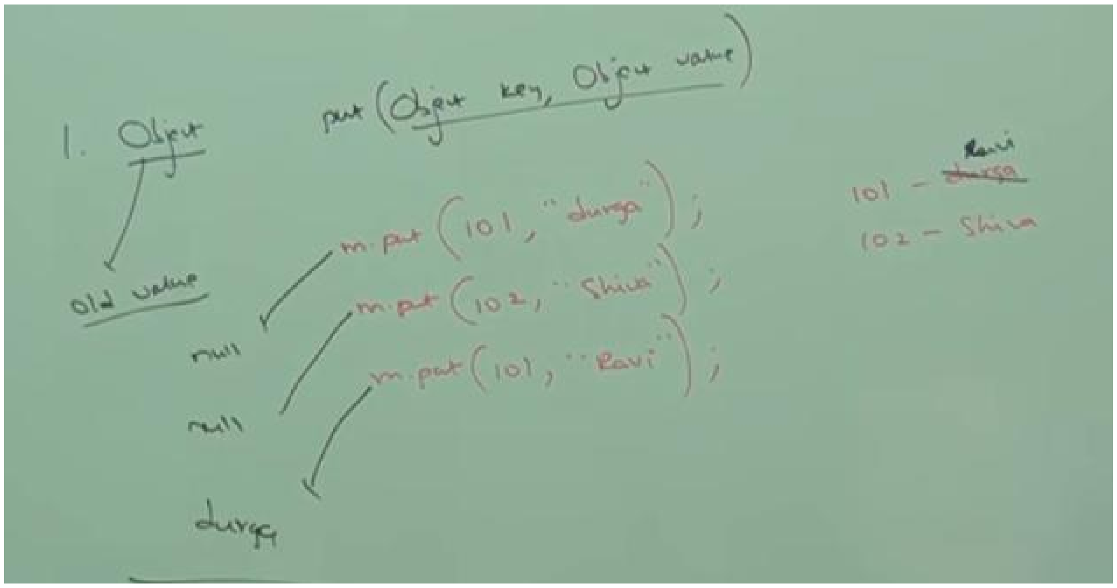
If we want to represent a group of objects as key value pairs then we should go for Map.



Both keys and values are objects only. Duplicates keys are not allowed but values can be dupilcated.

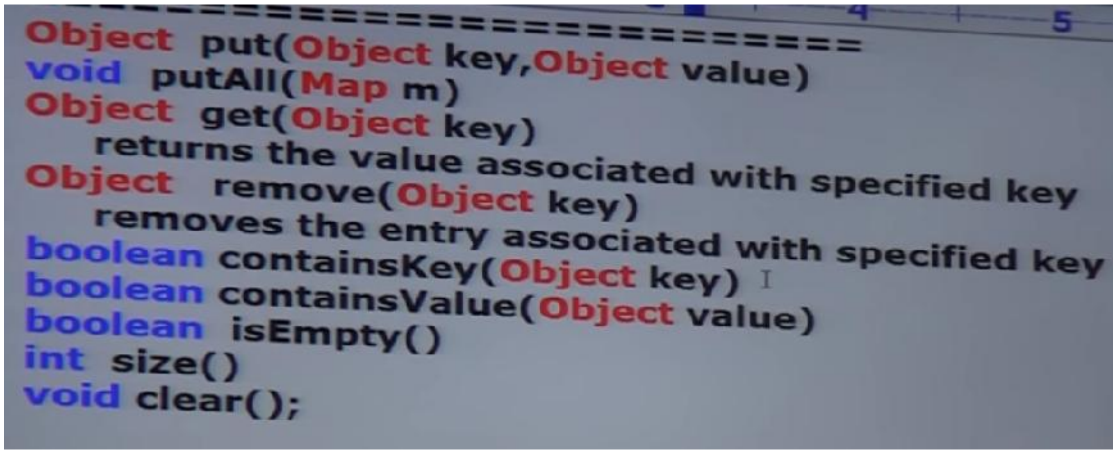
Each key value pair is called Entry. Hence Map is considered as a collection of Entry Objects.

Map Interface methods:

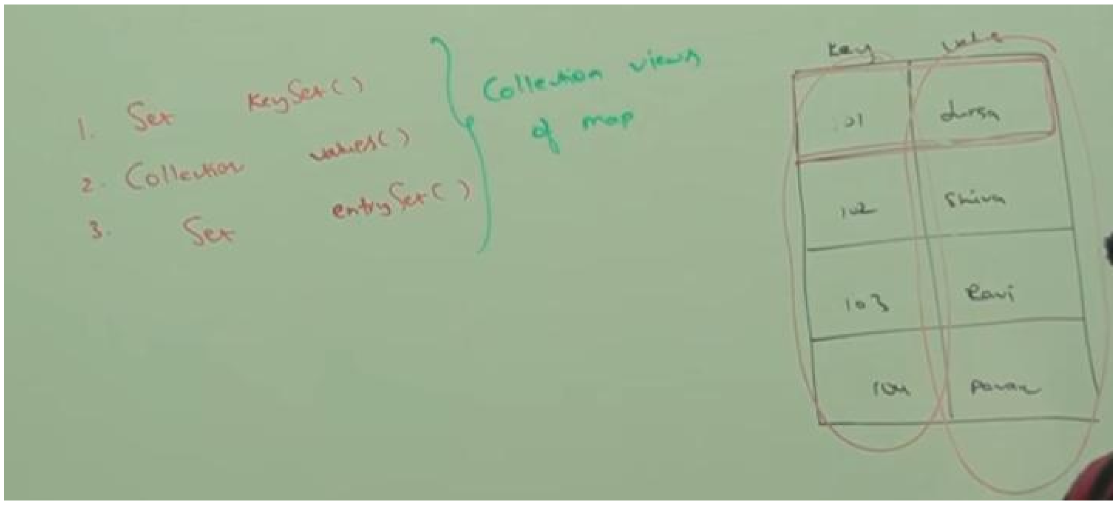


To add one key value to the map.

If the key is already present then old value will be replaced with new value and returns old value. see the example above.



```
===========================
Object  put(Object key,Object value)
void  putAll(Map m)
Object  get(Object key)
    returns the value associated with specified key
Object   remove(Object key)
    removes the entry associated with specified key
boolean containsKey(Object key)
boolean containsValue(Object value)
boolean  isEmpty()
int  size()
void clear();
```



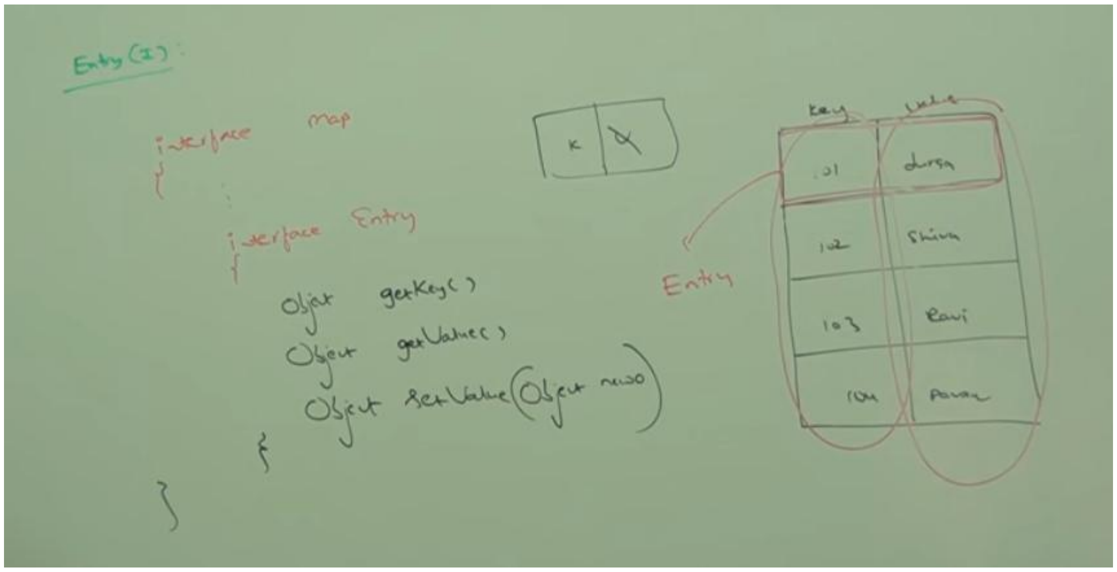
Entry(I):

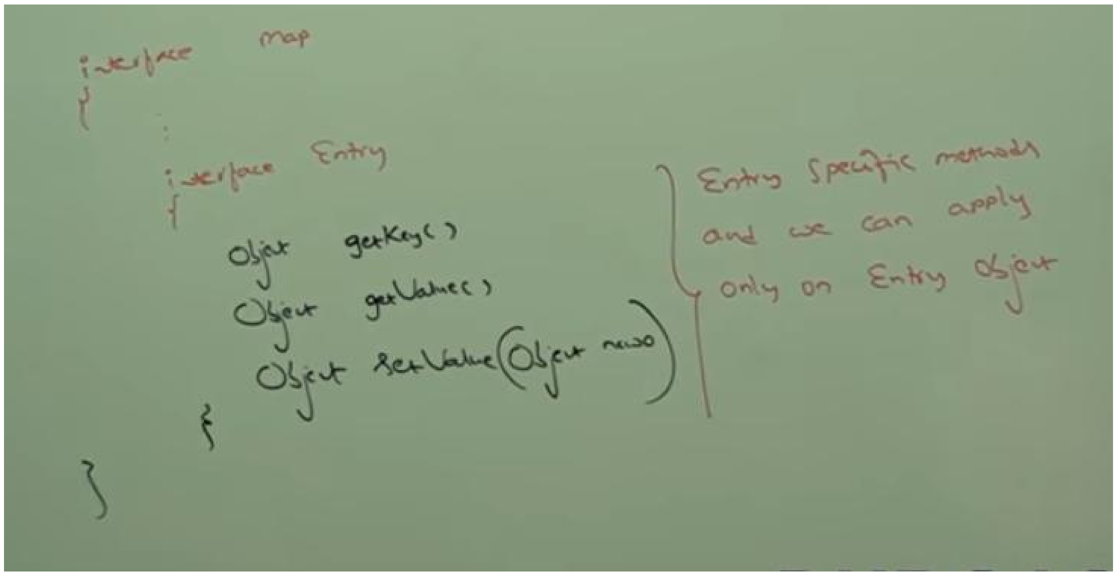
A Map is a group of key value pairs and each key value pair is called an Entry.

Hence Map is considered as a collection of Entry objects.

Without existing Map object there is no chance of existing Entry Object.

Hence Entry interface is defined inside Map interface.





```
interface Map
{
    :
    interface Entry
    {
        Object getKey()
        Object getValue()
        Object setValue(Object new)
    }
}
```

Entry Specific methods and we can apply only on Entry Object

HashMap(C)

1. The underlined data structure is Has Table.

2. Insertion order is not preserved and it is based on hashcode of keys.

3. Duplicates keys are not allowed but values can be duplicated.

4. Heterogeous objects are allowed for both key and value.

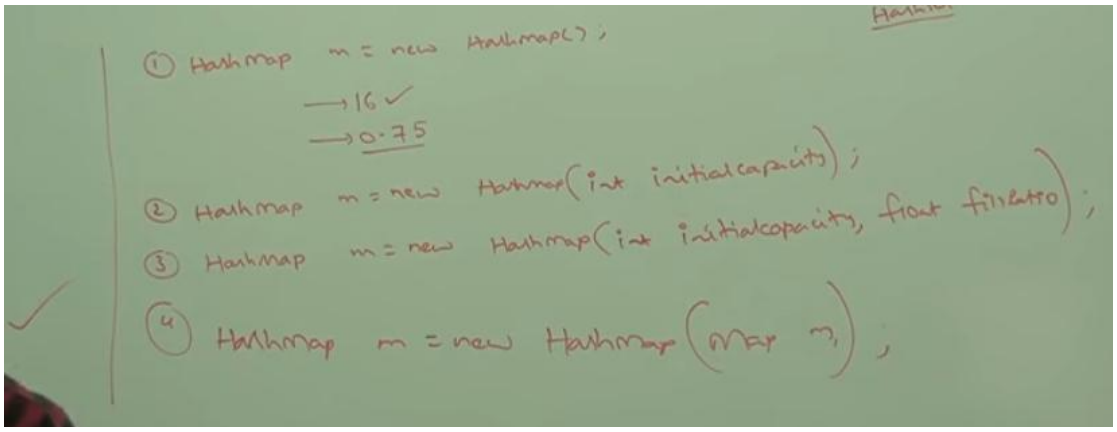
5. Null is allowed for key(only once),

6. Null is allowed for values(any no of times).

7. Hashmap implemets serializable and clonebale inetfaces but not random access.

8. Hashmap is the best choice if our frequent operation is search operation.

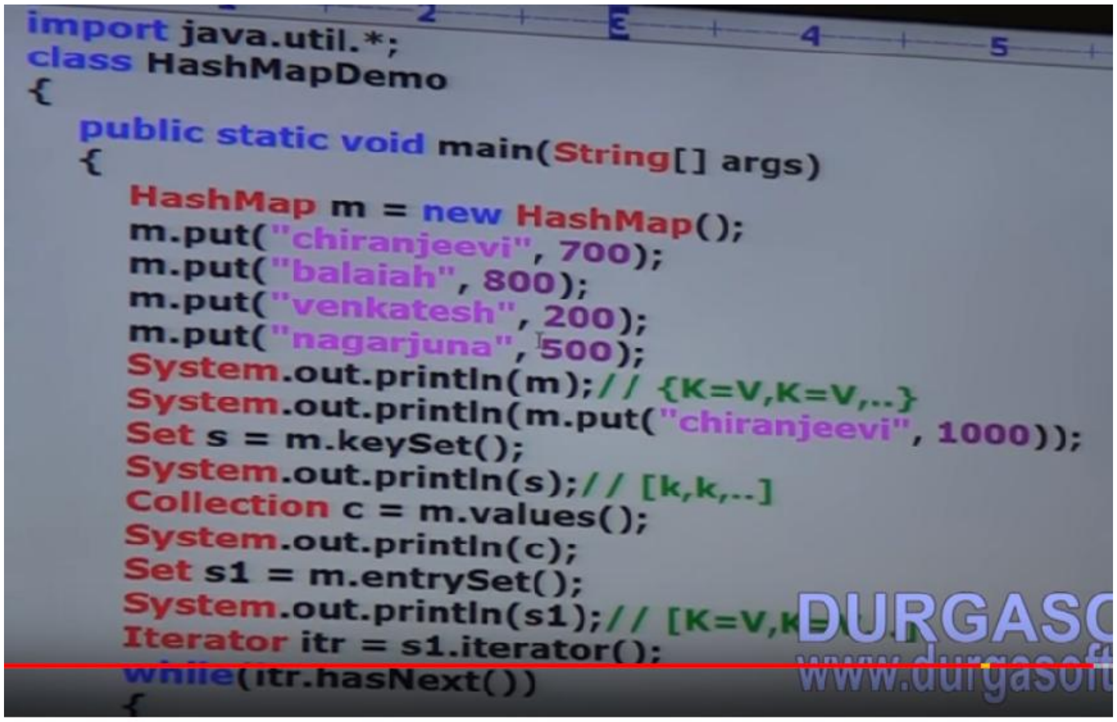
Constructors of HashMap:



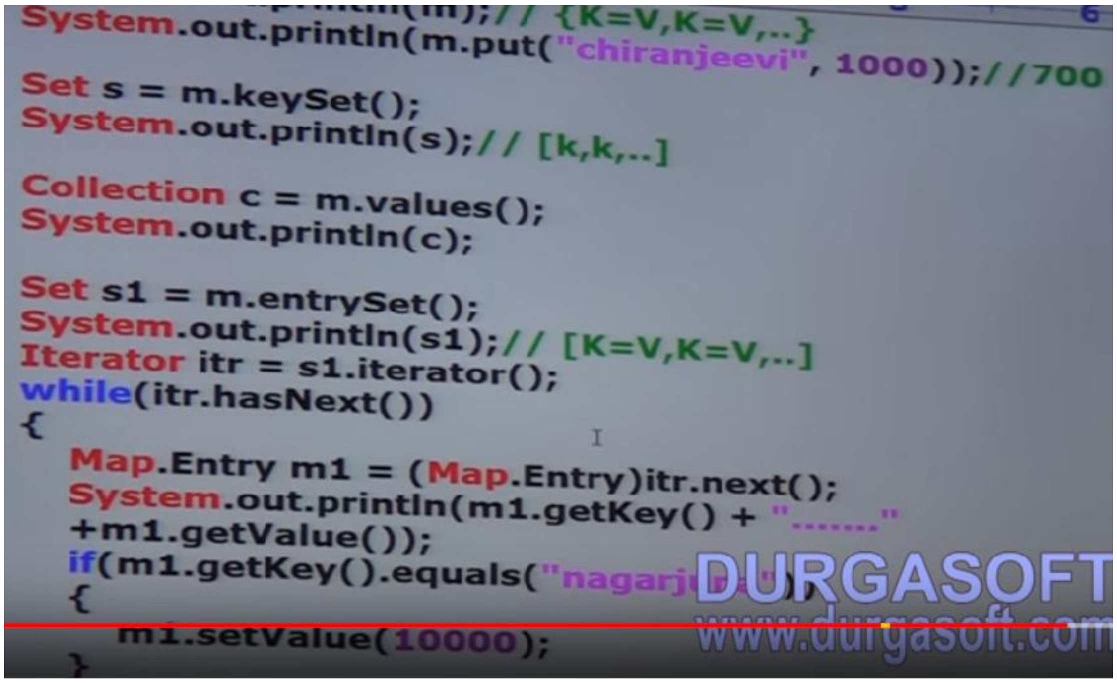
```
① HashMap m = new HashMap();
        → 16 ✓
        → 0.75
② HashMap m = new HashMap(int initialcapacity);
③ HashMap m = new HashMap(int initialcapacity, float fillRatio);
④ HashMap m = new HashMap(Map m);
```

1. creates an empty hashMap object with default initial capacity 16 and default fill ration 0.75.

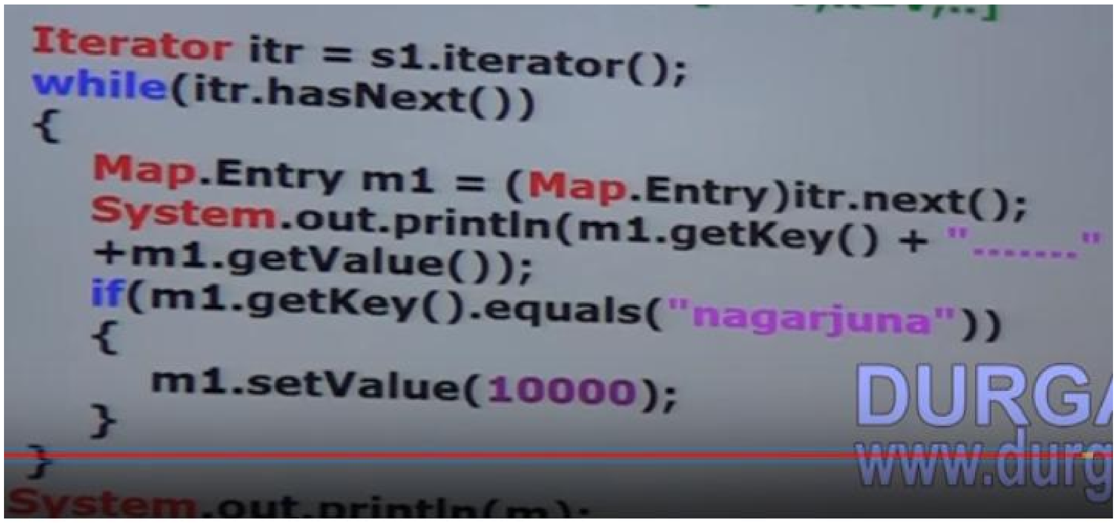
2. Creates an empty HashMap object with specified initial capacity and default fill ration 0.75.



```
import java.util.*;
class HashMapDemo
{
    public static void main(String[] args)
    {
        HashMap m = new HashMap();
        m.put("chiranjeevi", 700);
        m.put("balaiah", 800);
        m.put("venkatesh", 200);
        m.put("nagarjuna", 500);
        System.out.println(m);// {K=V,K=V,..}
        System.out.println(m.put("chiranjeevi", 1000));
        Set s = m.keySet();
        System.out.println(s);// [k,k,..]
        Collection c = m.values();
        System.out.println(c);
        Set s1 = m.entrySet();
        System.out.println(s1);// [K=V,K
        Iterator itr = s1.iterator();
        while(itr.hasNext())
        {
```

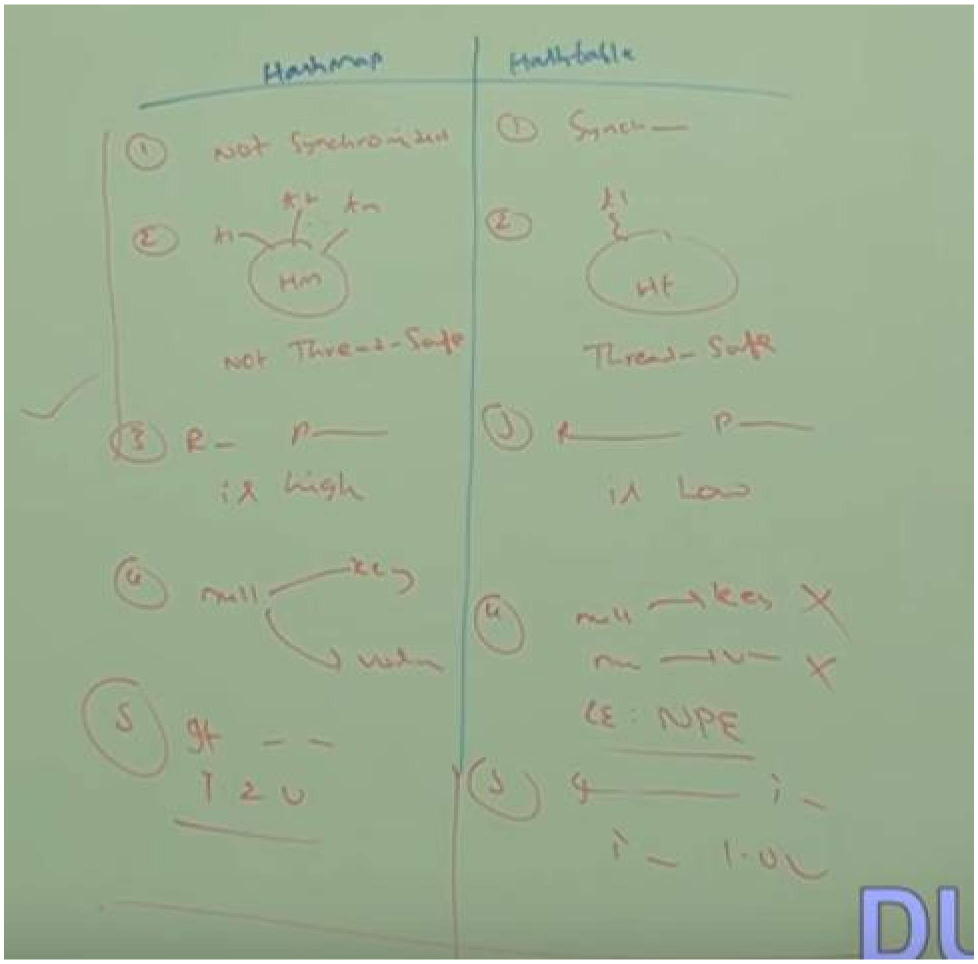


```
System.out.println(m);// {K=V,K=V,..}
System.out.println(m.put("chiranjeevi", 1000));//700

Set s = m.keySet();
System.out.println(s);// [k,k,..]

Collection c = m.values();
System.out.println(c);

Set s1 = m.entrySet();
System.out.println(s1);// [K=V,K=V,..]
Iterator itr = s1.iterator();
while(itr.hasNext())
{
   Map.Entry m1 = (Map.Entry)itr.next();
   System.out.println(m1.getKey() + "......."
   +m1.getValue());
   if(m1.getKey().equals("nagarjuna"))
   {
      m1.setValue(10000);
   }
```



```
Iterator itr = s1.iterator();
while(itr.hasNext())
{
   Map.Entry m1 = (Map.Entry)itr.next();
   System.out.println(m1.getKey() + "......."
   +m1.getValue());
   if(m1.getKey().equals("nagarjuna"))
   {
      m1.setValue(10000);
   }
}
System.out.println(m);
```

Q. Differences between HashMap and HashTable ?



Every method present in HashMap is non syncronized.

Every method present in HashTable is syncronized.

At a time multiple threads are allowed to operate on HashMap Object and hence it is not Thread safe.

At a time only one thread is allowed to operate on HashTable and hence it is thread safe.

Relatively performance is high because threads are not required to wait to operate on HashMap Object.

Relatively performacnce is low because threads are required to wait to operate on HashTable object.

Null is allowed for both key and value.

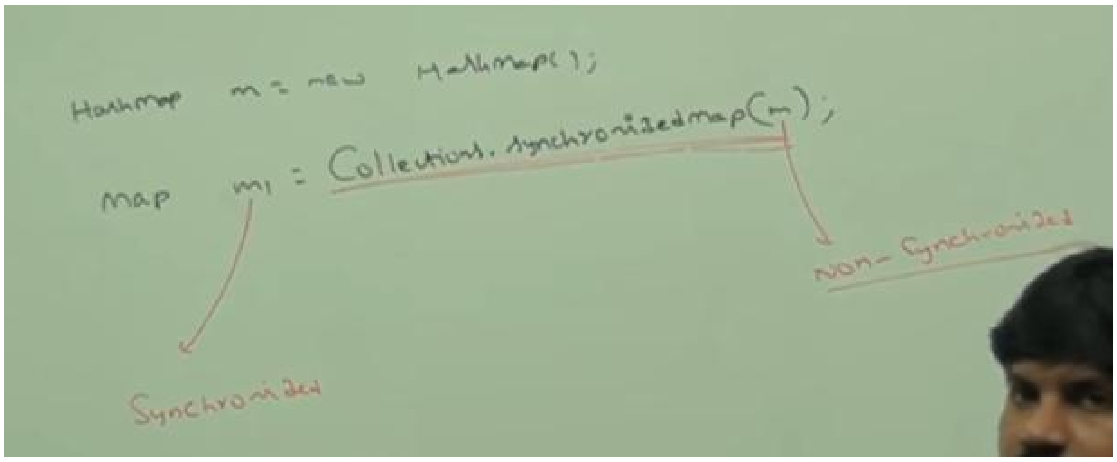
Null is not allowed for keys and values, othrwise we will get nullpointerException.

Introduced in 1.2 version and it is not legacy.

Introduced in 1.0 version and it is legacy.

Q. How to get syncronized version of HashMap Object ?

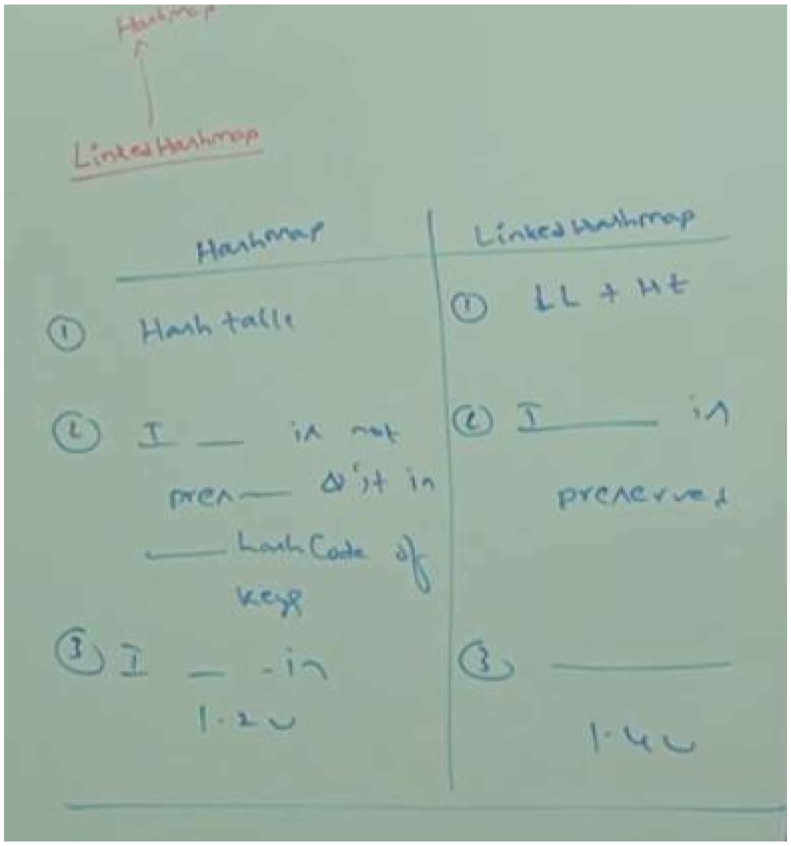
By default HashMap is non syncronized but we can get syncronized version of HashMap by using syncronizedMap(), method of Collections class.



```
HashMap   m = new   HashMap();
  Map   m1 = Collections.synchronizedMap(m);
```

Synchronized
non-Synchronized

LinkedHashMap:



HashMap
LinkedHashMap

HashMap	LinkedHashMap
① Hash table	① LL + Ht
② I ___ is not pres___ b'z it is ___ hash Code of keys	② I ___ is preserved
③ I ___ in 1.2 v	③ ___ 1.4 v

It is the child class of HashMap.

It is exactly same as HashMap(including methods and constructors except the above differences)

1. The underlying data structure is HashTable.

1. Underlying data structure is combination of Linked List and Hash Table.

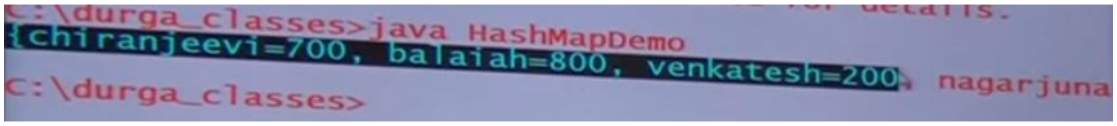
2. Insertion Order is not preserved and it is based on HashCode of keys.

2. Insertion Order is preserved.

3. Introduced in 1.2 version

3. introduced in 1.4 version

In the above HashMap program if we replace HashMap with Linked HashMap then out out is,



```
c:\durga_classes>java HashMapDemo
{chiranjeevi=700, balaiah=800, venkatesh=200, nagarjuna
c:\durga_classes>
```

Note:

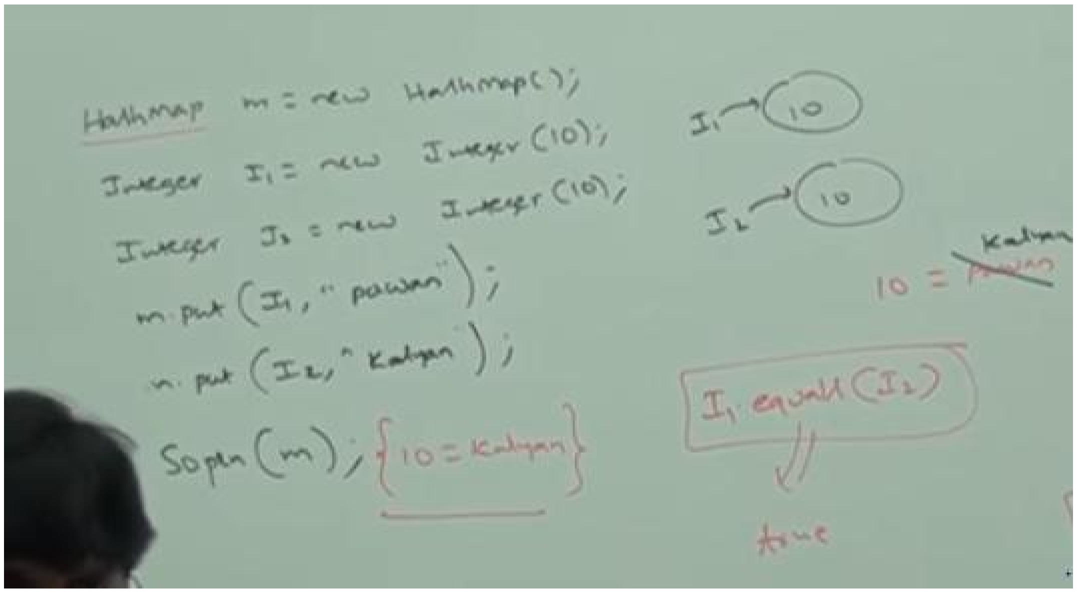
LinkedHashSet or LinkedHashMap are commonly used for developing cache based applications.

IdentityHashMap:

It is exactly same as HashMap, including methods and constructors, except the following differences:

In the case of normal HashMap JVM will use .equals(), method to identify duplicate keys, which is ment for content comparison.

but in the case of identity HashMap, JVM will use == operator to identify duplicate keys, which is ment for reference comarison.(Address comparison).



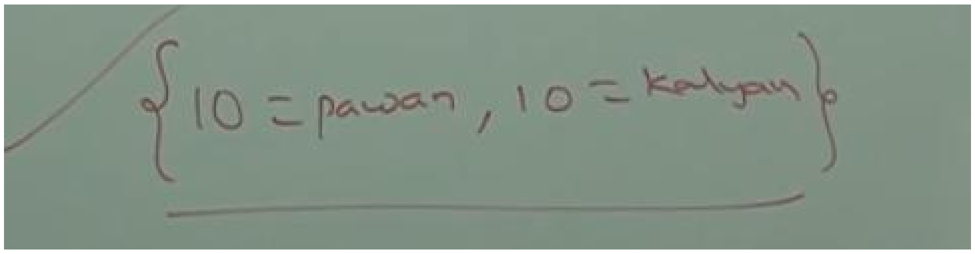
```
HashMap m = new HashMap();
Integer I1 = new Integer(10);
Integer I2 = new Integer(10);
m.put(I1, "pawan");
m.put(I2, "kalyan");
Sopln(m); {10=kalyan}
```

I1 → 10
I2 → 10
10 = ~~pawan~~ kalyan
I1.equals(I2)
⇓
true

I1 and I2 are duplicate keys because i1.equals(i2), returns true.

If we replace HashMap with IdentityHashMap, then I1 and I2 are not duplicate keys because I1 ==i1, returns false.

In this case output is



```
{10=pawan, 10=kalyan}
```

WeakHashMap:

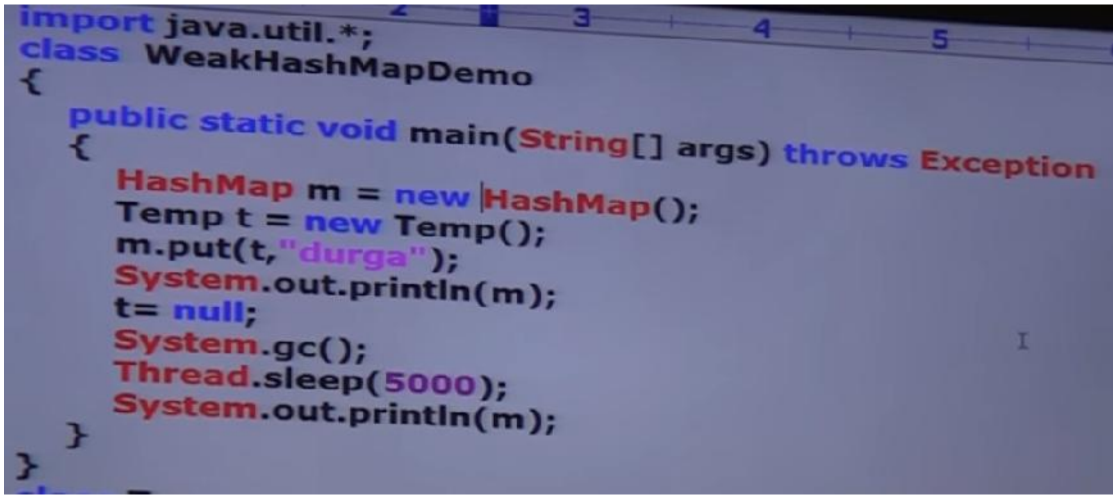
It is exactly same as HashMap except the following difference,

In the case of HashMap, even though object does not have any refereence it is not eligible for GC, if it is associated with Hashmap.

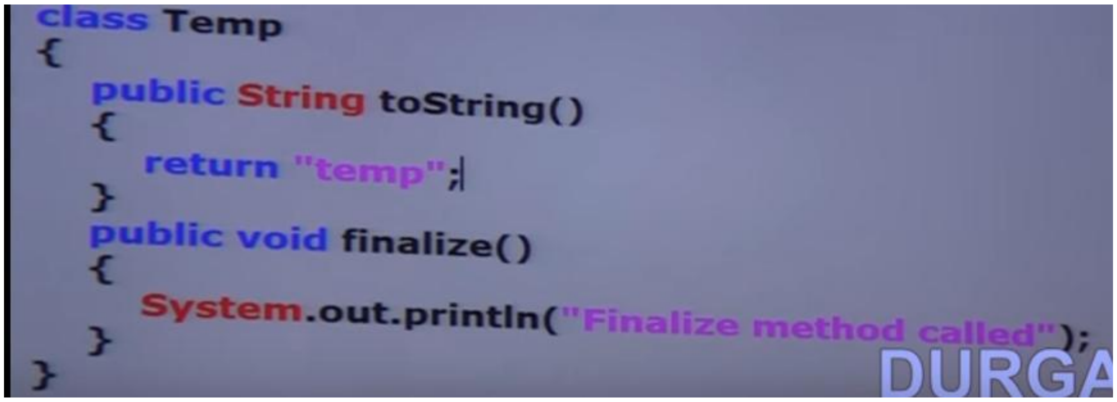
i.e. HashMap dominates GC.

But in the case of WeakHashMap, if Object does not contain any references it is eligibel for GC even though Object associated with WeakHashMap.

i.e. GC dominates WeakHashMap.



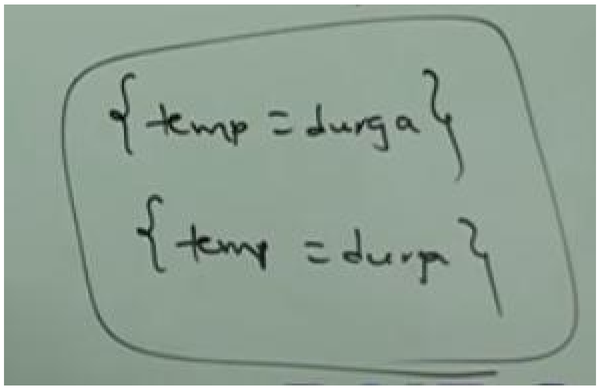
```
import java.util.*;
class WeakHashMapDemo
{
    public static void main(String[] args) throws Exception
    {
        HashMap m = new HashMap();
        Temp t = new Temp();
        m.put(t,"durga");
        System.out.println(m);
        t= null;
        System.gc();
        Thread.sleep(5000);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



```
class Temp
{
    public String toString()
    {
        return "temp";
    }
    public void finalize()
    {
        System.out.println("Finalize method called");
    }
}
```

In the above example Temp Object not eligible for GC because it is associated with HashMap.

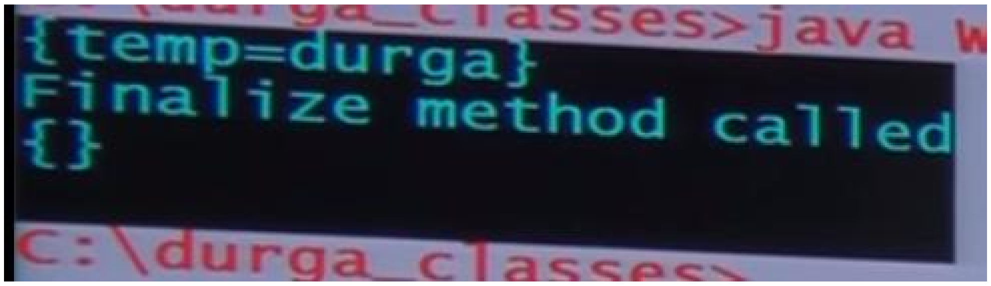
In this case output is,



```
{temp=durga}
{temp=durga}
```

In the above program if we replace HashMap with WeakHashMap, then Temp object is eligible for GC.

In this case output is,



```
ga_classes>java W
{temp=durga}
Finalize method called
{}

C:\durga_classes>
```

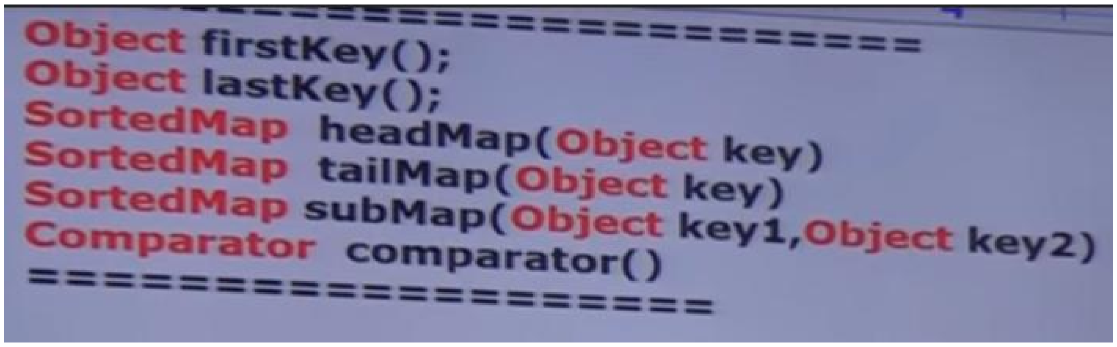
SortedMap(I):

It is the child interface of Map.

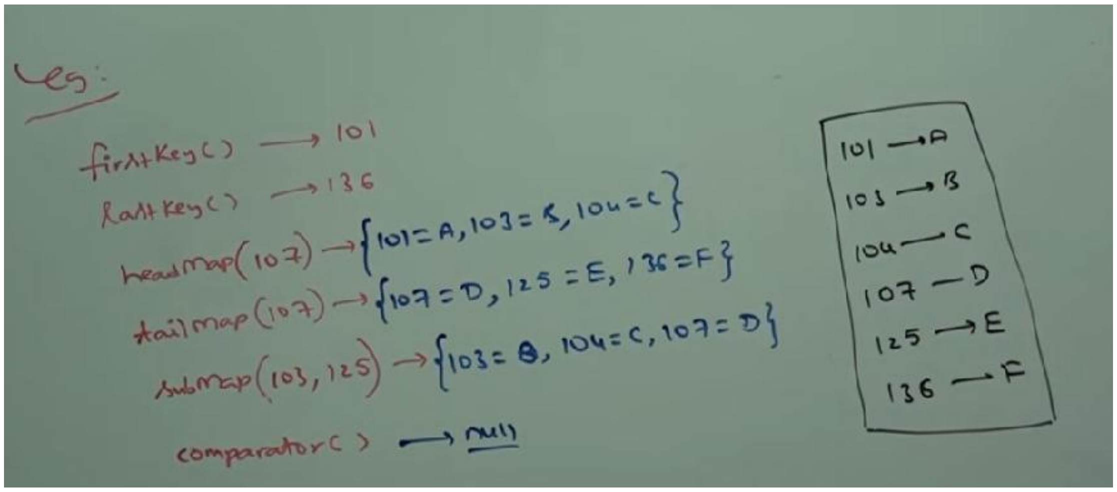
If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap.

Sorting is based on the key but not based o value.

SortedMap defines the following specific methods,



```
====================
Object firstKey();
Object lastKey();
SortedMap headMap(Object key)
SortedMap tailMap(Object key)
SortedMap subMap(Object key1,Object key2)
Comparator comparator()
====================
```



eg:

```
firstKey()  → 101
lastKey()   → 136
headMap(107) → {101=A,103=B,104=C}
tailMap(107) → {107=D,125=E,136=F}
subMap(103,125) → {103=B, 104=C,107=D}
comparator() → null
```

```
101 → A
103 → B
104 → C
107 → D
125 → E
136 → F
```

TreeMap:

The underlying data structure is Red-black tree.

Insertion order is not preserved and it is based on some sorting order of keys.

Duplicate keys are not allowed but values can be duplicated.

If we are depending on default natural sorting order then keys should be homogeneous and comparable, otherwise we will get runtime exception saying classCastException.

If we are defining our own sorting by comarator then keys need not be homogeneous and comparable. We can take heterogeneous and non comparable objects also.

Whether we are depending on default natural sorting order or customized sorting order there are no restrictions for values. We can take hterogeneous non-comparable objects also.

Null Accepatance:

1. For non empty tree map, if we are trying to insert an entry with null key, then we will get runtime exception saying nullpointer exception.

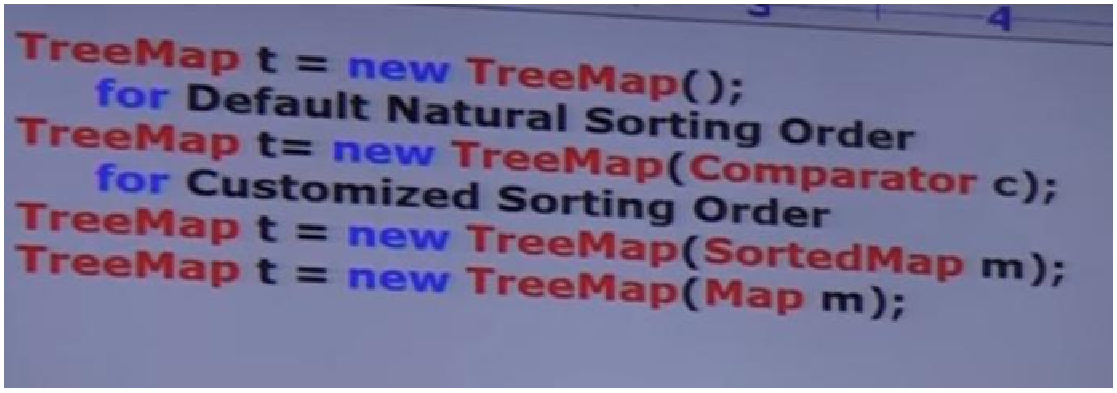
2. For empty treemap as the first entry with the null key is allowed, but after inserting that entry if we are trying to insert any other entry then we will get runtime exception saying nullpointer exception.

Note:

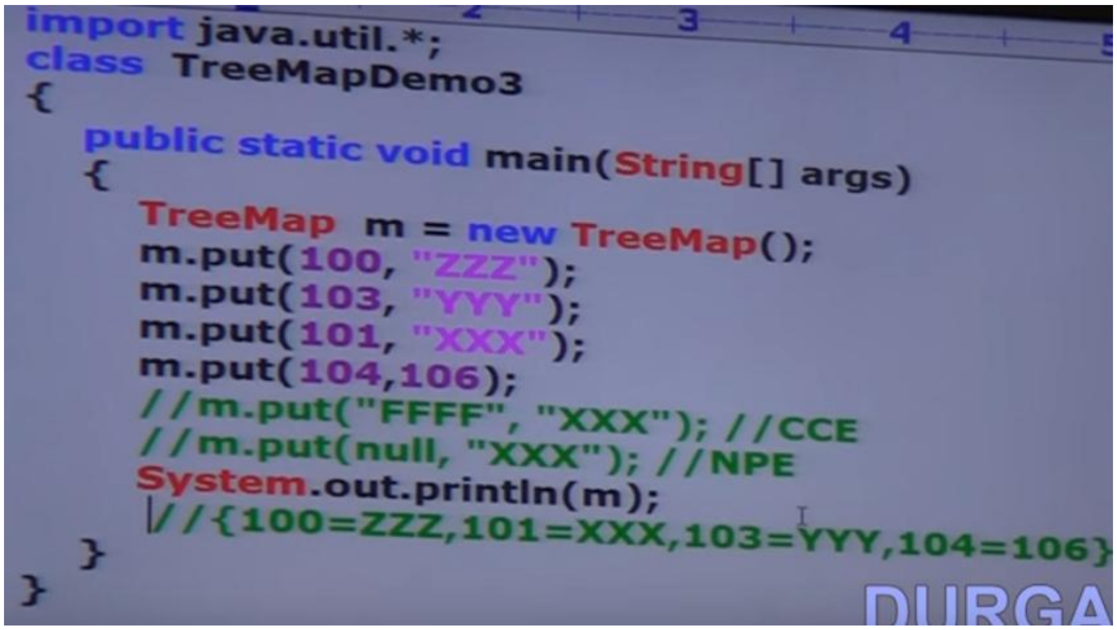
The above nullacceptance rule applicable until 1.6 version only. From 1.7 version onwards null is not allowed for key.

But for values we can use null any numer of times. There is no restriction whether it is 1.6 version or 1.7 version.

Constructors:

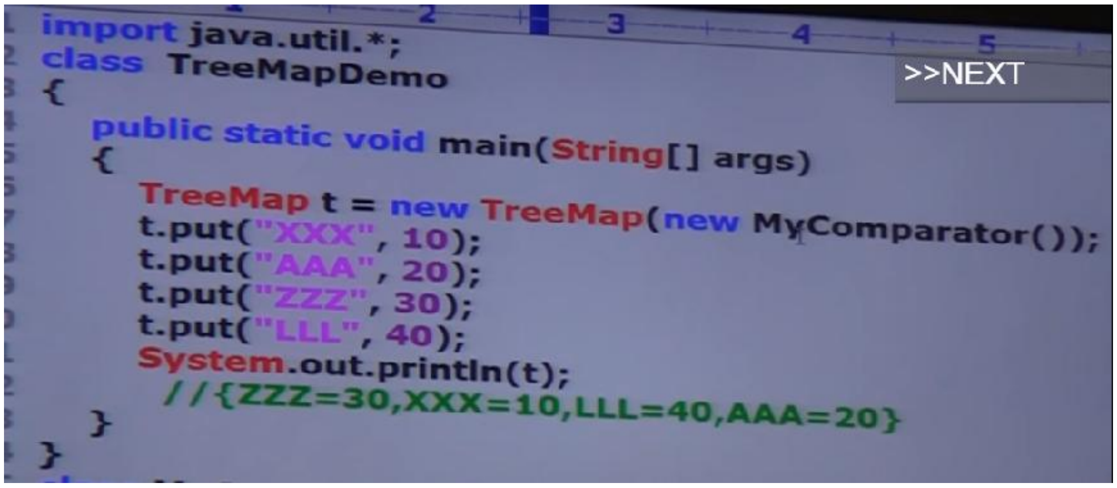


```
TreeMap t = new TreeMap();
    for Default Natural Sorting Order
TreeMap t= new TreeMap(Comparator c);
    for Customized Sorting Order
TreeMap t = new TreeMap(SortedMap m);
TreeMap t = new TreeMap(Map m);
```

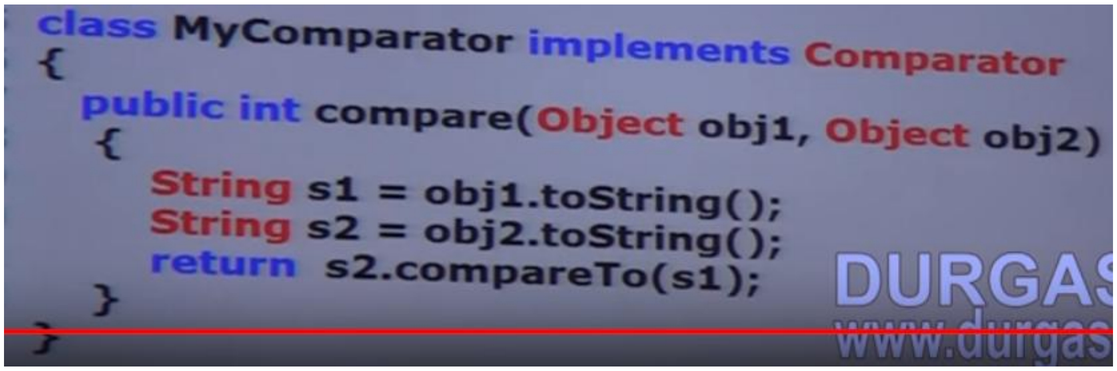


```
import java.util.*;
class TreeMapDemo3
{
    public static void main(String[] args)
    {
        TreeMap  m = new TreeMap();
        m.put(100, "ZZZ");
        m.put(103, "YYY");
        m.put(101, "XXX");
        m.put(104,106);
        //m.put("FFFF", "XXX"); //CCE
        //m.put(null, "XXX"); //NPE
        System.out.println(m);
        //{100=ZZZ,101=XXX,103=YYY,104=106}
    }
}
```

Customized Sorting:



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

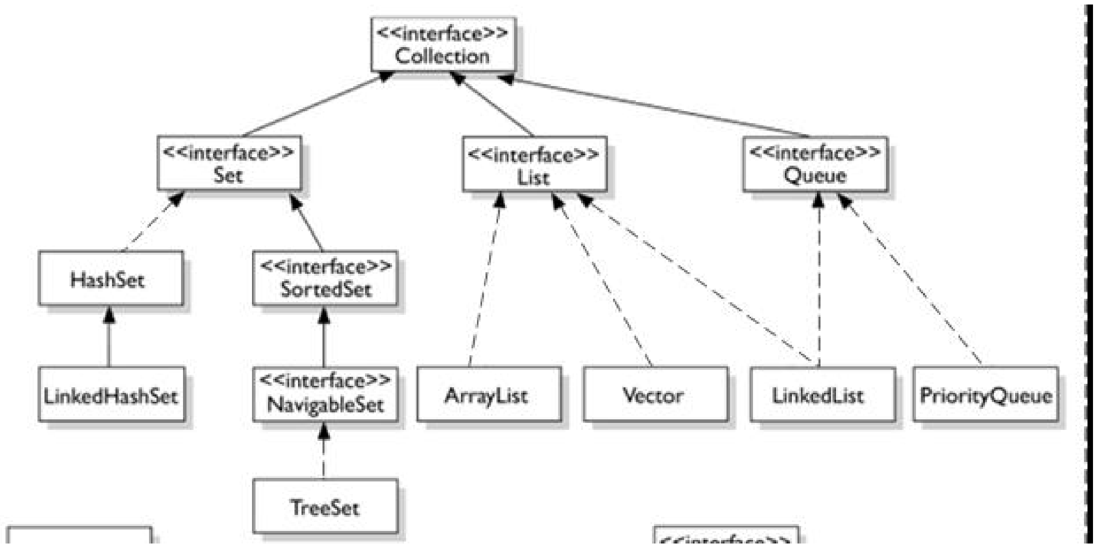


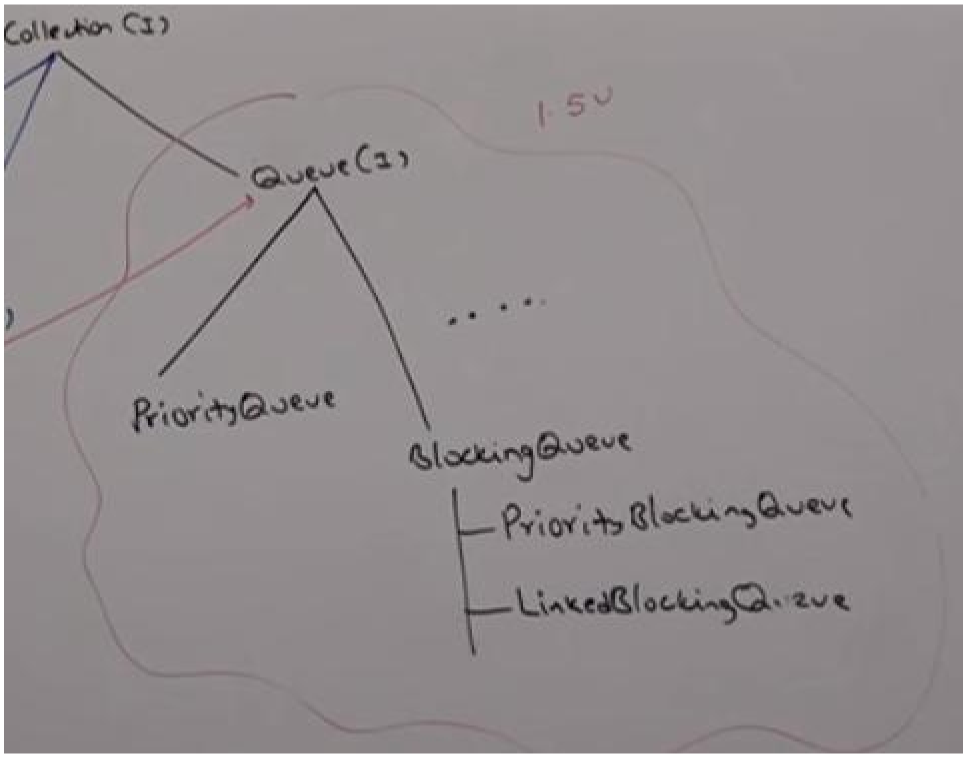
```
class MyComparator implements Comparator
{
  public int compare(Object obj1, Object obj2)
  {
     String s1 = obj1.toString();
     String s2 = obj2.toString();
     return  s2.compareTo(s1);
  }
}
```

Queue:

1.5 version Enhancements (Queue Interface)

It is the child interface of collection.





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

e.g.

Before sending SMS message all mobile numbers we have to store in some data structure.

In which order we added mobile numbers, in the same order only message should be sent.

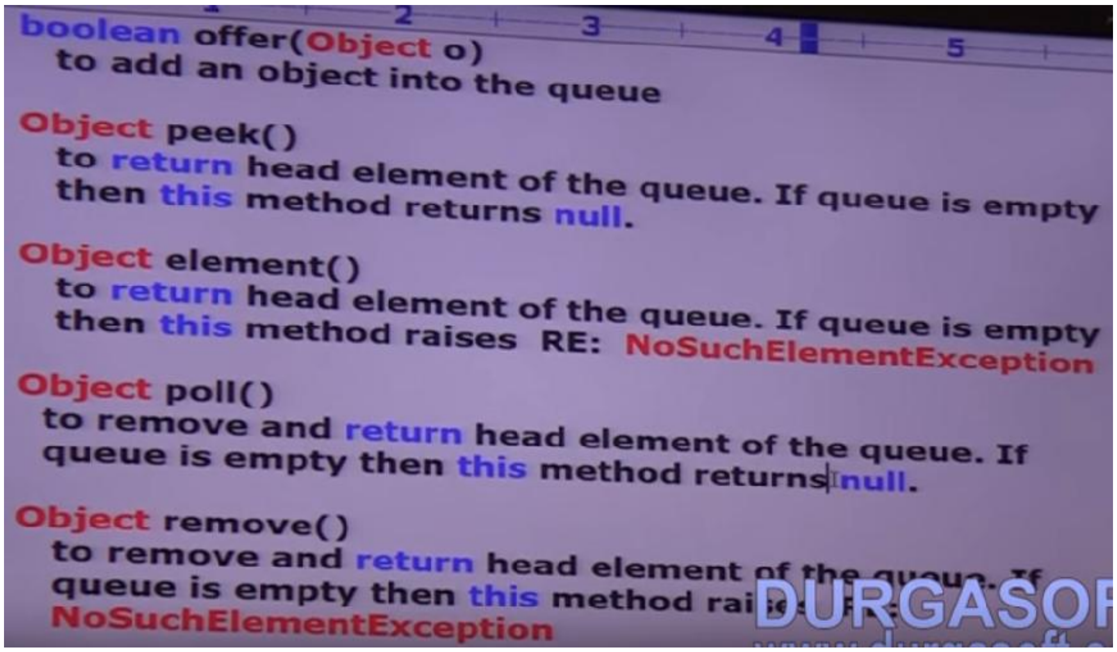
For this First-in-First out requirement Queue is the best choice.

Usually Queue follows First-in-First out order but based on our requirement we can implement our own priority order also.(ProrityQueue)

From 1.5 version onwards LinkedList class also implements Queue Interface.

LinkedList based implementation of Queue always follows FIFO order.

Queue Interface Specific Methods:


boolean offer(Object o)
to add an object into the queue

Object peek()
to return head element of the queue. If queue is empty then this method returns null.

Object element()
to return head element of the queue. If queue is empty then this method raises RE: NoSuchElementException

Object poll()
to remove and return head element of the queue. If queue is empty then this method returns null.

Object remove()
to remove and return head element of the queue. If queue is empty then this method raises RE:
NoSuchElementException

PriorityQueue:

prior to processing..

If we want to represent a group of individual objects prior to processing according to some priority then we should go for priority queue.

The priority can be either defaut natural sorting order or customized sorting order defined by comparator

Insertion order is not preserved and it is based on some prority.

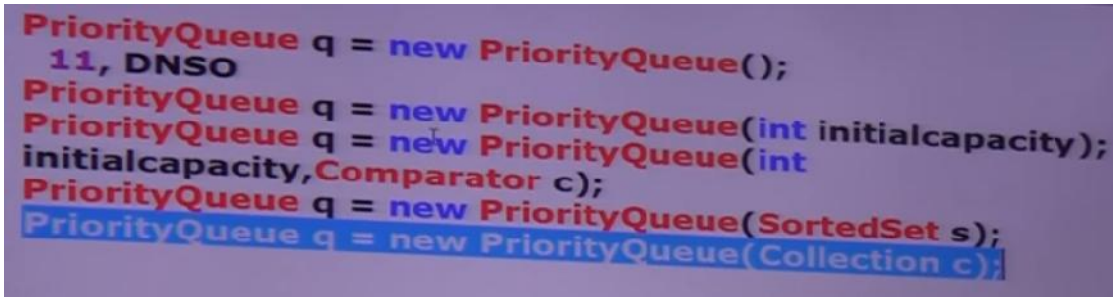
Duplicate Objects are not allowed.

If we are depending on default natural sorting order compulsory the objects should be homogeneous and comparable, otherwise we will get runtime exception saying classCastException.

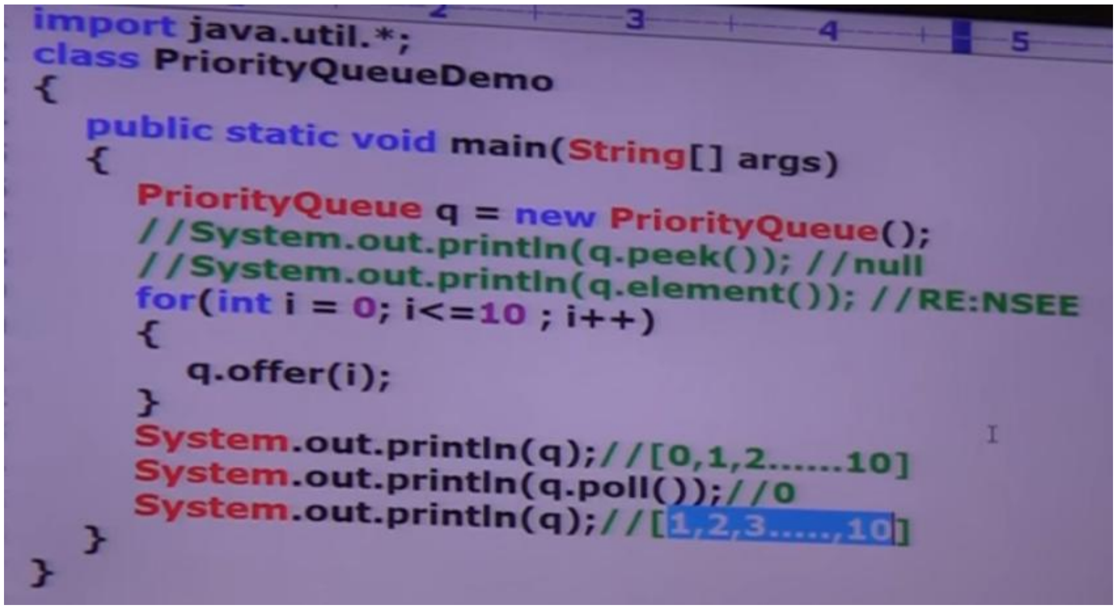
If we are defining our own sorting by comparator then Objects need not be homogeneous and comparable.

Null is not allowed even as the first element also.

Constructors:

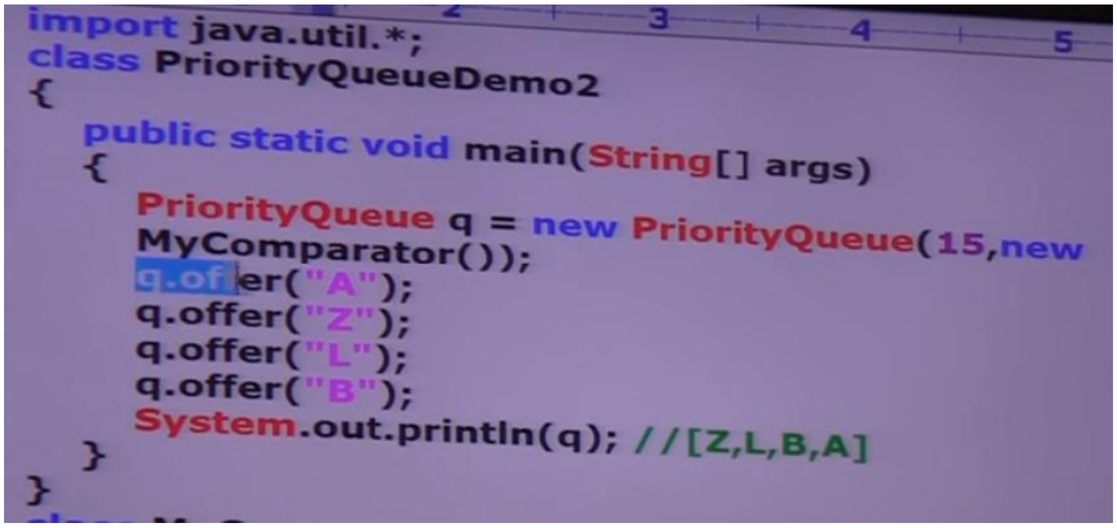


```
PriorityQueue q = new PriorityQueue();
 11, DNSO
PriorityQueue q = new PriorityQueue(int initialcapacity);
PriorityQueue q = new PriorityQueue(int
initialcapacity,Comparator c);
PriorityQueue q = new PriorityQueue(SortedSet s);
PriorityQueue q = new PriorityQueue(Collection c);
```

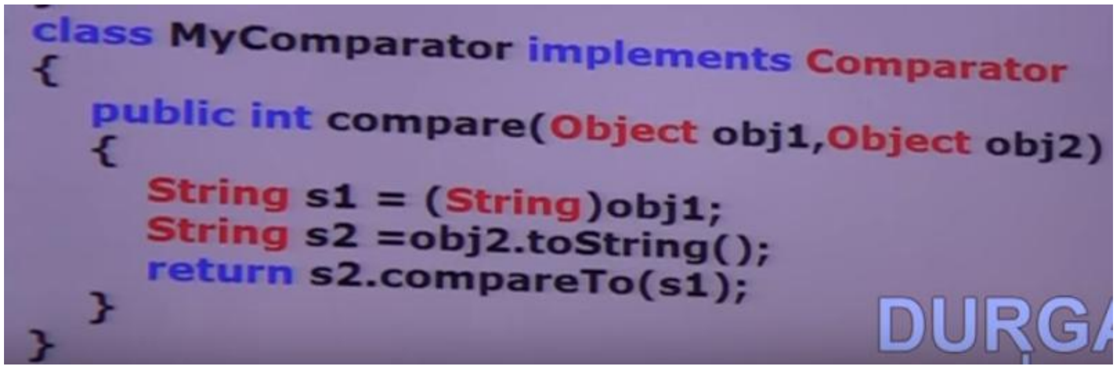


```
import java.util.*;
class PriorityQueueDemo
{
   public static void main(String[] args)
   {
      PriorityQueue q = new PriorityQueue();
      //System.out.println(q.peek()); //null
      //System.out.println(q.element()); //RE:NSEE
      for(int i = 0; i<=10 ; i++)
      {
         q.offer(i);
      }
      System.out.println(q);//[0,1,2......10]
      System.out.println(q.poll());//0
      System.out.println(q);//[1,2,3.....,10]
   }
}
```

Some platforms won't provide proper support for priority queues.



```
import java.util.*;
class PriorityQueueDemo2
{
   public static void main(String[] args)
   {
      PriorityQueue q = new PriorityQueue(15,new
      MyComparator());
      q.offer("A");
      q.offer("Z");
      q.offer("L");
      q.offer("B");
      System.out.println(q); //[Z,L,B,A]
   }
}
```



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

1.6 version enhancements in Collection Framework:

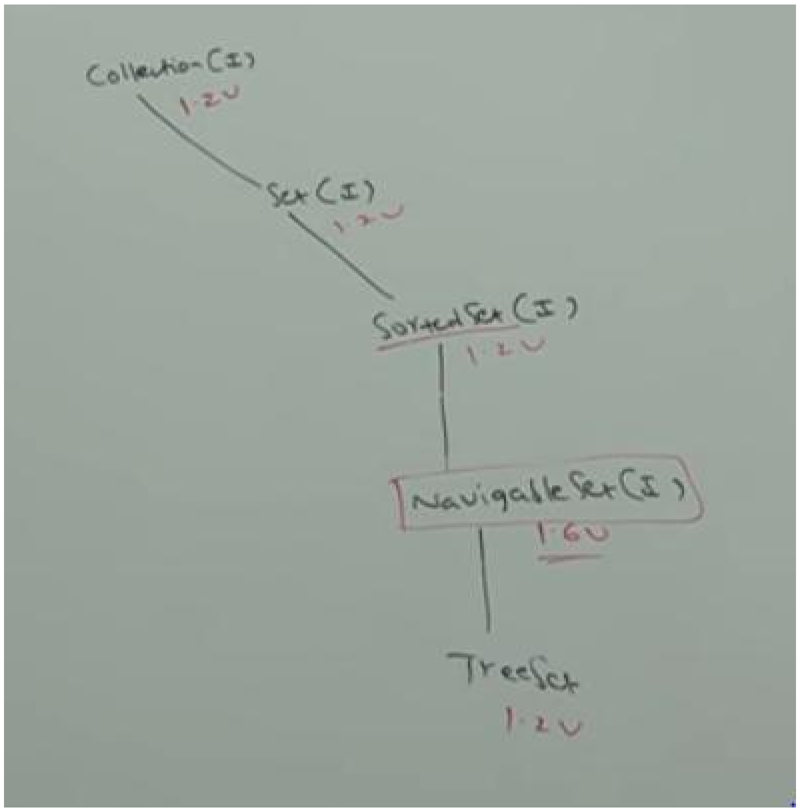
As a part of 1.6 version the following two concepts intoduced in collection framework,

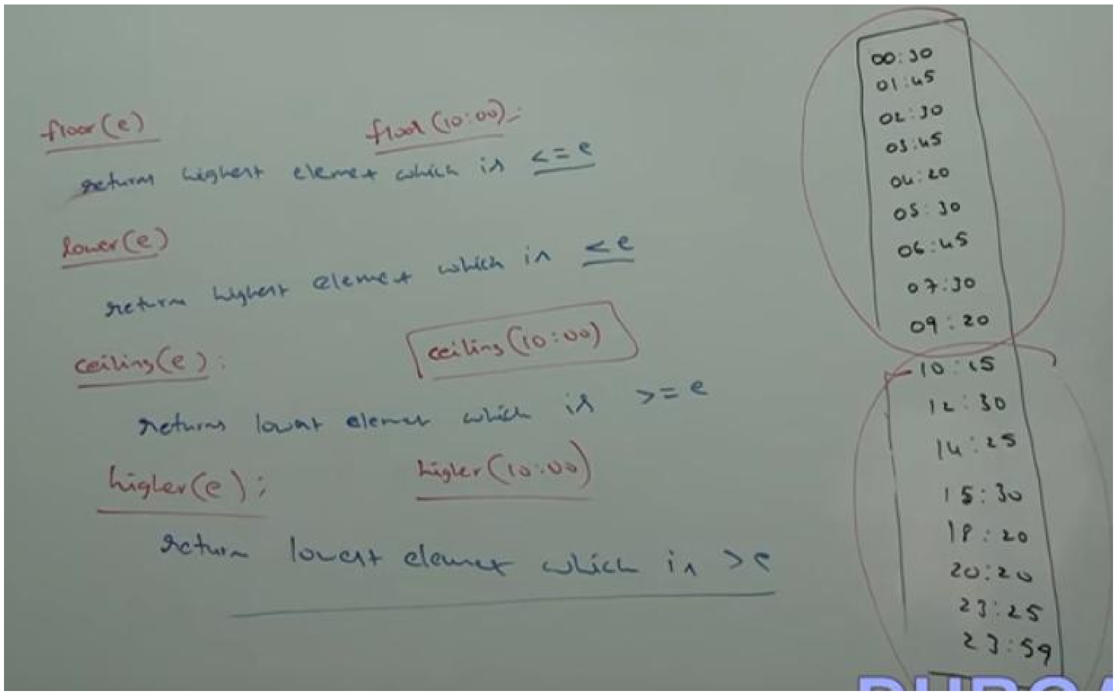
1. NavigableSet(I)

2. NavigableMap(I)

NavigableSet:

It is the child interface of SortedSet and it defines several methods for navigation purposes.





floor(e) floor(10:00)

return highest element which is <= e

lower(e)

return highest element which is < e

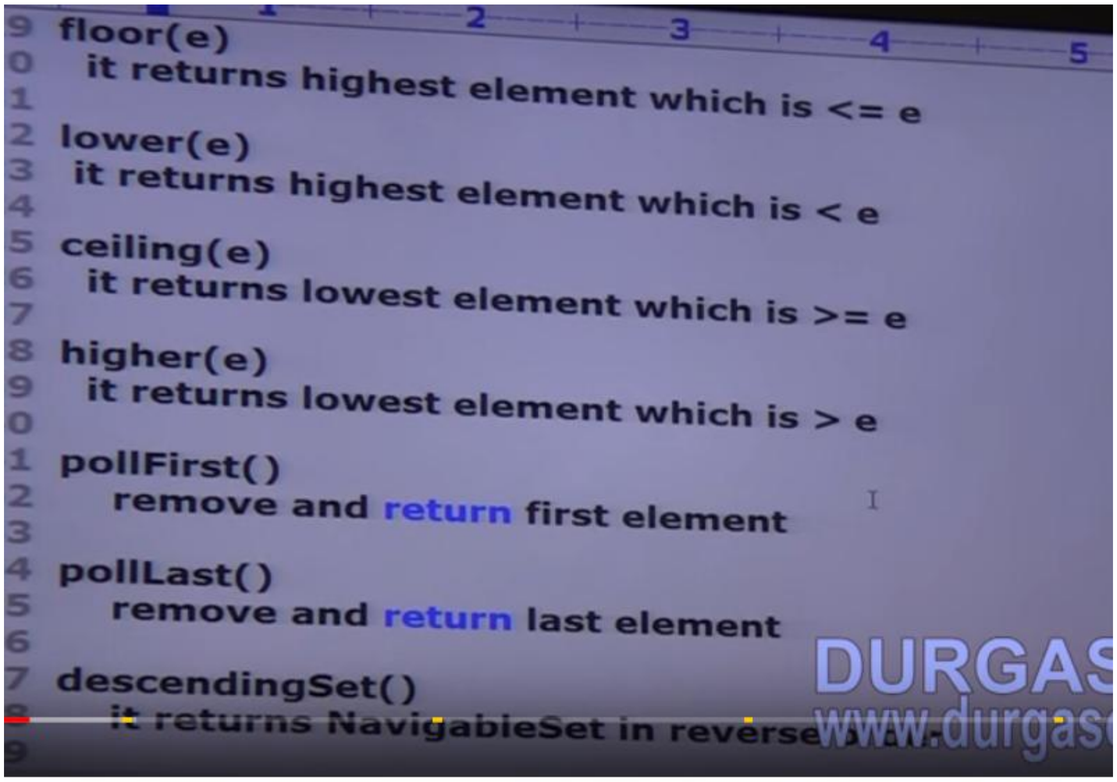
ceiling(e): ceiling(10:00)

returns lowest element which is >= e

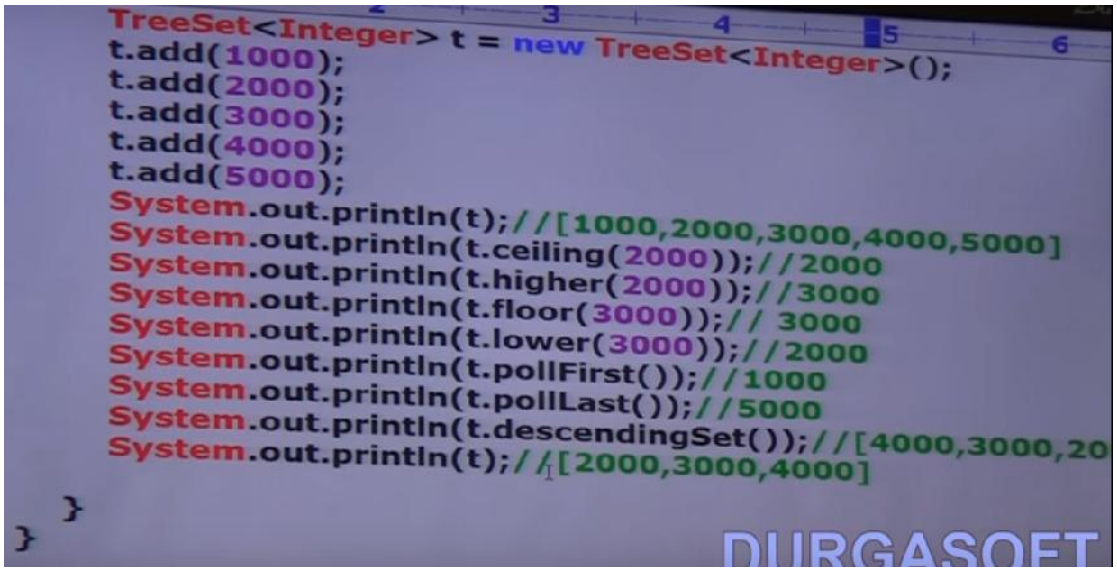
higher(e); higher(10:00)

return lowest element which is > e

00:30
01:45
02:30
03:45
04:20
05:30
06:45
07:30
09:20
10:15
12:30
14:25
15:30
18:20
20:20
23:25
23:59



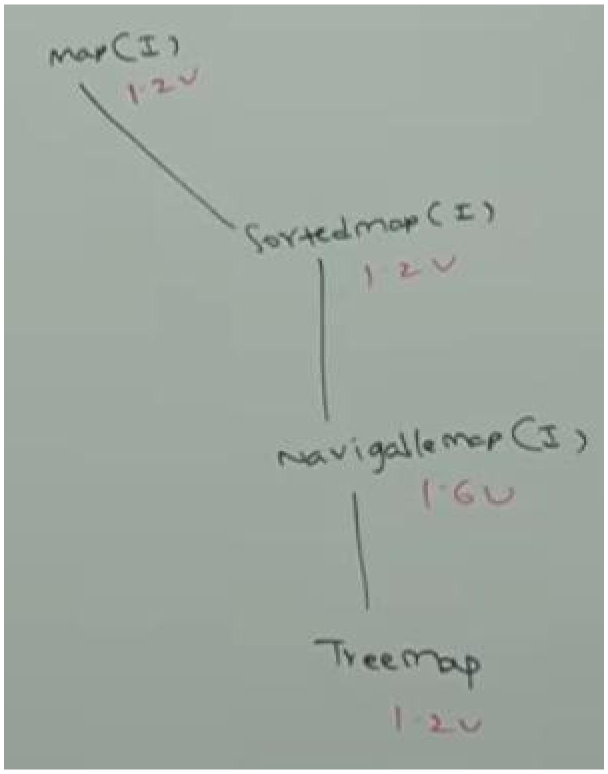
```
floor(e)
  it returns highest element which is <= e

lower(e)
 it returns highest element which is < e

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

higher(e)
  it returns lowest element which is > e

pollFirst()
    remove and return first element

pollLast()
    remove and return last element

descendingSet()
    it returns NavigableSet in reverse
```



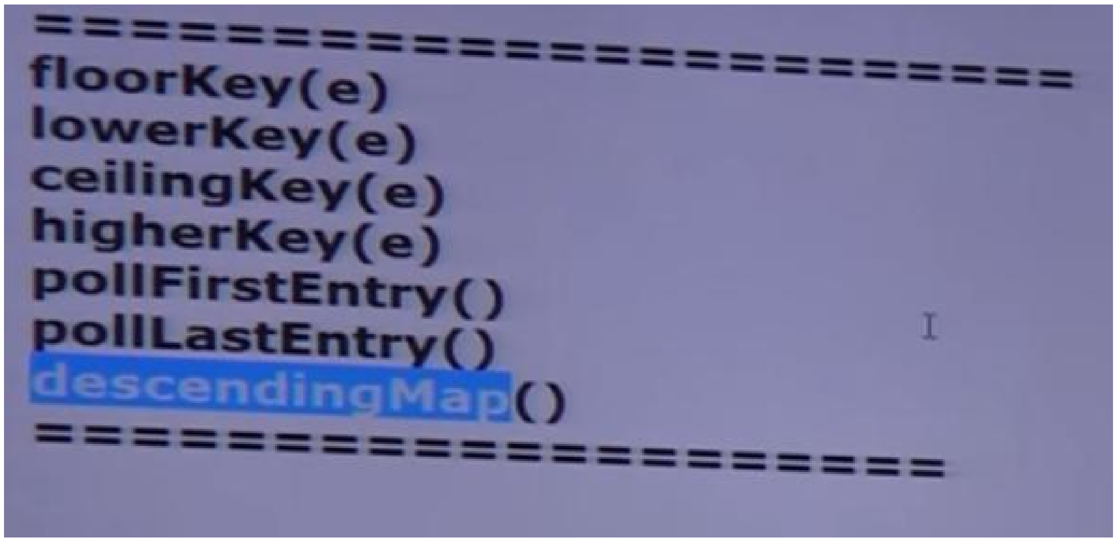
```
    TreeSet<Integer> t = new TreeSet<Integer>();
    t.add(1000);
    t.add(2000);
    t.add(3000);
    t.add(4000);
    t.add(5000);
    System.out.println(t);//[1000,2000,3000,4000,5000]
    System.out.println(t.ceiling(2000));//2000
    System.out.println(t.higher(2000));//3000
    System.out.println(t.floor(3000));// 3000
    System.out.println(t.lower(3000));//2000
    System.out.println(t.pollFirst());//1000
    System.out.println(t.pollLast());//5000
    System.out.println(t.descendingSet());//[4000,3000,20
    System.out.println(t);//[2000,3000,4000]

  }
}
```

NavigableMap:

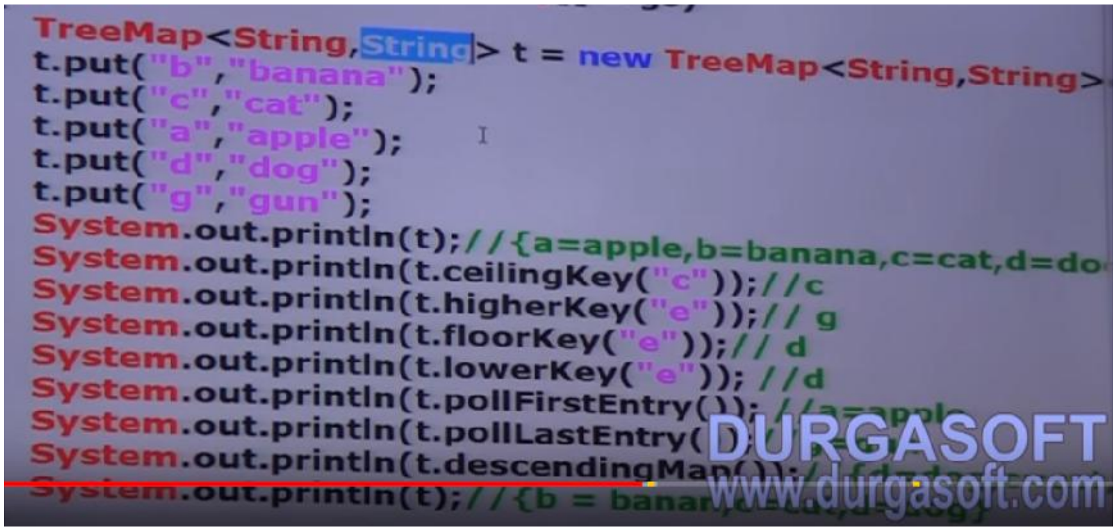
NavigableMap is the child interface of sortedMap. It defines several methods for navigation purposes.



NavigableMap defines the following methods



```
==========================
floorKey(e)
lowerKey(e)
ceilingKey(e)
higherKey(e)
pollFirstEntry()
pollLastEntry()
descendingMap()
======================
```



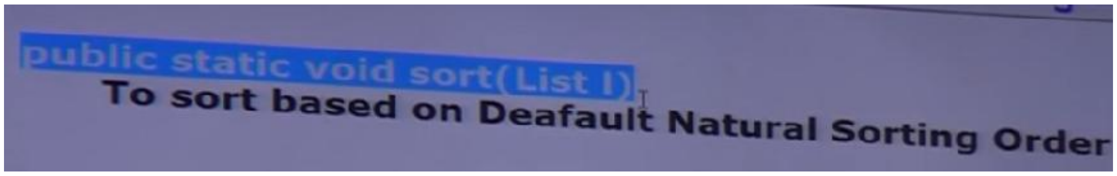
```
TreeMap<String,String> t = new TreeMap<String,String>
t.put("b","banana");
t.put("c","cat");
t.put("a","apple");
t.put("d","dog");
t.put("g","gun");
System.out.println(t);//{a=apple,b=banana,c=cat,d=do
System.out.println(t.ceilingKey("c"));//c
System.out.println(t.higherKey("e"));// g
System.out.println(t.floorKey("e"));// d
System.out.println(t.lowerKey("e")); //d
System.out.println(t.pollFirstEntry()); //a=appl
System.out.println(t.pollLastEntry(
System.out.println(t.descendingMap());
System.out.println(t);//{b = banan
```

Collections(c):

Collections class defines several utility methods for collection objects, like sorting, searching, reversing etc..

Sorting Elements of List:

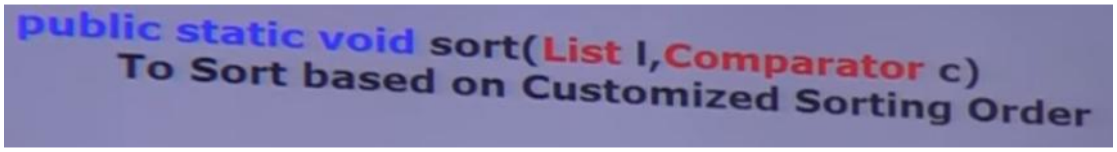
Collections class defines the following two sort(), methods



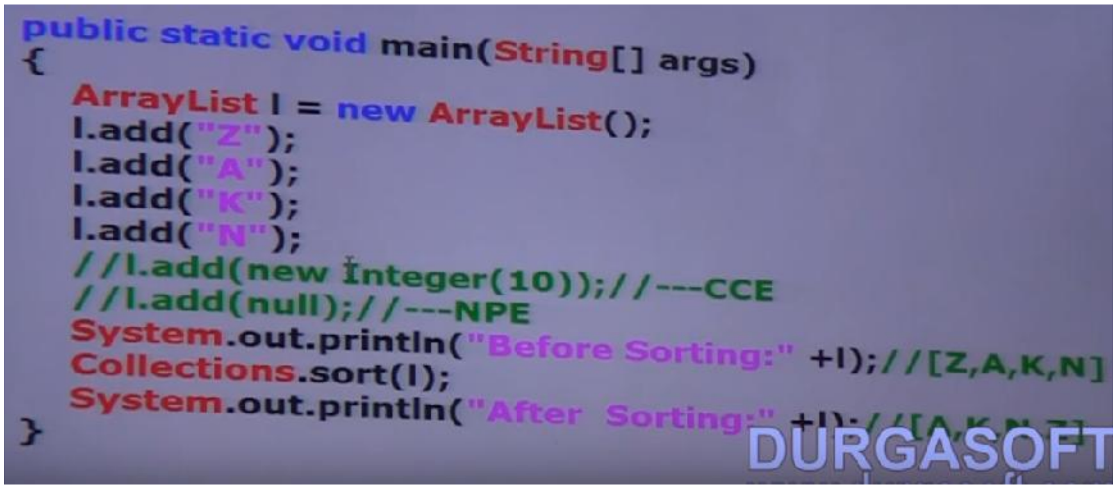
```
public static void sort(List l)
    To sort based on Deafault Natural Sorting Order
```

In this case List should compulsory contain homogeneous and comparable objects, otherwise we will get RuntimeException saying ClassCastException.

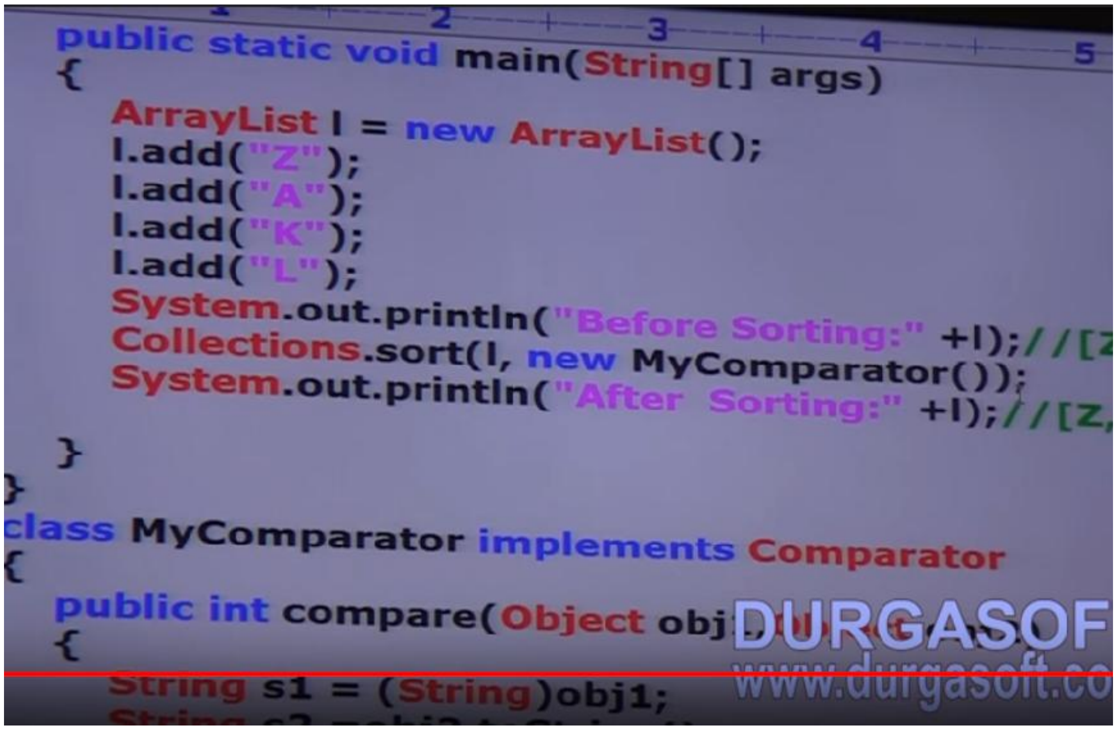
List should not contain null, otherwise we will get nullPointerException.



```
public static void sort(List l,Comparator c)
    To Sort based on Customized Sorting Order
```



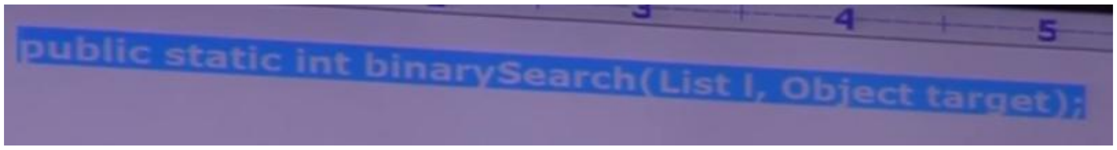
```
public static void main(String[] args)
{
   ArrayList l = new ArrayList();
   l.add("Z");
   l.add("A");
   l.add("K");
   l.add("N");
   //l.add(new Integer(10));//---CCE
   //l.add(null);//---NPE
   System.out.println("Before Sorting:" +l);//[Z,A,K,N]
   Collections.sort(l);
   System.out.println("After  Sorting:" +l);//[A,K,N,Z]
}
```



```
  public static void main(String[] args)
  {
     ArrayList l = new ArrayList();
     l.add("Z");
     l.add("A");
     l.add("K");
     l.add("L");
     System.out.println("Before Sorting:" +l);//[Z
     Collections.sort(l, new MyComparator());
     System.out.println("After  Sorting:" +l);//[Z,

  }
}
class MyComparator implements Comparator
{
  public int compare(Object obj
  {
     String s1 = (String)obj1;
```

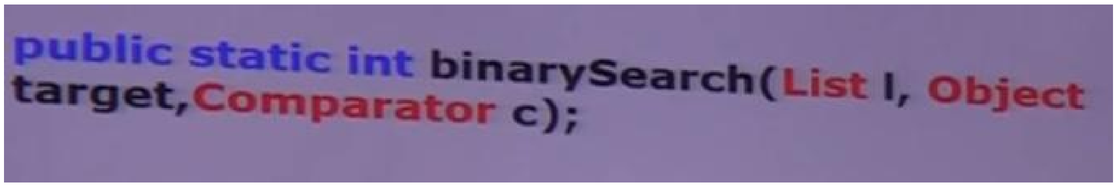
Searching Elements of List:

Collections class defines the following binary search methods



```
public static int binarySearch(List l, Object target);
```

If the list is sorted accoding to default natural sorting order then we have to use this method.



```
public static int binarySearch(List l, Object
target,Comparator c);
```

We have to use this method if the list is sored according to customized sorting order.

Conclusions:

1. The above search methods internally will use binary search algorithem.

2. Successful search returns index

3. Unsuccessful search returns insertion point.

4. Insertion point is the location where we can place target element in the sorted list.

5. Befor calling binary search method compulsory list should be sorted, otherwise we will get unpredictable results.

6. If the list is sorted according to comparator then at the time of search operation also we have to pass same comparator object, otherwise we will get unpredictable results.