**Set Interface in Java**

The **Set** interface defines a set. It extends Collection and specifies the behavior of a collection that does not allow duplicate elements. Therefore, the add( ) method returns false if an attempt is made to add duplicate elements to a set. It does not specify any additional methods of its own.

Set is a generic interface that has this declaration:

interface Set<E>

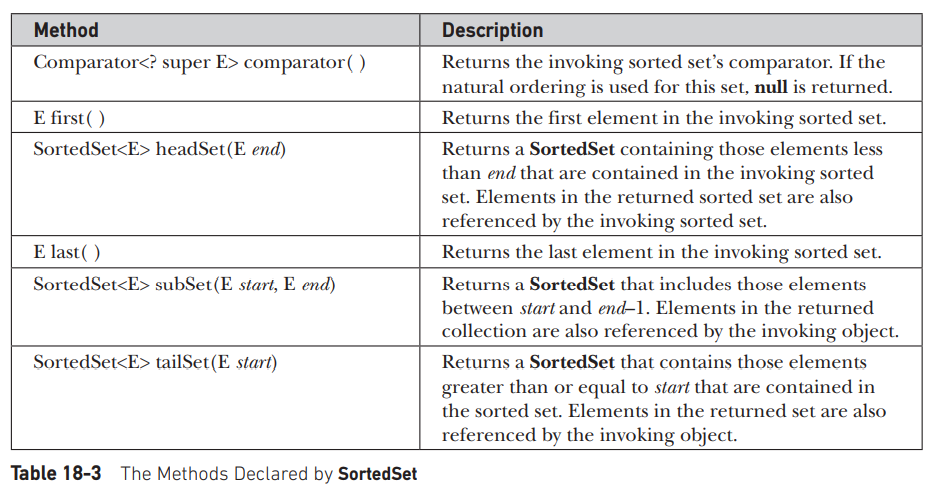
Here, E specifies the type of objects that the set will hold.

**The SortedSet Interface**

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order. SortedSet is a generic interface that has this declaration:

interface SortedSet<E>

Here, E specifies the type of objects that the set will hold. Several methods throw a **NoSuchElementException** when no items are contained in the invoking set. A **ClassCastException** is thrown when an object is incompatible with the elements in a set. A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the set. An **IllegalArgumentException** is thrown if an invalid argument is used. SortedSet defines several methods that make set processing more convenient. To obtain the first object in the set, call first( ). To get the last element, use last( ). You can obtain a subset of a sorted set by calling subSet( ), specifying the first and last object in the set. If you need the subset that starts with the first element in the set, use headSet( ). If you want the subset that ends the set, use tailSet( ).



**The NavigableSet Interface**

The NavigableSet interface extends SortedSet and declares the behavior of a collection

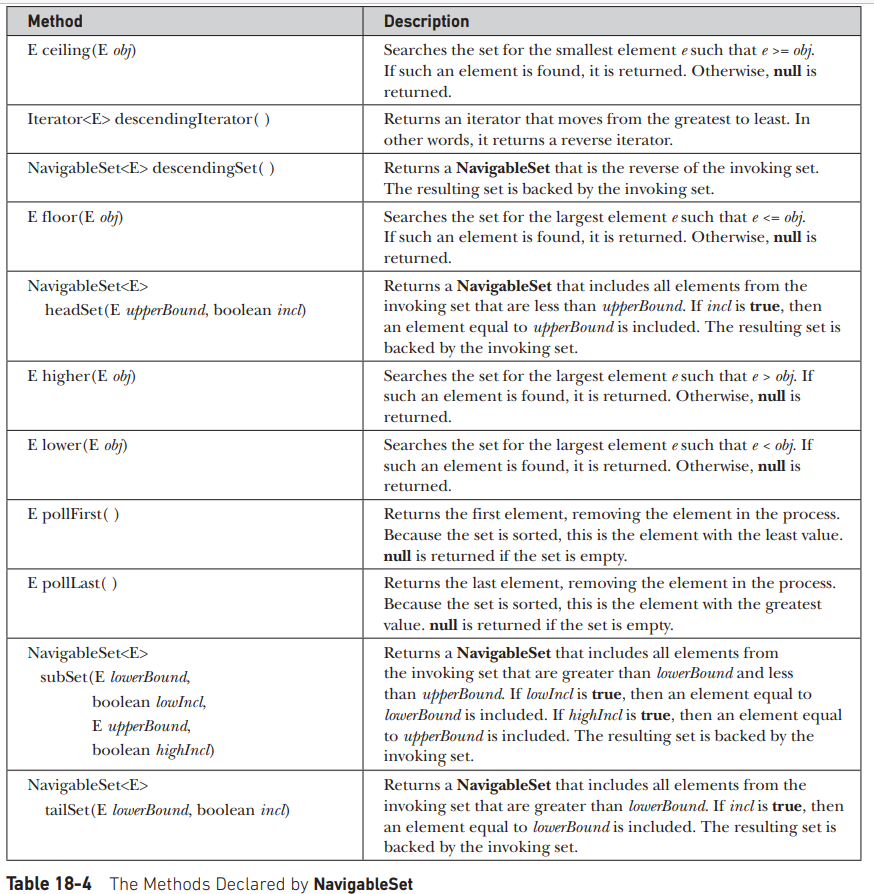
that supports the retrieval of elements based on the closest match to a given value or values.

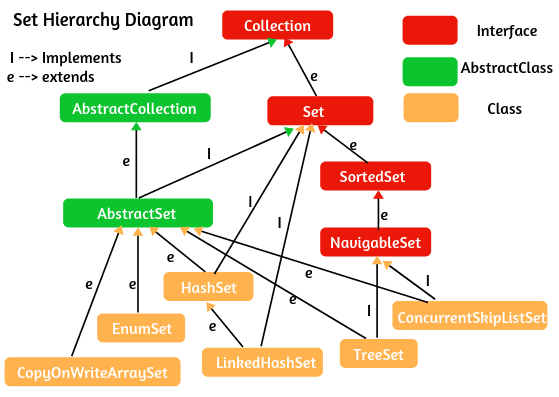
NavigableSet is a generic interface that has this declaration:

interface NavigableSet<E>

Here, E specifies the type of objects that the set will hold. In addition to the methods

that it inherits from SortedSet, NavigableSet adds those summarized in Table 18-4. A





Set is the child interface of collection if we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion is not preserved.

Set interface does not contain any new method and we have to use only collection interface methods.

**HashSet(1.2v):**

Important points about HashSet

1. The underlaying data Structure is Hash table.
2. Duplicate objects are not allowed.
3. Insertion order is not preserved and it is based on HashCode of Objects.
4. Null insertion is possible (Only once).
5. Heterogeneous objects are allowed.
6. Implements Serializable and Cloneable but not Random Access Interface ()
7. Hash set is the best choice if are frequent operation is a search operation.

NOTE: In Hashset duplicates are not allowed if we are trying to insert duplicates then we don’t get any compile time or run time errors, add method simply returns false.

Constructors

1. HashSet h = new HashSet() // Default initial capacity = 16 Default fill Ratio 0.75

An empty HashSet object is created with default initial capacity 16 and filled ration .75.

1. HashSet h = new HashSet(int initialCapacity);

An empty HashSet object is created with specified initial capacity and default filled ratio 0.75.

1. HashSet h = new HashSet(int initialCapacity,float filledRatio);

An empty HashSet object is created with specified initial capacity and specified filled ratio.

1. HashSet h = new HashSet(Collection c); Creates an eqivalent HashSet for the given collection. This constructor is ment for inter conversion between collection Object.

Filled Ratio / Load Factor: After filling how much ratio a new HashSet Object will be Created, this ratio is called fill Ratio. For example, fill ratio 0.75 means after filling 75% ratio a new HashSet Object will be created.

**Linked HashSet**

It is the child class of HashSet. It is exactly same as Hashset including Constructors and methods except the fallowing diffences.

|  |  |
| --- | --- |
| HashSet | Linked HashedSet |
| The under laying data structure is HashTable | Underlying Data structure is the combination of Linked List and HashTable |
| Insertion order not preserved. | Insertion order preserved |
| Introduced in 1.2v | Introduced in 1.4v |

NOTE: In general we can use Linked Hashed set to develop cache-based Applications where duplicates are not allowed and insertion order preserved.

Sorted Set.

Sorted Set is the child Interface of a set. If we want to represent a group of Individual Objects according to some sorting order without duplicates then we should go for sorted set.

The SortedSet interface is present in java.util package extends the Set interface present in the collection framework. It is an interface that implements the mathematical set. This interface contains the methods inherited from the Set interface and adds a feature that stores all the elements in this interface to be stored in a sorted manner.

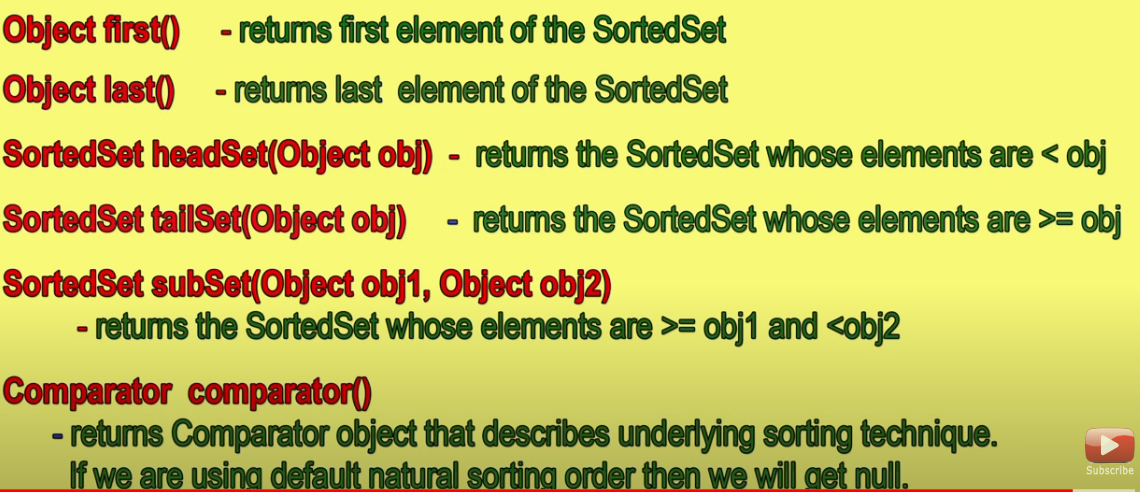


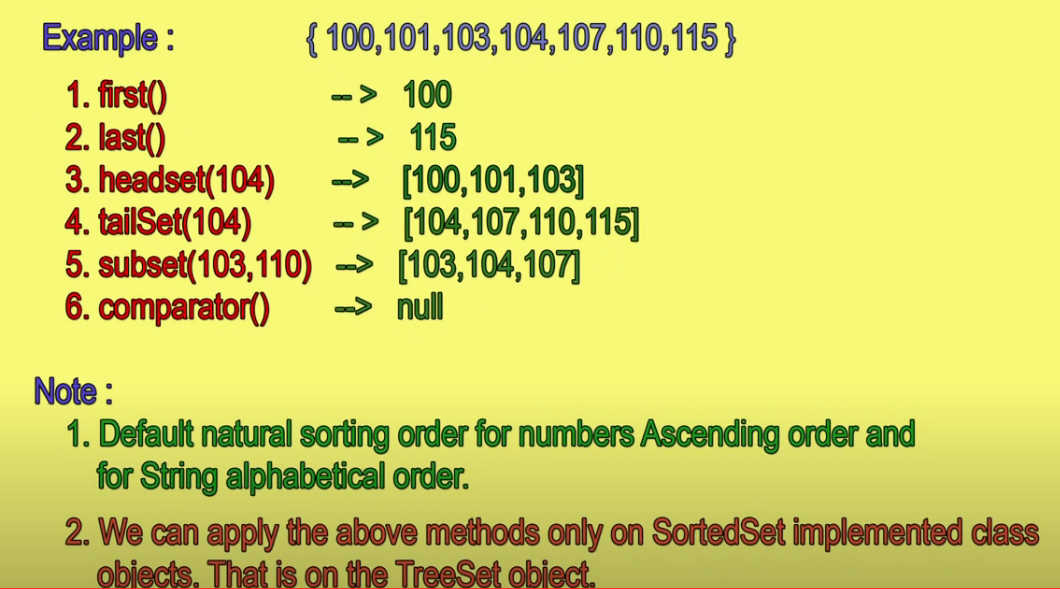
In the above image, the navigable set extends the sorted set interface. Since a set doesn’t retain the insertion order, the navigable set interface provides the implementation to navigate through the Set. The class which implements the navigable set is a TreeSet which is an implementation of a self-balancing tree. Therefore, this interface provides us with a way to navigate through this tree.

**Declaration:** The SortedSet interface is declared as:

*public interface SortedSet extends Set*

**Sorted set specific methods**





**Tree Set:**

Important points about Tree Set

1. The Underlying data structure for TreeSet is Balanced Tree.
2. Duplicate Objects are not allowed.
3. Insertion order not preserved but, all the objects will be inserted in some sorting order.
4. Heterogeneous objects are not allowed. If we are trying to inster heterogeneous objects then we will get 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 the default sorting order. For numbers default sorting is ascending order and for strings Alphabetical order.

1. TreeSet t = new TreeSet(Comparator c);

Creates an empty TreeSet Object where elements will be inserted according to customized sorting order.

1. TreeSet t = new TreeSet(SortedSet s);
2. TreeSet t = new TreeSet(Collection c);

Example

**class** TreeSetDemo {

**public** **static** **void** main(String[] args) {

TreeSet<String> treeSet = **new** TreeSet<>();

treeSet.add("A");

treeSet.add("B");

treeSet.add("F");

treeSet.add("A");

treeSet.add("C");

// treeSet.add(new Integer(12)); // ClassCastException

// treeSet.add(null); //if you try to add null values to a TreeSet it generates a NullPointerException at the run time

}

}

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 other element we will get NullPointerException.
2. For Non-empty TreeSet if we are trying to insert Null then we will get NullPOinterException.

Special cases with TreeSet:

class TreeSetDemo {

public static void main(String[] args) {

TreeSet<StringBuffer> treeSet = new TreeSet<>();

treeSet.add(new StringBuffer("A"));

treeSet.add(new StringBuffer("C"));

treeSet.add(new StringBuffer("R"));

treeSet.add(new StringBuffer("B"));

}

}

Output:

ClassCastException

If we are depending on the default natural sorting order then objects should be homogeneous and comparable. Otherwise, we will get runtime exception ClassCaseException. In the above program in the TreeSet we are adding homogeneous objects StringBuffer type but we will get ClassCastException as StringBuffer is not implementing a comparable interface.

String Class and all the wrapper classes already implement a comparable interface. But StringBuffer doesn’t implement comparable interface.

**Comparable Interface:**

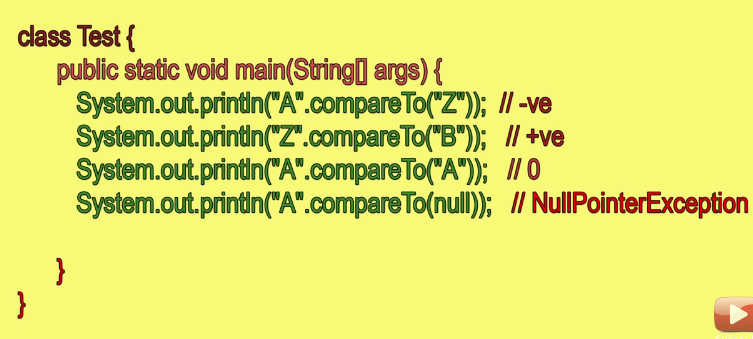
This interface is present in java.lang package it contains only one method ComareTo(Object obj)

Public int ***compareTo***(Object obj)

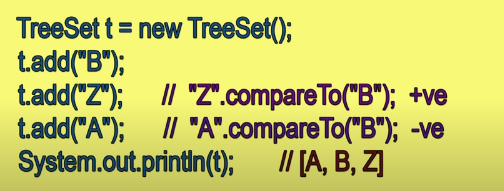
Example:

obj1.compareo(obj2)

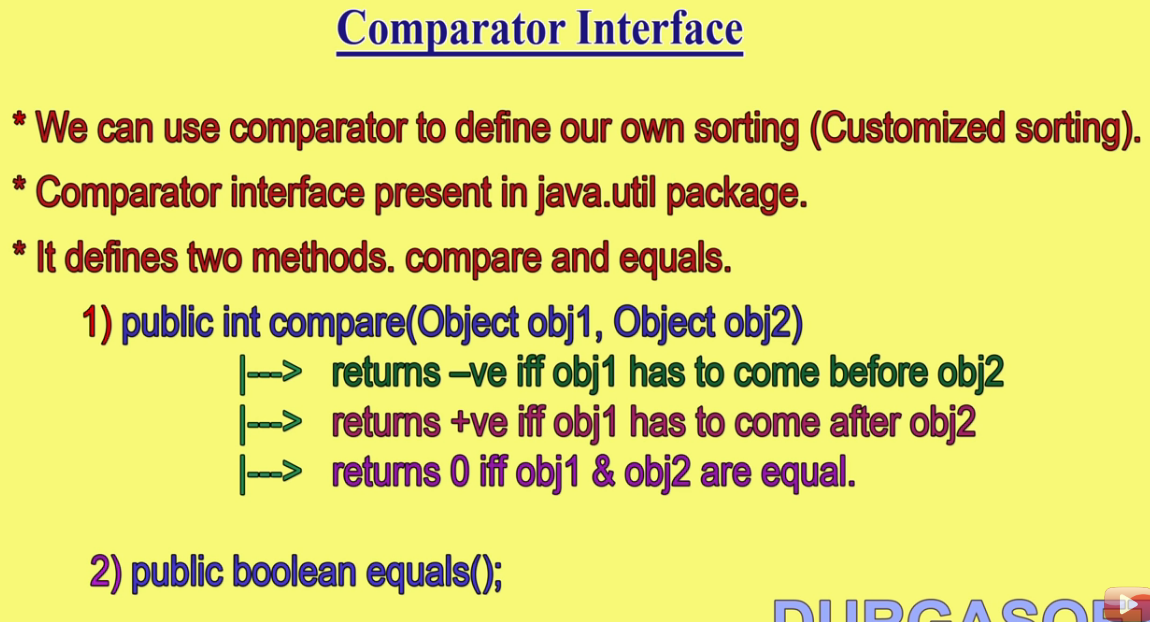
* Returns -ve iff Obj1 has to come before obj2
* Returns +ve iff obj1 has to come after obj2
* Returns zero iff obj1 and obj2 are equal.



Note: If we are depending on the default natural sorting order internally JVM will call ***CompareTo***() method which inserts the objects to the TreeSet. Hence the Objects should be Comparable.



1. If we are not satisfied with the 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** is used for default natural sorting order whereas ***Comparator*** is used for customized sorting order.



class TreeSetDemo {

public static void main(String[] args) {

TreeSet<Integer> treeSet = new TreeSet<>(new MyComparator());

treeSet.add(10);

treeSet.add(5);

treeSet.add(20);

sop(treeSet);

}

}

// If we want descending order sorting with numbers then we need to create a new

// Comparator and pass as arguments to Tree set.

class MyComparator implements Comparator<Integer> {

@Override

public int compare(Integer o1, Integer o2) {

if (o1 < o2)

return 1;

else if (o1 > o2)

return -1;

else

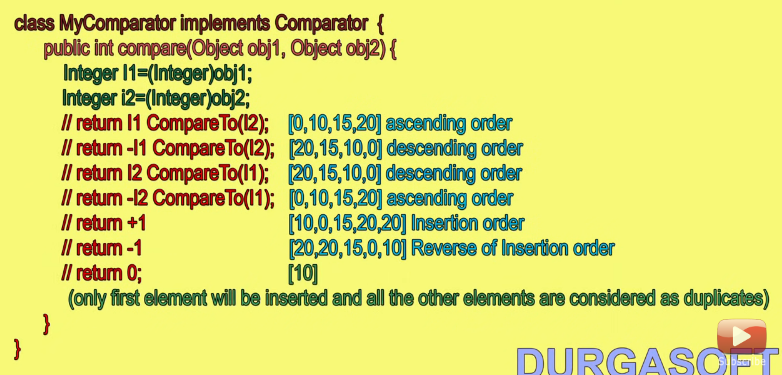
return 0;

}

}

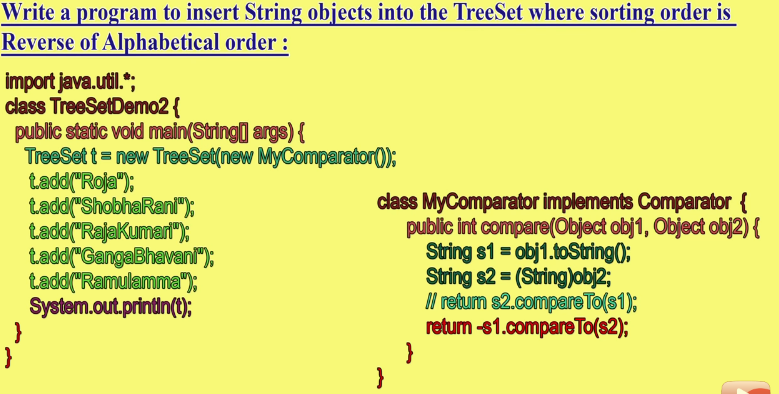
Output

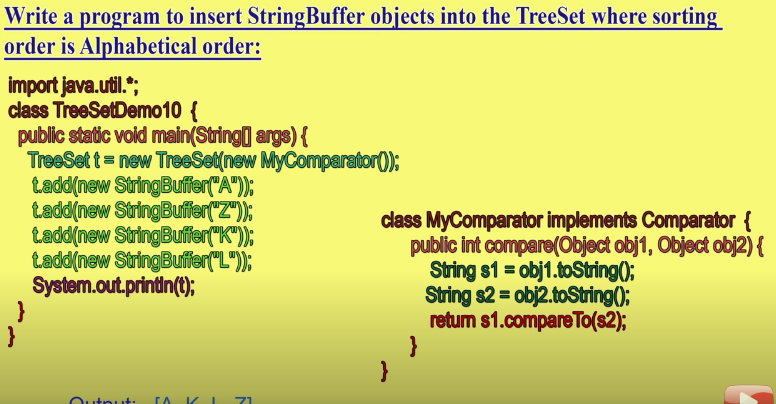
20 10 5



Whenever we are implementing the Comparator interface, compulsory we should provide an implementation for ***compare()*** method.

Implementing the ***equals()*** method is optional because it is already available in every java Class from Object class through inheritance.





If we are adding our own sorting by Comparator, the objects need not be Comparable but they need to be homogeneous.

Difference between Comparable and Comparator

|  |  |
| --- | --- |
| Comparable | Comparator |
| 1. It is meant for default natural sorting order. | 1. It is meant for customized sorting order. |
| 1. Present in java.lang.package | 2. present in java.util.package |
| 1. This method defines only one method compareTo() | 3. This method defines two methods compare() and equals(). |
| 1. All wrapper classes and String classes implements comparable Interface. | 4. The only implemented classes of Comparator are Collactor and RuleBasedCollector. |

