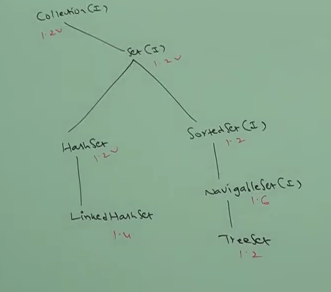
**Set (I):**

It is child interface of Collection.

Duplicates are not allowed

Insertion order not preserved.



Set interface doesn’t contain any new method. We have to use, only Collection interface methods.

**1) Hashset:**

1. The underlying data structure is **hashtable**
2. Duplicates not allowed
3. Insertion not preserved
4. All objects will be inserted based on hashcode only
5. 5)Null insertion is possible.
6. Serializable, Cloneable, but not RandomAccess
7. Best choice if our frequent operation is Search

**Note - > Treemap and TreeSet does not store heterogenous objects but others are good with it.**

Hashset h = new Hashset();

sop(h.add("anbu")); -> return true

sop(h.add("anbu")); -> return false

In hashset duplicates are not allowed. If we are trying to insert duplicates,then we wont get any Exception and add() simply returns false.

* **Hashing based datastructures:**

Hashset, Linkedhashset, hashmap, LinkedHashmap, WeakHashmap, IdentityHashmap. Here all objects are inserted based on hashing algorithm. So the insertion order is not preserved. They are good for search operation.

**Constructors:**

1. HashSet h=new HashSet()

For above, **initial capacity is 16 and default fill ratio or load factor is 0.75**.

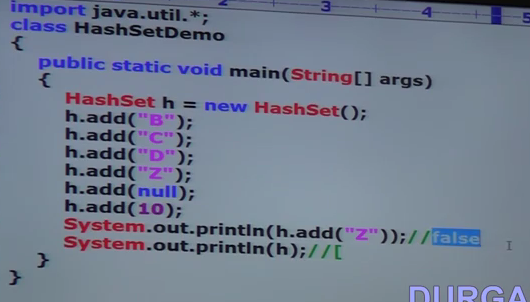
Creates a empty hashset object with default initial capacity 16 and default fill ratio 0.75.

1. HashSet h=new HashSet(int initialcapacity)
2. HashSet h=new HashSet(int initialcapacity, float fillRatio)
3. HashSet h=new HashSet(Collection c)

**Load Factor: Fill Ratio:**

After filling 75 percent of elements, a new hashset object is created for copying.

Example:

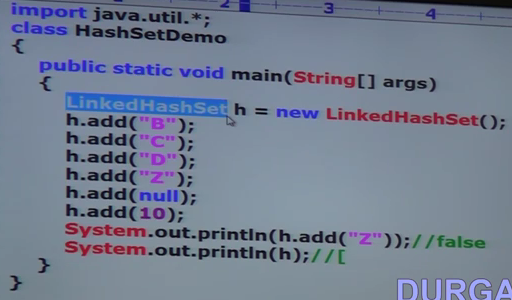


**LinkedHashSet:**

1. It is the child class of HashSet.
2. It is exactly same as Hashset including constructors and methods. Except the following have differences.

|  |  |
| --- | --- |
| **Diff b/w HashSet and LinkedHashSet** | |
| **Hashset** | **LinkedHashSet** |
| Underlying data structure is hashtable | Underlying data structure is combination of LinkedList and Hashtable (Hybrid one) |
| Insertion order not preserved | Insertion order preserved |
| Introduced in 1.2V | Introduced in 1.4V |

Example:

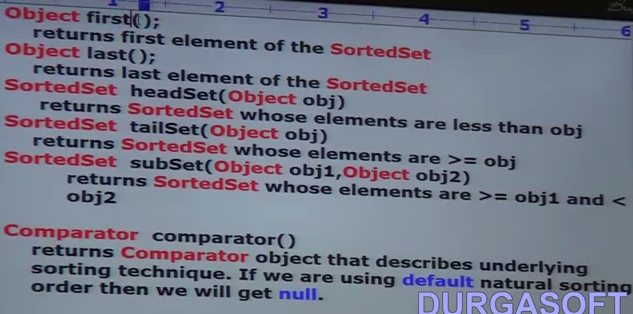


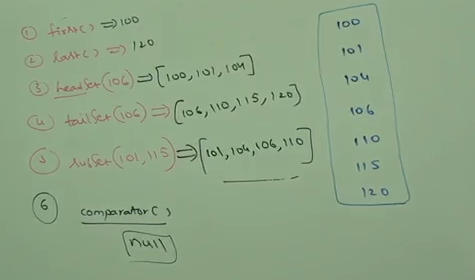
In General, We can use LinkedHashet to develop Cache based applications where duplicates are not allowed and insertion order is preserved.

**SortedSet (I):**

1. It is child interface of Set interface.
2. If we want to represent a group of individual objects according to some sorting order without duplicates, then we go for SortedSet.

**Methods:**





**Default Sorting Order:** For numbers, default sorting is ASC and for String, default sorting order is alphabets.

**NavigableSet (I)**

**TreeSet:**

1. The underlying data structure is **BalancedTree.**
2. Duplicates not allowed
3. Insertion not preserved
4. Heterogeneous objects cannot be stored -> if we try to store, we get Runtime Ex: ClassCast Exception
5. **Allow Null only once -> From 1.7 version. Null is not allowed.**
6. Implements Serializable, Cloneable but not RandomAccess
7. All objects will be inserted based on some sorting order. It may be default natural sorting order or customized sorting order.

**Constructors**:

1. TreeSet t=new TreeSet(); -> a empty treeset is created, where all objects are inserted using natural sorting order
2. TreeSet t=new TreeSet(Comparator c) -> a empty treeset is created, where all objects are inserted using customized sorting order. The customized sorting is defined in Comparator object.
3. TreeSet t =new TreeSet(Collection c)
4. TreeSet t =new TreeSet(SortedSet s) -> based on the sorting order of Sorted object, the treeset object is sorted

**Note: If we are using no argument constructor, then we have to insert homogeneous object only and also those objects should implement Comparable. Comparable provides the default sorting order.**

**The Treeset balancedTree always have “before values” on the left hand side and “after values” on the right hand side. That’s how the tree forms.**

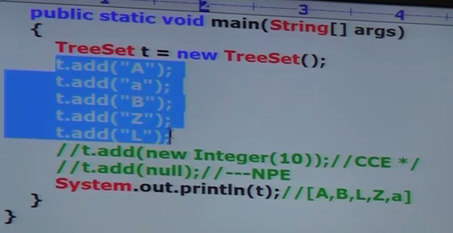
**Null acceptance:**

**t.add(null);**

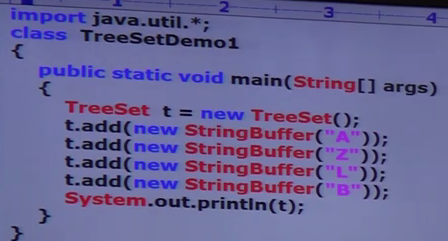
1. For non empty treeset, if we are trying to insert null, then we get NullPointer Exception because null is compared against existing elements of treeset. It doesn’t know how to compare or where to keep before or after the comparing element.
2. If it is empty treeset, we can insert null as a first element, because it doesn’t need to compare null since there is no elements.
3. If null is a first element, after that if we try to insert any element then we get NullPointer Exception because of Null comaprision with newly inserted element.

* **Until 1.6 version, null is allowed as a first allowed to the empty treeset. But from 1.7 onwards, null is not allowed even as a first element. NULL is not applicable for treeset from 1.7 onwards.**

Example 1:



Example 2:



In above example 2 , we get class cast exception eventhough we inserted homogenous objects in default argument treeset constructor. The reason is stringbuffer is not implements comparable that is why the exception. If we change to string , it will work cause strings are comparable that is it implements comparable interface which provides the default sorting mechanism.

Note: If we are depending on default sorting order treeset, compulsory the object should be homogeneous and implements comparable interface otherwise we get ClassCast exception

An object is said to be comparable, if and only if, the corresponding object should implements Comparable interface. The String class and All wrapper classes are already implement comparable interface.

**Comparable (I) :**

1. It is present in java.lang.
2. It contains only one method, compareTo().
3. It is meant for Natural Sorting order

public int comapreTo(Object obj)

**Usage:**

Obj1.compareTo(Obj2) -> **Obj1 is the object which is to be inserted and Obj2 is already inserted.**

Above call returns 3 types of values,

1. Return –1, if obj1 has to come before obj2
2. Return +1, if obj1 has to come after obj2
3. Return 0, if obj1 and obj2 are equal

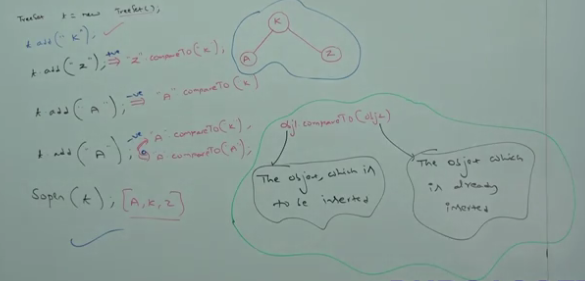
**Example:**

Sop(“A”.compareTo(“Z”)) -> return -1

Sop(“Z”.compareTo(“K”)) -> return +1

Sop(“A”.compareTo(“A”)) -> return 0

Sop(“A”.compareTo(null)) -> NULL POINTER EXCEPTION



If default natural sorting order is not available or if we are not satisfied with default natural sorting, then order, then we go for Customized Sorting by using Comparator interface.

**Comparator (I):**

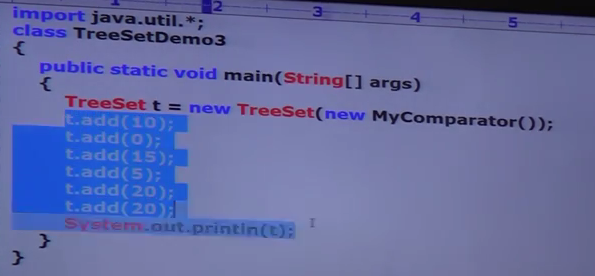
1. It is present in java.util.
2. It contains two methods, compare() and equals().
3. It is meant for Customized Sorting.

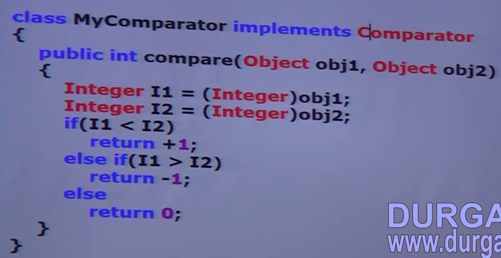
**Methods**:

1. public int compare(Object obj1, Object obj2)
   1. Return –1, if obj1 has to come before obj2
   2. Return +1, if obj1 has to come after obj2
   3. Return 0, if obj1 and obj2 are equal
2. public boolean equals(Object obj)

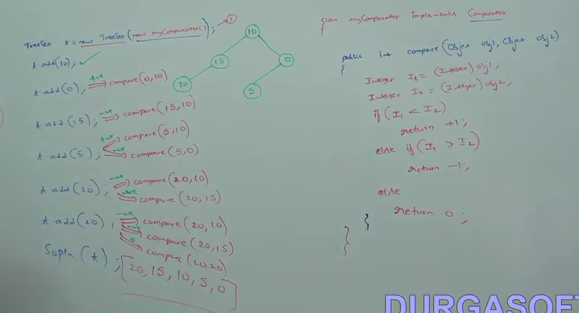
Example:

Write a program to insert integer objects into the treeset where the sorting order is descending order.

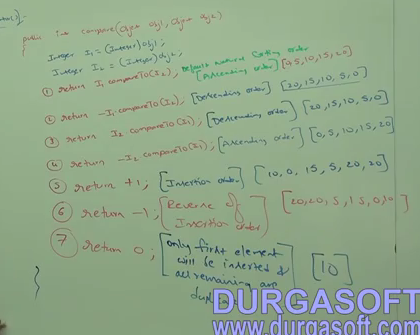
****

****

**How it works:**

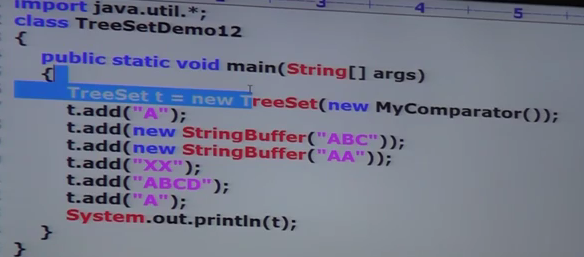
****

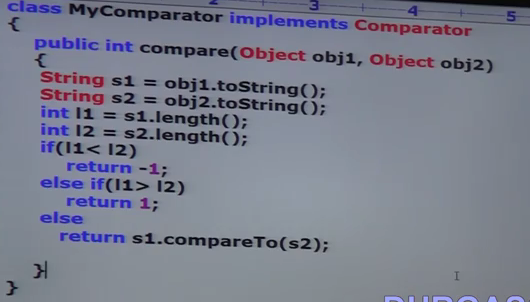
**Various ways implementing Compare():**

****

* If we are depending on default natural sorting order, then compulsory objects should be homogeneous and comparable otherwise we will get runtime time exception saying classcast exception.
* If we are defining our own sorting by comparator, then objects need not comparable and homogeneous. That is we add heterogenous non comparable objects also.

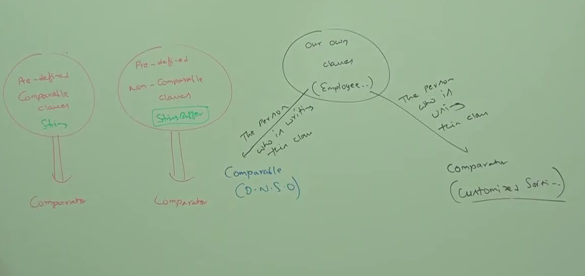
Example:



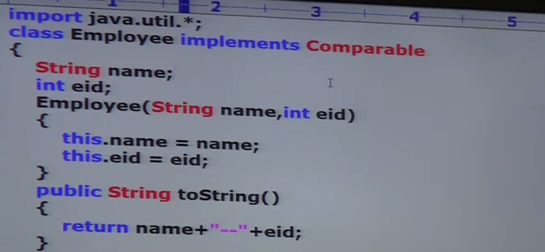


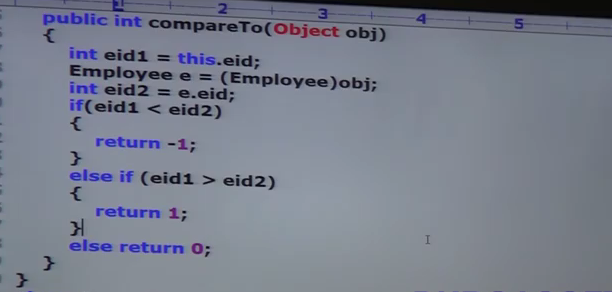
**Comparable VS Comparator: Which and when to use?**

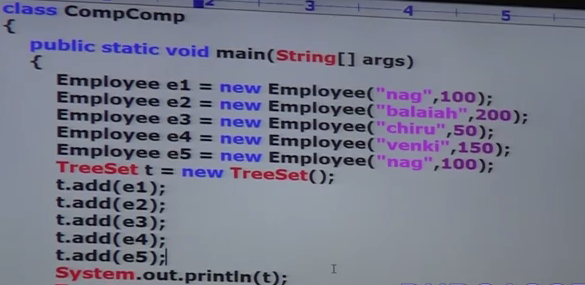
1. For **predefined comparable classes** **(String, Wrapper classes),** default natural sorting order is already available. If we are not satisfied with that default natural sorting order, then we can define our own sorting by using comparator.
2. For **predefined non comparable classes (StringBuffer),** default natural sorting order is not already available. We can define our own sorting by using comparator.
3. For our own classes like Employee, the person who is writing the class is responsible to define default natural sorting order by implementing Comparable interface.
4. The person who is using our Employee class, if it is not satisfied with default natural sorting order, then he can define his own sorting by using comparator.

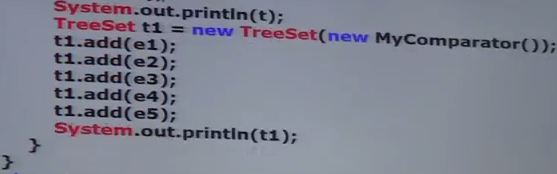


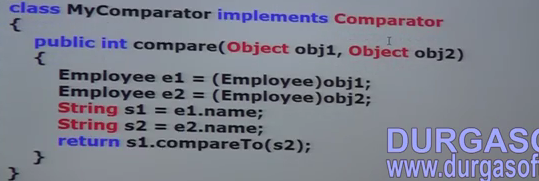
**Example for our own class using both comparable and comparator**:











|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| Meant for Default natural sorting order | Meant for Customized sorting order |
| Java.lang | Java.util |
| Defines Only one method  -> public int compareTo(Object obj) | Defines two methods  ->public int compare(Object obj1, Object obj2)  ->public Boolean equals(Object obj) |
| String & All Wrapper classes implements comparable interface | The only implemented comparator are Collator and RuleBasedCollator which are defined GUI based applications |

