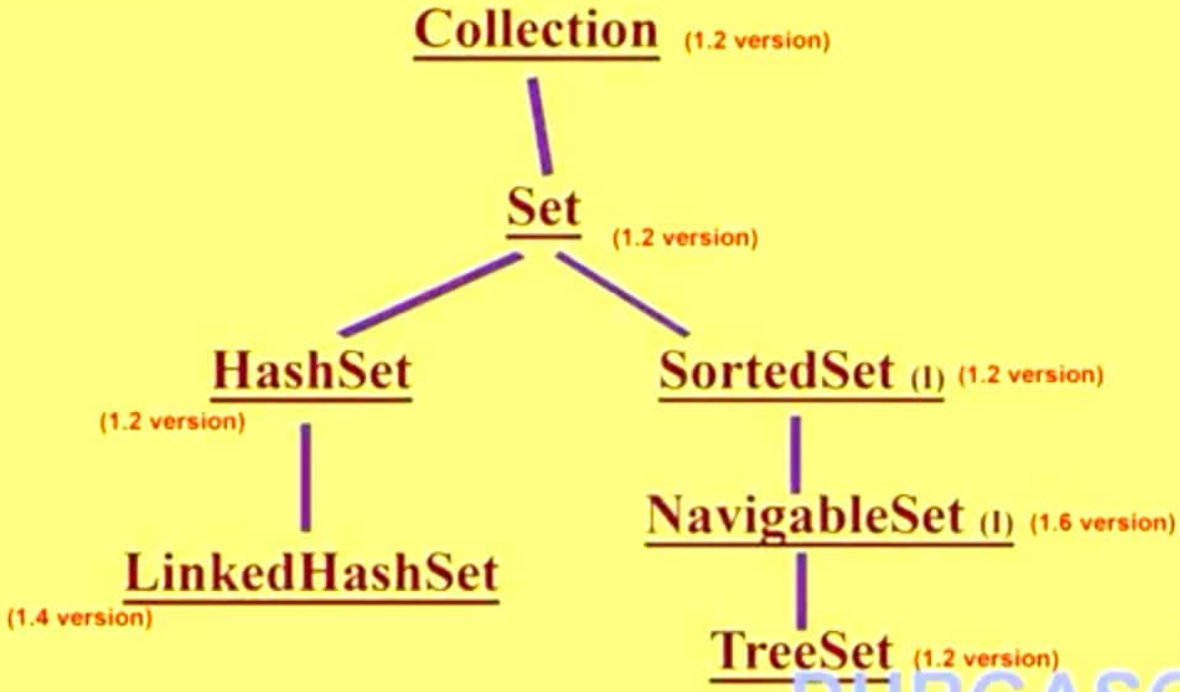
Set Interface :



Set

1. Set is the child interface of Collection.

2. If we want to represent a group of individual objects as a single entity, where duplicates are not allowed and insertion order is not preserved then we should go for Set.

3. Set interface doesn't contain any new methods. So we have to use only Collection interface methods.

HashSet

* The underlying data structure is Hashtable.
* Duplicates are not allowed. If we are trying to insert duplicates, we won't get any compiletime or runtime errors. add() method simply returns false.
* Insertion order is not preserved and all objects will be inserted based on hash-code of objects.
* Heterogeneous objects are allowed.
* ' null ' insertion is possible.
* implements Serializable and Clonable interfaces but not RandomAccess.
* HashSet is the best choice, if our frequent operation is Search operation.

1) HashSet h = new HashSet();

- Creates an empty HashSet object with default initial capacity 16 & default Fill Retio 0.75

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

- Creates an empty HashSet object with specified initial capacity & default Fill Retio 0.75

3) HashSet h = new HashSet(int initalCapacity, float loadFactor);

- Creates an empty HashSet object with specified initial capacity & specified Load Factor (or Fill Ratio)

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

- For inter conversion between Collection objects.

Load Factor / Fill Ratio :

- After loading the how much factor, a new HashSet object will be created, that factor is called as Load Factor or Fill Ratio.

Demo program for HashSet

```
import java.util.*;
class HashSetDemo {
public static void main (String [] args) {
    HashSet h=new HashSet ();
    h.add("B");
    h.add("C");
    h.add("D");
    h.add("Z");
    h.add(null);
    h.add(10);
    System.out.println (h.add("Z"));   // false
    System.out.println (h);  // [null, D, B, C, 10, Z]
} }
```

HashSet	LinkedHashSet
The underlying datastructure is Hash table.	The underlying datastructure is Hash table + Linked List . (that is hybrid data structure)

```
import java.util.*;
class LinkedHashSetDemo {
public static void main (String [] args) {
    LinkedHashSet h=new LinkedHashSet ();
    h.add("B");
    h.add("C");
    h.add("D");
    h.add("Z");
    h.add(null);
    h.add(10);
    System.out.println (h.add("Z"));   // false
    System.out.println (h);  // [B, C, D, Z, null, 10]
} }
```

LinkedHashSet

Note :

- LinkedHashSet is the best choice to develop cache based applications, where duplicates are not allowed and insertion order must be preserved.

SortedSet (I)

1. It is the child interface of set.

2. If we want to represent a group of individual objects according to some sorting order and duplicates are not allowed then we should go for SortedSet.

Object first() - returns first element of the SortedSet

Object last() - returns last element of the SortedSet

SortedSet headSet(Object obj) - returns the SortedSet whose elements are < obj

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

SortedSet subSet(Object obj1, Object obj2)

- returns the 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.

Example : { 100,101,103,104,107,110,115 }

1. first() -> 100
2. last() -> 115
3. headset(104) -> [100,101,103]
4. tailSet(104) -> [104,107,110,115]
5. subset(103,110) -> [103,104,107]
6. comparator() -> null

Note :

1. Default natural sorting order for numbers Ascending order and for String alphabetical order.
2. We can apply the above methods only on SortedSet implemented class objects. That is on the TreeSet object.

TreeSet

1. The underlying data structure for TreeSet is Balanced Tree.
2. Duplicate objects are not allowed.
3. Insertion order not preserved, but all objects will be inserted according to some sorting order.
4. Heterogeneous objects are not allowed. If we are trying to insert heterogeneous objects then we will get runtime exception saying ClassCastException.
5. Null Insertion is allowed, but only once.

TreeSet Constructors

1. TreeSet t=new TreeSet();
   - Creates an empty TreeSet object where elements will be inserted according to default natural sorting order.

2. TreeSet t=new TreeSet(Comparator c);
   - Creates an empty TreeSet Object where elements will be inserted according to customized sorting order.

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

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

Example

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

Null Acceptance

1. For empty TreeSet as the first element null insertion is possible.
   But After inserting that null if we are trying to insert any another element we will get NullPointerException.

2. For Non empty TreeSet If we are trying to insert Null then we will get NullPointerExcpetion.

Example

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

Note :

1. If we are depending on default natural sorting order then objects should be homogeneous and comparable. Otherwise we will get runtime exception saying ClassCastException.
2. An object is Said to be comparable if and only if the corresponding class implements java.lang.comparable interface.
3. String Class and all wrapper classes already implements comparable interface. But StringBuffer doesn't implement comparable interface.

** Hence in the above program we got ClassCastException **

* This interface present in java.lang package it contains only one method CompareTo().

```
public int  CompareTo(Object obj)
```

Example :

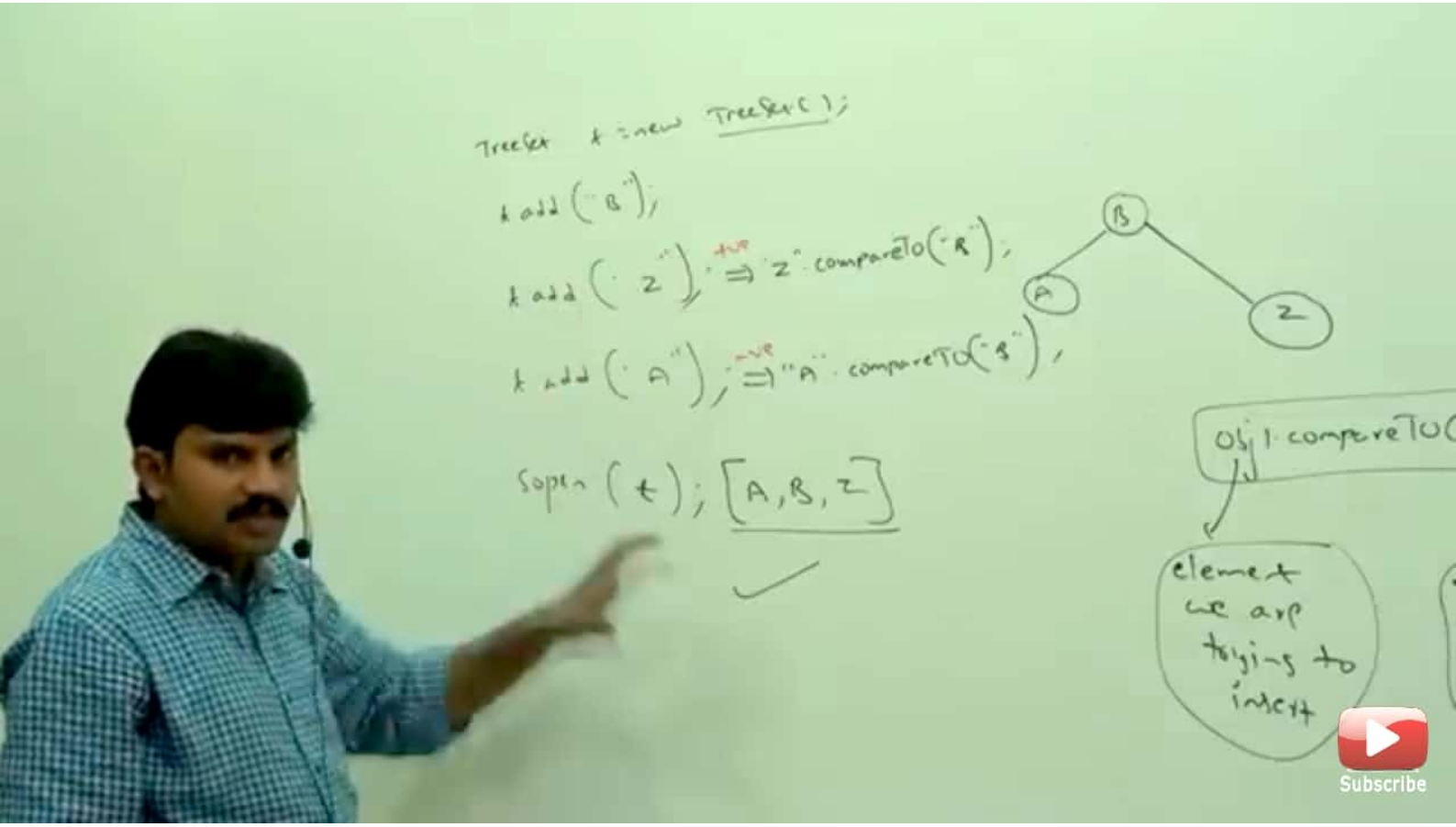
obj1.CompareTo(obj2)

- |—> returns –ve iff obj1 has to come before obj2
- |—> returns +ve iff obj1 has to come after obj2
- |—> returns 0 iff obj1 & obj2 are equal.

Example

```
class Test {
    public static void main(String[] args) {
        System.out.println("A".compareTo("Z"));  // -ve
        System.out.println("Z".compareTo("B"));   // +ve
        System.out.println("A".compareTo("A"));   // 0
        System.out.println("A".compareTo(null));   // NullPointerException

    }
}
```



* If we depending on default natural sorting order internally JVM will call CompareTo() method will inserting objects to the TreeSet. Hence the objects should be Comparable.

```
TreeSet t = new TreeSet();
t.add("B");
t.add("Z");      // "Z".compareTo("B");  +ve
t.add("A");      // "A".compareTo("B");  -ve
System.out.println(t);       // [A, B, Z]
```

Note :

1. If we are not satisfied with default natural sorting order or if the default natural sorting order is not already available then we can define our own customized sorting by using Comparator.

2. Comparable ment for Default Natural Sorting order where as Comparator ment for customized Sorting order.

Comparator Interface

* We can use comparator to define our own sorting (Customized sorting).
* Comparator interface present in java.util package.
* It defines two methods. compare and equals.

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.

2) public boolean equals();

DURGASOFT

* When ever we are implementing Comparator interface, compulsory we should provide implementation for compare() method.

* And implementing equals() method is optional, because it is already available in every java class from Object class through inheritance.

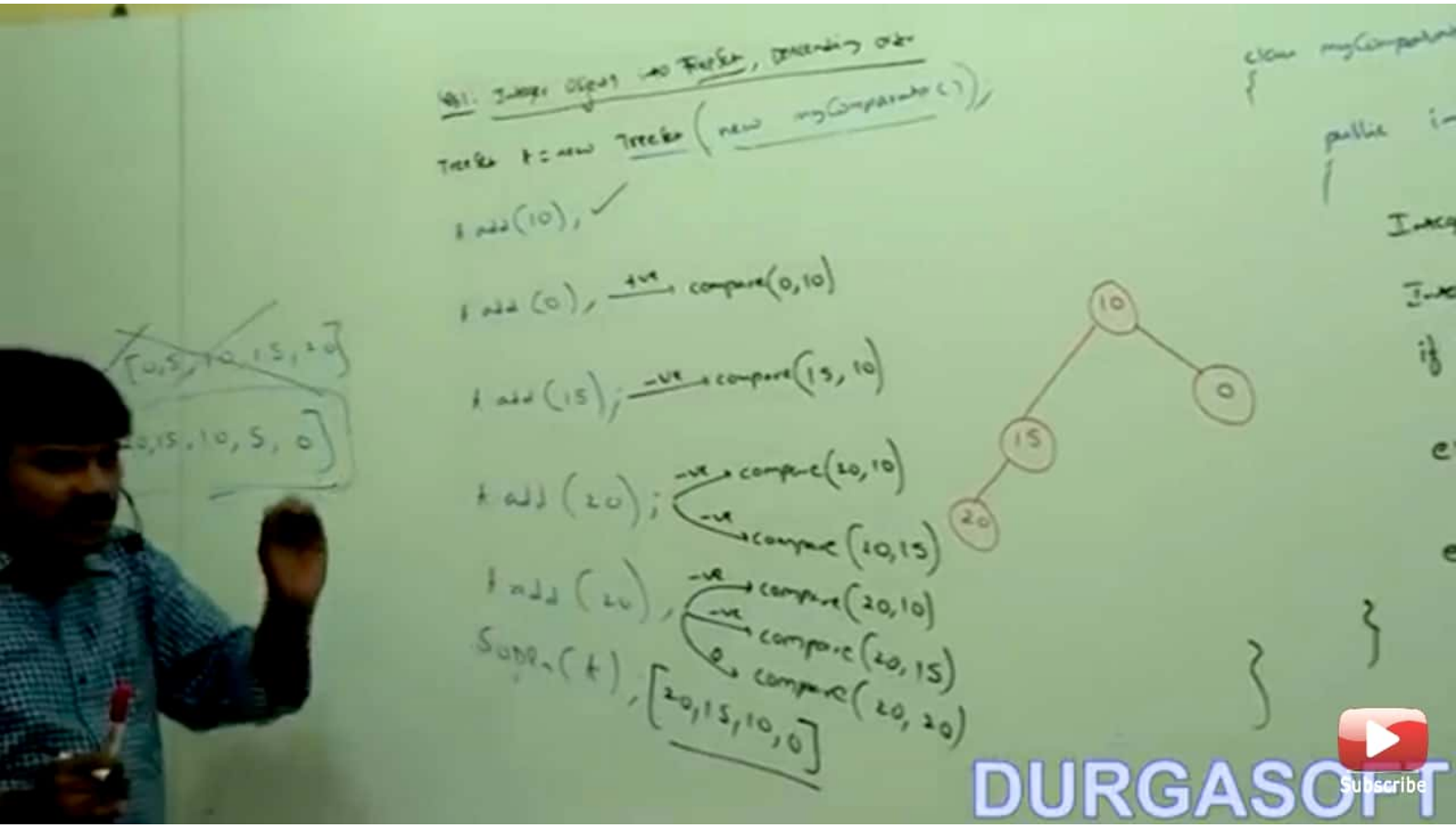
is descending order :

```
import java.util.*;
class TreeSetDemo3 {
  public static void main(String[] args) {
    TreeSet t = new TreeSet(new MyComparator()); -- line1
    t.add(10);
    t.add(0);
    t.add(15);
    t.add(20);t
    t.add(20);
    System.out.println(t);
  }
}
```

```
class MyComparator implements Comparator  {
    public int compare(Object obj1, Object obj2) {
        Integer I1=(Integer)obj1;
        Integer i2=(Integer)obj2;
        if(I1<I2)
            return +1;
        else if(I1>I2)
            return -1;
         else
            return 0;
    }
}
```

Output: [20,15,10,0]

DURGASOFT



```
class MyComparator implements Comparator  {
    public int compare(Object obj1, Object obj2) {
        Integer I1=(Integer)obj1;
        Integer i2=(Integer)obj2;
        // return I1 CompareTo(I2);    [0,10,15,20] ascending order
        // return -I1 CompareTo(I2);   [20,15,10,0] descending order
        // return I2 CompareTo(I1);    [20,15,10,0] descending order
        // return -I2 CompareTo(I1);   [0,10,15,20] ascending order
        // return +1                   [10,0,15,20,20] Insertion order
        // return -1                   [20,20,15,0,10] Reverse of Insertion order
        // return 0;                   [10]
         (only first element will be inserted and all the other elements are considered as duplicates)
```

Reverse of Alphabetical order :

```
import java.util.*;
class TreeSetDemo2 {
  public static void main(String[] args) {
    TreeSet t = new TreeSet(new MyComparator());
    t.add("Roja");
    t.add("ShobhaRani");
    t.add("RajaKumari");
    t.add("GangaBhavani");
    t.add("Ramulamma");
    System.out.println(t);
  }
}
```

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

Output: [ShobhaRani, Roja, Ramulamma, RajaKumari, GangaBhavani]

order is Alphabetical order:

```
import java.util.*;
class TreeSetDemo10 {
  public static void main(String[] args) {
    TreeSet t = new TreeSet(new MyComparator());
    t.add(new StringBuffer("A"));
    t.add(new StringBuffer("Z"));
    t.add(new StringBuffer("K"));
    t.add(new StringBuffer("L"));
    System.out.println(t);
  }
}
```

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

Output: [A, K, L, Z]

Note :

* If we are defining our own sorting by Comparator, the objects need not be Comparable.

```
import java.util.*;
class TreeSetDemo12 {
  public static void main(String[] args) {
    TreeSet t = new TreeSet(new MyComparator());
    t.add("A");
    t.add(new StringBuffer("ABC"));
    t.add(new StringBuffer("AA"));
    t.add("XX");
    t.add("ABCD");
    t.add("A");
    System.out.println(t);
  }
}
```

Output: [A,AA,XX,ABC,ABCD]

```
class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2) {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        int l1=s1.length();
        int l2=s2.length();
        if(l1<l2)
            return -1;
        else if(l1>l2)
            return 1;
        else
        return s1.compareTo(s2);
    }
}
```

Note :

1. If we are depending on default natural sorting order then objects should be homogeneous and comparable otherwise we will get runtime exception saying ClassCastException.

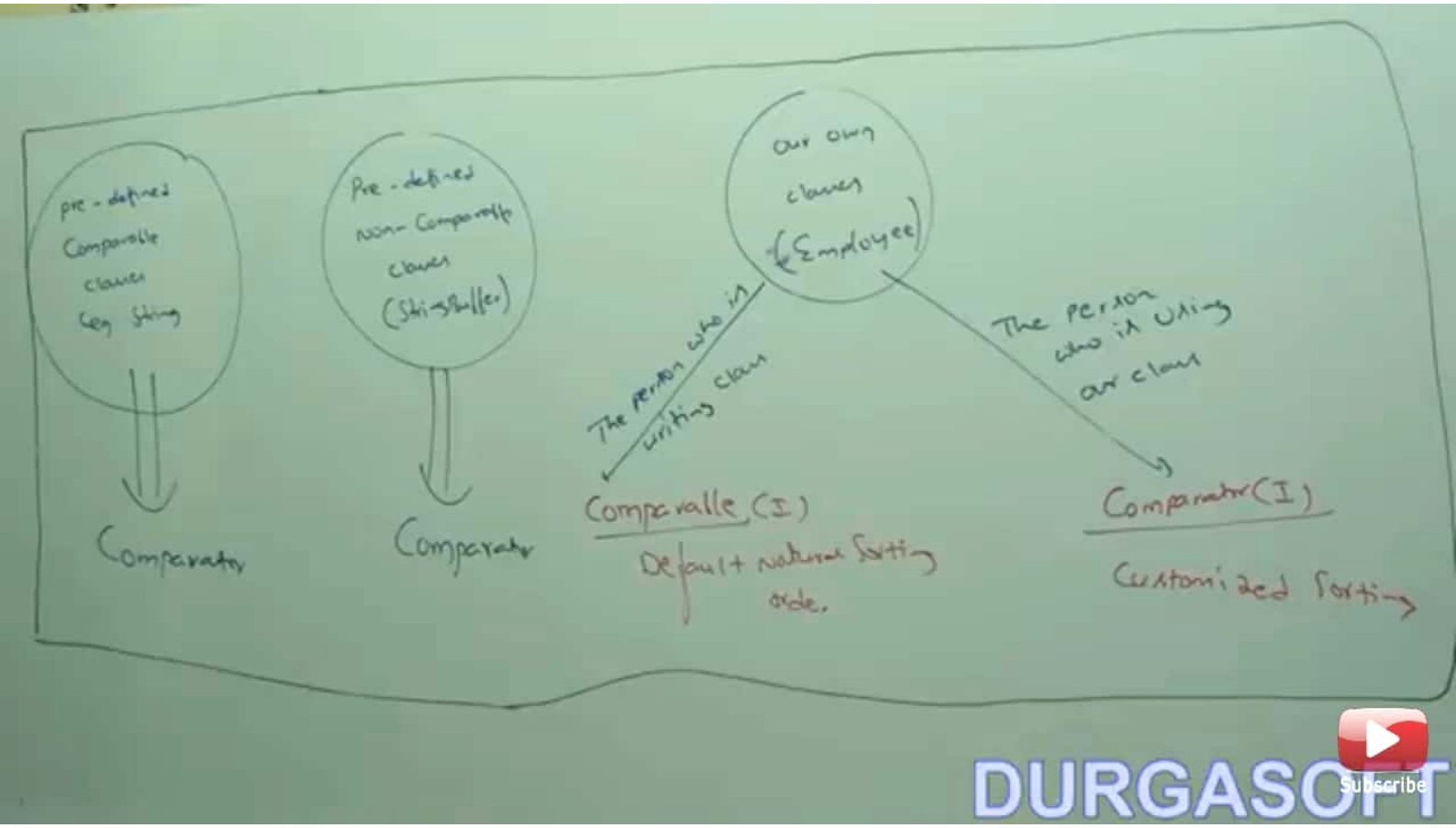
2. But if we are defining our own sorting by comparator then objects need not be homogeneous and comparable. we can insert heterogeneous non comparable objects also.

1) For predefined Comparable classes like String default natural sorting order already available. If we are not satisfied with that, we can define our own sorting by Comparator object.

2) For Predefined non comparable classes like StringBuffer, default natural sorting order is not already available. We can define required sorting by implementing Comparator interface.

3) For our own classes like Employee, Student, Customer), the person who is writing our own class, he is responsible to define default natural sorting order by implementing Comparable interface.

The person who is using our class, if he is not satisfied with default natural sorting order, then he can define his own sorting by using Comparator.



```
import java.util.*;
class Employee implements Comparable {
   String name;
   int eid;
   Employee(String name,int eid) {
      this.name = name;
      this.eid = eid;
   }
   public String toString()  {
      return 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;
      }
   }
}
```

```
class CompCompDemo {
    public static void main(String[] args) {
        Employee e1 = new Employee("nag",100);
        Employee e2 = new Employee("balaiah",200);
        Employee e3 = new Employee("chiru",50);
        Employee e4 = new Employee("venki",150);
        Employee e5 = new Employee("nag",100);
        TreeSet t = new TreeSet();
        t.add(e1);
        t.add(e2);
        t.add(e3);
        t.add(e4);
        t.add(e5);
        System.out.println(t);
        TreeSet t1 = new TreeSet(new MyComparator());
        t1.add(e1);
        t1.add(e2);
        t1.add(e3);
        t1.add(e4);
        t1.add(e5);
        System.out.println(t1);
    }
}
```

```
class MyComparator implements Comparator {
   public int compare(Object obj1, Object obj2) {
         Employee e1 = (Employee)obj1;
         Employee e2 = (Employee)obj2;
         String s1 = e1.name;
         String s2 = e2.name;
         return s1.compareTo(s2);
   }
}
```

Comparable	Comparator
1. It is meant for default natural sorting order.	1. It is meant for customized sorting order.
2. Present in java.lang package .	2. Present in java.util package
3. This interface defines only one method compareTo ().	3. This interface defines two methods compare() and equals().
4. All wrapper classes and String class implement comparable interface.	4. The only implemented classes of Comparator are Collator and RuleBasedCollator.

Property	HashSet	LinkedHashSet	TreeSet
1. Underlying Data Structure	Hashtable	Hashtable + LinkedList	Balanced Tree
2. Insertion Order	Not Preserved	Preserved	Not Applicable
3. Sorting order	Not Applicable	Not Applicable	Applicable
4. Heterogeneous objects	Allowed	Allowed	Not Allowed
5. Duplicate Objects	Not Allowed	Not Allowed	Not Allowed
6. Null Acceptance	Allowed (only once)	Allowed(only once)	For Empty TreeSet as first element Null is allowed and in all other cases we will get NullPointerException