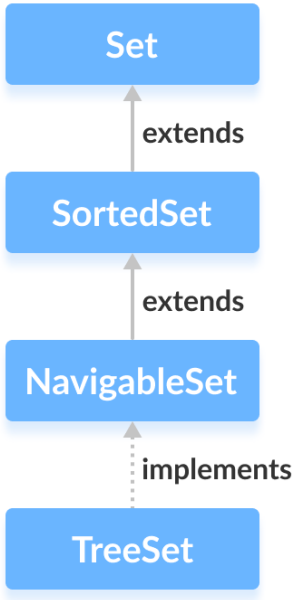
Java TreeSet

In this tutorial, we will learn about the Java TreeSet class and its various operations and methods with the help of examples.

The `TreeSet` class of the Java collections framework provides the functionality of a tree data structure.

It extends the NavigableSet interface.



Creating a TreeSet

In order to create a tree set, we must import the `java.util.TreeSet` package first.

Once we import the package, here is how we can create a `TreeSet` in Java.

```
TreeSet<Integer> numbers = new TreeSet<>();
```

Here, we have created a `TreeSet` without any arguments. In this case, the elements in `TreeSet` are sorted naturally (ascending order).

However, we can customize the sorting of elements by using the `Comparator` interface. We will learn about it later in this tutorial.

Methods of TreeSet

The `TreeSet` class provides various methods that allow us to perform various operations on the set.

Insert Elements to TreeSet

- `add()` - inserts the specified element to the set
- `addAll()` - inserts all the elements of the specified collection to the set

For example,

```java
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {

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

        // Using the add() method
        evenNumbers.add(2);
        evenNumbers.add(4);
        evenNumbers.add(6);
        System.out.println("TreeSet: " + evenNumbers);

        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);

        // Using the addAll() method
        numbers.addAll(evenNumbers);
        System.out.println("New TreeSet: " + numbers);
    }
}
```

Output

```
TreeSet: [2, 4, 6]
New TreeSet: [1, 2, 4, 6]
```

Access TreeSet Elements

To access the elements of a tree set, we can use the `iterator()` method. In order to use this method, we must import `java.util.Iterator` package. For example,

```
import java.util.TreeSet;
import java.util.Iterator;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Calling iterator() method
        Iterator<Integer> iterate = numbers.iterator();
        System.out.print("TreeSet using Iterator: ");
        // Accessing elements
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

Output

```
TreeSet: [2, 5, 6]
TreeSet using Iterator: 2, 5, 6,
```

Remove Elements

- `remove()` - removes the specified element from the set
- `removeAll()` - removes all the elements from the set

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using the remove() method
        boolean value1 = numbers.remove(5);
        System.out.println("Is 5 removed? " + value1);

        // Using the removeAll() method
        boolean value2 = numbers.removeAll(numbers);
        System.out.println("Are all elements removed? " + value2);
    }
}
```

Output

```
TreeSet: [2, 5, 6]
Is 5 removed? true
Are all elements removed? true
```

Methods for Navigation

Since the `TreeSet` class implements `NavigableSet`, it provides various methods to navigate over the elements of the tree set.

1. first() and last() Methods

- `first()` - returns the first element of the set
- `last()` - returns the last element of the set

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using the first() method
        int first = numbers.first();
        System.out.println("First Number: " + first);

        // Using the last() method
        int last = numbers.last();
        System.out.println("Last Number: " + last);
    }
}
```

Output

```
TreeSet: [2, 5, 6]
First Number: 2
Last Number: 6
```

---

2. ceiling(), floor(), higher() and lower() Methods

- **higher(element)** - Returns the lowest element among those elements that are greater than the specified `element`.
- **lower(element)** - Returns the greatest element among those elements that are less than the specified `element`.
- **ceiling(element)** - Returns the lowest element among those elements that are greater than the specified `element`. If the `element` passed exists in a tree set, it returns the `element` passed as an argument.
- **floor(element)** - Returns the greatest element among those elements that are less than the specified `element`. If the `element` passed exists in a tree set, it returns the `element` passed as an argument.

For example,

```java
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using higher()
        System.out.println("Using higher: " + numbers.higher(4));

        // Using lower()
        System.out.println("Using lower: " + numbers.lower(4));

        // Using ceiling()
        System.out.println("Using ceiling: " + numbers.ceiling(4));

        // Using floor()
        System.out.println("Using floor: " + numbers.floor(3));

    }
}
```

Output

```
TreeSet: [2, 4, 5, 6]
Using higher: 5
Using lower: 2
Using ceiling: 4
Using floor: 2
```

---

3. pollfirst() and pollLast() Methods

- `pollFirst()` - returns and removes the first element from the set
- `pollLast()` - returns and removes the last element from the set

For example,

```java
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using pollFirst()
        System.out.println("Removed First Element: " + numbers.pollFirst());

        // Using pollLast()
        System.out.println("Removed Last Element: " + numbers.pollLast());

        System.out.println("New TreeSet: " + numbers);
    }
}
```

Output

```
TreeSet: [2, 4, 5, 6]
Removed First Element: 2
Removed Last Element: 6
New TreeSet: [4, 5]
```

4. headSet(), tailSet() and subSet() Methods

---

headSet(element, booleanValue)

The `headSet()` method returns all the elements of a tree set before the specified `element` (which is passed as an argument).

The `booleanValue` parameter is optional. Its default value is `false`.

If `true` is passed as a `booleanValue`, the method returns all the elements before the specified element including the specified element.

For example,

```java
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using headSet() with default boolean value
        System.out.println("Using headSet without boolean value: " + numbers.headSet(5));

        // Using headSet() with specified boolean value
        System.out.println("Using headSet with boolean value: " + numbers.headSet(5, true));
    }
}
```

Output

```
TreeSet: [2, 4, 5, 6]
Using headSet without boolean value: [2, 4]
Using headSet with boolean value: [2, 4, 5]
```

---

tailSet(element, booleanValue)

The `tailSet()` method returns all the elements of a tree set after the specified `element` (which is passed as a parameter) including the specified `element`.

The `booleanValue` parameter is optional. Its default value is `true`.

If `false` is passed as a `booleanValue`, the method returns all the elements after the specified `element` without including the specified `element`.

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using tailSet() with default boolean value
        System.out.println("Using tailSet without boolean value: " + numbers.tailSet(4));

        // Using tailSet() with specified boolean value
        System.out.println("Using tailSet with boolean value: " + numbers.tailSet(4, false));
    }
}
```

Output

```
TreeSet: [2, 4, 5, 6]
Using tailSet without boolean value: [4, 5, 6]
Using tailSet with boolean value: [5, 6]
```

---

subSet(e1, bv1, e2, bv2)

The `subSet()` method returns all the elements between `e1` and `e2` including `e1`.

The `bv1` and `bv2` are optional parameters. The default value of `bv1` is `true`, and the default value of `bv2` is `false`.

If `false` is passed as `bv1`, the method returns all the elements between `e1` and `e2` without including `e1`.

If `true` is passed as `bv2`, the method returns all the elements between `e1` and `e2`, including `e1`.

For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(4);
        numbers.add(6);
        System.out.println("TreeSet: " + numbers);

        // Using subSet() with default boolean value
        System.out.println("Using subSet without boolean value: " + numbers.subSet(4, 6));

        // Using subSet() with specified boolean value
        System.out.println("Using subSet with boolean value: " + numbers.subSet(4, false, 6, true));
    }
}
```

Output

```
TreeSet: [2, 4, 5, 6]
Using subSet without boolean value: [4, 5]
Using subSet with boolean value: [5, 6]
```

Set Operations

The methods of the `TreeSet` class can also be used to perform various set operations.

Union of Sets

To perform the union between two sets, we use the `addAll()` method. For example,

```
import java.util.TreeSet;;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> evenNumbers = new TreeSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("TreeSet1: " + evenNumbers);

        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("TreeSet2: " + numbers);

        // Union of two sets
        numbers.addAll(evenNumbers);
        System.out.println("Union is: " + numbers);

    }
}
```

Output

```
TreeSet1: [2, 4]
TreeSet2: [1, 2, 3]
Union is: [1, 2, 3, 4]
```

Intersection of Sets

To perform the intersection between two sets, we use the `retainAll()` method. For example,

```
import java.util.TreeSet;;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> evenNumbers = new TreeSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("TreeSet1: " + evenNumbers);

        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("TreeSet2: " + numbers);

        // Intersection of two sets
        numbers.retainAll(evenNumbers);
        System.out.println("Intersection is: " + numbers);
    }
}
```

Output

```
TreeSet1: [2, 4]
TreeSet2: [1, 2, 3]
Intersection is: [2]
```

Difference of Sets

To calculate the difference between the two sets, we can use the `removeAll()` method. For example,

```
import java.util.TreeSet;;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> evenNumbers = new TreeSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("TreeSet1: " + evenNumbers);

        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        System.out.println("TreeSet2: " + numbers);

        // Difference between two sets
        numbers.removeAll(evenNumbers);
        System.out.println("Difference is: " + numbers);
    }
}
```

Output

```
TreeSet1: [2, 4]
TreeSet2: [1, 2, 3, 4]
Difference is: [1, 3]
```

Subset of a Set

To check if a set is a subset of another set or not, we use the `containsAll()` method. For example,

```
import java.util.TreeSet;

class Main {
    public static void main(String[] args) {
        TreeSet<Integer> numbers = new TreeSet<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        System.out.println("TreeSet1: " + numbers);

        TreeSet<Integer> primeNumbers = new TreeSet<>();
        primeNumbers.add(2);
        primeNumbers.add(3);
        System.out.println("TreeSet2: " + primeNumbers);

        // Check if primeNumbers is subset of numbers
        boolean result = numbers.containsAll(primeNumbers);
        System.out.println("Is TreeSet2 subset of TreeSet1? " + result);
    }
}
```

Output

```
TreeSet1: [1, 2, 3, 4]
TreeSet2: [2, 3]
Is TreeSet2 subset of TreeSet1? True
```

Other Methods of TreeSet

Method	Description
`clone()`	Creates a copy of the TreeSet
`contains()`	Searches the TreeSet for the specified element and returns a boolean result
`isEmpty()`	Checks if the `TreeSet` is empty
`size()`	Returns the size of the `TreeSet`
`clear()`	Removes all the elements from the `TreeSet`

To learn more, visit Java TreeSet (official Java documentation).

TreeSet Vs. HashSet

Both the `TreeSet` as well as the `HashSet` implements the `Set` interface. However, there exist some differences between them.

- Unlike `HashSet`, elements in `TreeSet` are stored in some order. It is because `TreeSet` implements the `SortedSet` interface as well.
- `TreeSet` provides some methods for easy navigation. For example, `first()`, `last()`, `headSet(`), `tailSet()`, etc. It is because `TreeSet` also implements the `NavigableSet` interface.
- `HashSet` is faster than the `TreeSet` for basic operations like add, remove, contains and size.

TreeSet Comparator

In all the examples above, tree set elements are sorted naturally. However, we can also customize the ordering of elements.

For this, we need to create our own comparator class based on which elements in a tree set are sorted. For example,

```
import java.util.TreeSet;
import java.util.Comparator;

class Main {
    public static void main(String[] args) {

        // Creating a tree set with customized comparator
        TreeSet<String> animals = new TreeSet<>(new CustomComparator());

        animals.add("Dog");
        animals.add("Zebra");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("TreeSet: " + animals);
    }

    // Creating a comparator class
    public static class CustomComparator implements Comparator<String> {

        @Override
        public int compare(String animal1, String animal2) {
            int value =  animal1.compareTo(animal2);

            // elements are sorted in reverse order
            if (value > 0) {
                return -1;
            }
            else if (value < 0) {
                return 1;
```

Output

```
TreeSet: [Zebra, Horse, Dog, Cat]
```

In the above example, we have created a tree set passing `CustomComparator` class as an argument.

The `CustomComparator` class implements the `Comparator` interface.

We then override the `compare()` method. The method will now sort elements in reverse order.