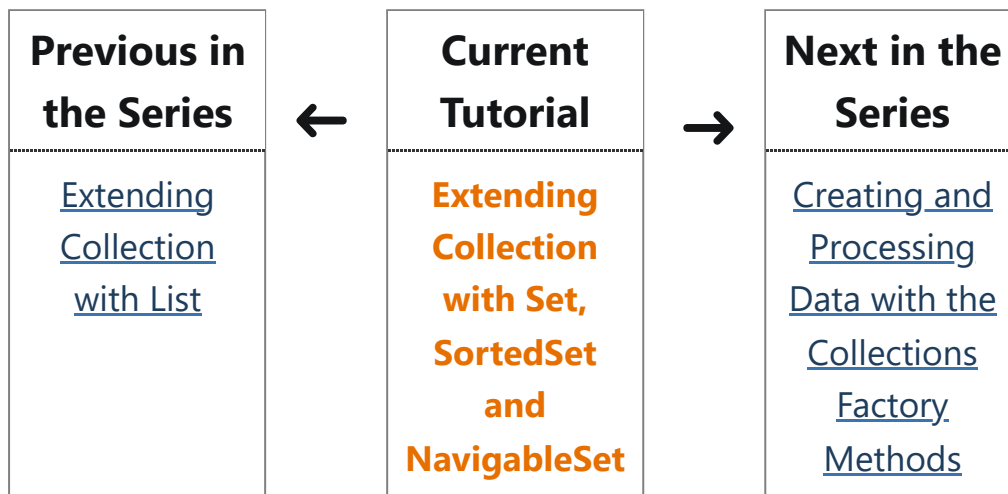


[Tutorials](#)[Watch & Listen](#)[FAQ](#)[Oracle University](#)

[Home](#) > [Tutorials](#) > [The Collections Framework](#) > Extending Collection with Set, SortedSet and NavigableSet



Extending Collection with Set, SortedSet and NavigableSet

Exploring the Set Interface

The [Set](#) interface does not bring any new method to the [Collection](#) interface. The Collections Framework gives you one plain implementation of the [Set](#) interface: [HashSet](#). Internally, a [HashSet](#) wraps an instance of [HashMap](#), a class that will be covered later, that acts as a delegate for [HashSet](#).

In this tutorial

Exploring the Set Interface

Extending Set with SortedSet

Extending SortedSet with


NavigableSet

As you already saw, what a [Set](#) brings to a [Collection](#) is that it forbids duplicates.

What you lose over the [List](#) interface is that your elements are stored in no particular order. There is very little chance that you will iterate over them in the same order as you added them to your set.

You can see this in the following example:

```
1 | List<String> strings = List.of("one",  
2 | Set<String> set = new HashSet<>();  
3 | set.addAll(strings);  
4 | set.forEach(System.out::println);
```



Running this code will produce the following:

```
1 | six  
2 | four  
3 | one  
4 | two  
5 | three  
6 | five
```

Some implementations of [Set](#) give you the same order when you iterate over their elements, but since this is not guaranteed, your code should not rely on that.

Extending Set with SortedSet

The first extension of [Set](#) is the [SortedSet](#) interface. The [SortedSet](#) interface keeps its

elements sorted according to a certain comparison logic. The Collections Framework gives you one implementation of [SortedSet](#), called [TreeSet](#).

As you already saw, either you provide a comparator when you build a [TreeSet](#), or you implement the [Comparable](#) interface for the elements you put in the [TreeSet](#). If you do both, then the comparator takes precedence.

The [SortedSet](#) interface adds new methods to [Set](#).

- [first\(\)](#) and [last\(\)](#) returns the lowest and the largest elements of the set
- [headSet\(toElement\)](#) and [tailSet\(fromElement\)](#) returns you subsets containing the elements lower than `toElement` or greater than `fromElement`
- [subSet\(fromElement, toElement\)](#) gives you a subset of the element between `fromElement` and `toElement`.

The `toElement` and `fromElement` do not have to be elements of the main set. If they are, then `toElement` is not included in the result and `fromElement` is, following the usual convention.

Consider the following example:

```
SortedSet<String> strings = new TreeSet<String>();  
SortedSet<String> subSet = strings.subSet("a", "c");
```

```
System.out.println("sub set = " + subSet);
```

Running this code will give you the following:

```
1 | sub set = [b, c]
```

The three subsets that these methods return are *views* on the main set. No copy is made, meaning that any change you make to these subsets will be reflected in the set, and the other way round.

You can remove or add elements to the main set through these subsets. There is one point you need to keep in mind though. These three subsets remember the limits on which they have been built. For consistency reasons, it is not legal to add an element through a subset outside its limits. For instance, if you take a [headSet](#) and try to add an element greater than [toElement](#), then you will get an [IllegalArgumentException](#).

Extending SortedSet with NavigableSet

Java SE 6 saw the introduction of an extension of [SortedSet](#) with the addition of more methods. It turns out that the [TreeSet](#) class was retrofitted to implement

[NavigableSet](#). So you can use the same class for both interfaces.

Some methods are overloaded by [NavigableSet](#).

- [headSet\(\)](#), [tailSet\(\)](#), and [subSet\(\)](#) may take a further `boolean` arguments to specify whether the limits (`toElement` or `fromElement`) are to be included in the resulting subset.

Other methods have been added.

- [ceiling\(element\)](#), and [floor\(element\)](#) return the greatest element lesser or equal than, or the lowest element greater or equal than the provided `element`. If there is no such element then `null` is returned
- [lower\(element\)](#), and [higher\(element\)](#) return the greater element lesser than, or the lowest element greater than the provided `element`. If there is no such element then `null` is returned.
- [pollFirst\(\)](#), and [pollLast\(\)](#) return and removes the lowest or the greatest element of the set.

Furthermore, [NavigableSet](#) also allows you to iterate over its elements in descending order. There are two ways to do this.

- You can call [descendingIterator\(\)](#): it gives you a regular [Iterator](#) that

traverses the set in the descending order.

- You can also call [descendingSet\(\)](#). What you get in return is another [NavigableSet](#) that is a view on this set and that makes you think you have the same set, sorted in the reversed order.

The following example demonstrates this.

```
1 | NavigableSet<String> sortedStrings =  
2 | System.out.println("sorted strings =  
3 | NavigableSet<String> reversedStrings  
4 | System.out.println("reversed strings
```

Running this code will give you the following:

```
1 | sorted strings = [a, b, c, d, e, f]  
2 | reversed strings = [f, e, d, c, b, a]
```

Last update: September 14, 2021

