**Set**

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.

***Set (interface)***

* Set is a linear collection of objects.
* Set contains only unique elements, not allowed duplicates.
* Random access of elements not possible.
* Orders of elements in set is implementation dependent. Like **HashSet** are ordered on hashcode of elements, **LinkedHashSet** maintains insertion order and **TreeSet** maintains order according to supplied comparator (if not supplied then ascending order).
* It has stronger contract between equals() and hasCode() methods. User can compare two set instances of different implementation type (HashSet, TreeSet, LinkedHashSet).
* Two set instances irrespective of their implementation types, are said to be equal if they contain same element.

***HashSet (class)***

* HashSet internally uses HashMap to store the object. The keys of that HashMap object will be element of HashSet and their value will be constant.
* Does not allow duplicate elements.
* HashSet maximum have one null element.
* HashSet does not maintain any order and also does not guarantee that order will be remain constant over time.
* Provide constant time complexity performance [ O(1) ] for insertion, removal and retrieval.
* It is not synchronized. If want to then use Collections.synchronizedSet() method.
* **HashSet Example**
* import java.util.HashSet;
* public class HashSetExample {
* public static void main(String args[]) {
* // HashSet declaration
* HashSet<String> hset =
* new HashSet<String>();
* // Adding elements to the HashSet
* hset.add("Apple");
* hset.add("Mango");
* hset.add("Grapes");
* hset.add("Orange");
* hset.add("Fig");
* //Addition of duplicate elements
* hset.add("Apple");
* hset.add("Mango");
* //Addition of null values
* hset.add(null);
* hset.add(null);
* //Displaying HashSet elements
* System.out.println(hset);
* }
* }
* Output:
* [null, Mango, Grapes, Apple, Orange, Fig]
* As you can see there all the duplicate values are not present in the output including the duplicate null value.

## Iterate Set Elements

There are two ways to iterate the elements of a Java Set:

* Using an Iterator obtained from the Set.
* Using the for-each loop.

Both of these options are covered in the following sections.

When iterating the elements in the Set the order of the elements depends on what Set implementation you use, as mentioned earlier.

### Iterate Set Using Iterator

To iterate the elements of a Set using an [**Java Iterator**](http://tutorials.jenkov.com/java-collections/iterator.html), you must first obtain an Iterator from the Set. You obtain an Iterator from a Set by calling the iterator() method. Here is an example of obtaining an Iterator from a Set:

Iterator iterator = set.iterator();

while(iterator.hasNext(){

String element = (String) iterator.next();

}

### Iterate Set Using For-Each Loop

The second way to iterate the elements of a Set is by using a for-each loop. Here is how iterating the elements of a Set using a for-each loop looks:

for(Object object : set) {

String element = (String) object;

}

The Set interface implements the [**Java Iterable**](http://tutorials.jenkov.com/java-collections/iterable.html) interface. That is why you can iterate the elements of a Set using the for-each loop.

### Iterate Set Using the Java Stream API

The third way to iterate a Java Set is via the [**Java Stream API**](http://tutorials.jenkov.com/java-functional-programming/streams.html) . To itereate a Java Set using the Java Stream API you must create a Stream from the Set. Here is an example of creating a Java Stream from a Set and iterate the Stream:

Set set = new HashSet();

set.add("one");

set.add("two");

set.add("three");

Stream stream = set.stream();

stream.forEach((element) -> { System.out.println(element); });

## Remove Elements From Set

You remove elements from a Java Set by calling the remove(Object o) method. Here is an example of removing an element from a Java Set:

set.remove("object-to-remove");

There is no way to remove an object based on index in a Set, since the order of the elements depends on the Set implementation.

## Remove All Elements From Set

You can remove all elements from a Java Set using the clear() method. Here is an example of removing all elements from a Java Set:

set.clear();

**how to convert a HashSet to a List (ArrayList).**

#### Program

Here we have a HashSet of String elements and we are creating an ArrayList of Strings by copying all the elements of HashSet to ArrayList. Following is the complete code:

import java.util.HashSet;

import java.util.List;

import java.util.ArrayList;

class ConvertHashSetToArrayList{

public static void main(String[] args) {

// Create a HashSet

HashSet<String> hset = new HashSet<String>();

//add elements to HashSet

hset.add("Steve");

hset.add("Matt");

hset.add("Govinda");

hset.add("John");

hset.add("Tommy");

// Displaying HashSet elements

System.out.println("HashSet contains: "+ hset);

// Creating a List of HashSet elements

List<String> list = new ArrayList<String>(hset);

// Displaying ArrayList elements

System.out.println("ArrayList contains: "+ list);

}

}

**Output:**

HashSet contains: [Tommy, Matt, Steve, Govinda, John]

ArrayList contains: [Tommy, Matt, Steve, Govinda, John]

**TreeSet**

TreeSet is similar to [**HashSet**](https://beginnersbook.com/2013/12/hashset-class-in-java-with-example/) except that it sorts the elements in the ascending order while HashSet doesn’t maintain any order. TreeSet allows null element but like HashSet it doesn’t allow. Like most of the other collection classes this class is also not synchronized, however it can be synchronized explicitly like this: SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));

***TreeSet (class)***

* The elements in TreeSet are sorted according to specified comparator. It not specified then elements are places according to natural ascending order.
* Elements inserted in TreeSet must be of type comparable and elements must be mutually comparable.
* It does not allow null element.
* It is not synchronized. To synchronize it use Collections.synchronizedSortedSet() method.
* Its time complexity performance is constant O(log n)

## TreeSet Example:

In this example we have two TreeSet (TreeSet<String> & TreeSet<Integer>). We have added the values to both of them randomly however the result we got is sorted in ascending order.

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String args[]) {

// TreeSet of String Type

TreeSet<String> tset = new TreeSet<String>();

// Adding elements to TreeSet<String>

tset.add("ABC");

tset.add("String");

tset.add("Test");

tset.add("Pen");

tset.add("Ink");

tset.add("Jack");

//Displaying TreeSet

System.out.println(tset);

// TreeSet of Integer Type

TreeSet<Integer> tset2 = new TreeSet<Integer>();

// Adding elements to TreeSet<Integer>

tset2.add(88);

tset2.add(7);

tset2.add(101);

tset2.add(0);

tset2.add(3);

tset2.add(222);

System.out.println(tset2);

}

}

Output: You can see both the TreeSet have been sorted in ascending order implicitly.

**Converting HashSet to TreeSet**

Here is the complete code for HashSet to TreeSet conversion. We have a HashSet of Strings and we are creating a TreeSet of strings by copying all the elements of HashSet to TreeSet.

import java.util.HashSet;

import java.util.TreeSet;

import java.util.Set;

class ConvertHashSettoTreeSet{

public static void main(String[] args) {

// Create a HashSet

HashSet<String> hset = new HashSet<String>();

//add elements to HashSet

hset.add("Element1");

hset.add("Element2");

hset.add("Element3");

hset.add("Element4");

// Displaying HashSet elements

System.out.println("HashSet contains: "+ hset);

// Creating a TreeSet of HashSet elements

Set<String> tset = new TreeSet<String>(hset);

// Displaying TreeSet elements

System.out.println("TreeSet contains: ");

for(String temp : tset){

System.out.println(temp);

}

}

}

**Output:**

HashSet contains: [Element1, Element2, Element3, Element4]

TreeSet contains:

Element1

Element2

Element3

Element4

Consider the following example:

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection9{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=**new** TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Ajay

Ravi

Vijay

### Java TreeSet Example 2:

Let's see an example of traversing elements in descending order.

1. **import** java.util.\*;
2. **class** TreeSet2{
3. **public** **static** **void** main(String args[]){
4. TreeSet<String> set=**new** TreeSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ajay");
8. System.out.println("Traversing element through Iterator in descending order");
9. Iterator i=set.descendingIterator();
10. **while**(i.hasNext())
11. {
12. System.out.println(i.next());
13. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection11)

Output:

Traversing element through Iterator in descending order

Vijay

Ravi

Ajay

Traversing element through NavigableSet in descending order

Vijay

Ravi

Ajay

### Java TreeSet Example 3:

Let's see an example to retrieve and remove the highest and lowest Value.

1. **import** java.util.\*;
2. **class** TreeSet3{
3. **public** **static** **void** main(String args[]){
4. TreeSet<Integer> set=**new** TreeSet<Integer>();
5. set.add(24);
6. set.add(66);
7. set.add(12);
8. set.add(15);
9. System.out.println("Highest Value: "+set.pollFirst());
10. System.out.println("Lowest Value: "+set.pollLast());
11. }
12. }

Output:

Highest Value: 12

Lowest Value: 66