# Renewed Comparator: Everything You Should Know

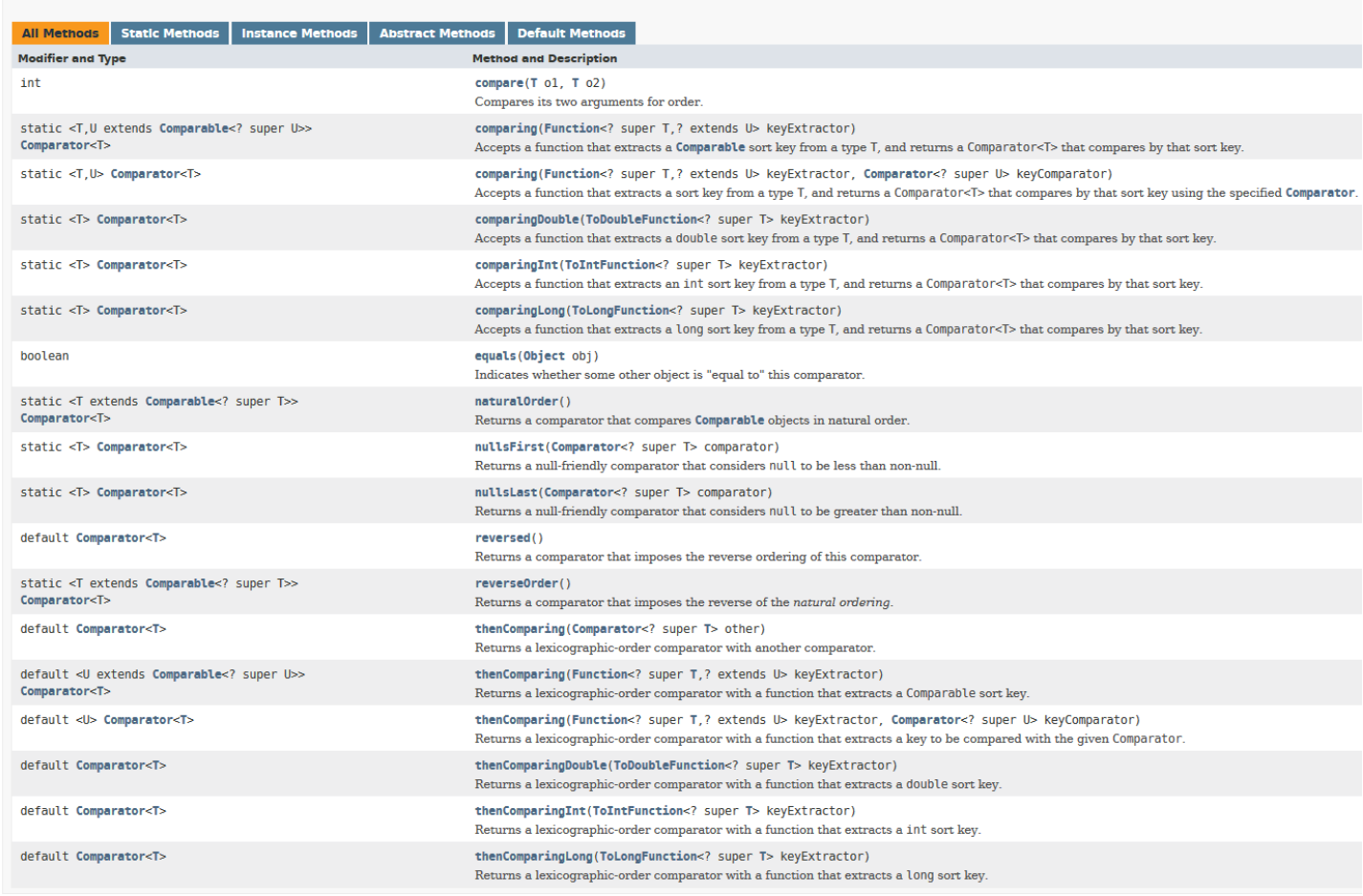
This lesson teaches you how to use the power of the newly added methods in the Comparator interface to write concise and efficient Comparators.

**We'll cover the following**



## Introduction to the `Comparator` interface #

`Comparator` is an interface that is used to define how a collection must be sorted. In Java 7, it had just 2 methods – `compare()` and `equals()`. The enhanced `Comparator` in Java 8 now has 19 methods. However, the `Comparator` is still a functional interface as it has only one abstract method, i.e., `compare()`. `Comparator` now supports declarations via lambda expressions as it is a Functional Interface. We saw this in our lambda expressions chapter as well. Below is the list of methods in the `Comparator` interface.



All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

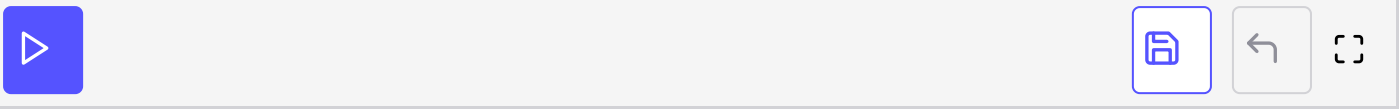
| Modifier and Type | Method and Description |
|---|---|
| int | **compare**(T o1, T o2)<br>Compares its two arguments for order. |
| static <T,U extends Comparable<? super U>> Comparator<T> | **comparing**(Function<? super T,? extends U> keyExtractor)<br>Accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator<T> that compares by that sort key. |
| static <T,U> Comparator<T> | **comparing**(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)<br>Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified Comparator. |
| static <T> Comparator<T> | **comparingDouble**(ToDoubleFunction<? super T> keyExtractor)<br>Accepts a function that extracts a double sort key from a type T, and returns a Comparator<T> that compares by that sort key. |
| static <T> Comparator<T> | **comparingInt**(ToIntFunction<? super T> keyExtractor)<br>Accepts a function that extracts an int sort key from a type T, and returns a Comparator<T> that compares by that sort key. |
| static <T> Comparator<T> | **comparingLong**(ToLongFunction<? super T> keyExtractor)<br>Accepts a function that extracts a long sort key from a type T, and returns a Comparator<T> that compares by that sort key. |
| boolean | **equals**(Object obj)<br>Indicates whether some other object is "equal to" this comparator. |
| static <T extends Comparable<? super T>> Comparator<T> | **naturalOrder**()<br>Returns a comparator that compares Comparable objects in natural order. |
| static <T> Comparator<T> | **nullsFirst**(Comparator<? super T> comparator)<br>Returns a null-friendly comparator that considers null to be less than non-null. |
| static <T> Comparator<T> | **nullsLast**(Comparator<? super T> comparator)<br>Returns a null-friendly comparator that considers null to be greater than non-null. |
| default Comparator<T> | **reversed**()<br>Returns a comparator that imposes the reverse ordering of this comparator. |
| static <T extends Comparable<? super T>> Comparator<T> | **reverseOrder**()<br>Returns a comparator that imposes the reverse of the *natural ordering*. |
| default Comparator<T> | **thenComparing**(Comparator<? super T> other)<br>Returns a lexicographic-order comparator with another comparator. |
| default <U extends Comparable<? super U>> Comparator<T> | **thenComparing**(Function<? super T,? extends U> keyExtractor)<br>Returns a lexicographic-order comparator with a function that extracts a Comparable sort key. |
| default <U> Comparator<T> | **thenComparing**(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)<br>Returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator. |
| default Comparator<T> | **thenComparingDouble**(ToDoubleFunction<? super T> keyExtractor)<br>Returns a lexicographic-order comparator with a function that extracts a double sort key. |
| default Comparator<T> | **thenComparingInt**(ToIntFunction<? super T> keyExtractor)<br>Returns a lexicographic-order comparator with a function that extracts a int sort key. |
| default Comparator<T> | **thenComparingLong**(ToLongFunction<? super T> keyExtractor)<br>Returns a lexicographic-order comparator with a function that extracts a long sort key. |

We will look at how these new methods work and discuss the functionalities they provide. Before Java 8, there was only one way to use the `Comparator` interface. We would create an implementation of the `Comparator<T>` interface, override the `compare()` method of the interface with the desired comparison logic and use `Collections.sort()`, or a similar method in `Collections` API, to sort the collection of objects.

Below is an example of sorting a `List` of objects **before** Java 8.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class ComparatorDemo {

    public static void main(String args[]){
        List<Person> personList = new ArrayList<>();
        personList.add(new Person("Jane",54));
        personList.add(new Person("Dave",21));
        personList.add(new Person("Carl",34));

        // Here we are using an anonymous comparator to sort the List.
        Collections.sort(personList, new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
```

```
            public int compare(Person o1, Person o2) {
                return o1.getName().compareTo(o2.getName());
            }
        });

        personList.forEach(System.out::println);
    }
}

class Person {
    String name;
    int age;
    int yearsOfService;

    Person(String name, int age){
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

}
```



Instead of using an anonymous class, we can use a lambda expression in Java 8. We already saw this in the lambdas chapter so I will not repeat it again.

## New methods in the `Comparator` interface #

This lesson primarily focuses on the new methods that have been added to the `Comparator` interface.

### a) `comparing()` #

The `comparing()` is a static method introduced in Java 8. It takes a `Function<T, R>` functional interface instance as an input and returns a `Comparator` instance.

Let's see how the above program can be written using the `comparing()` method.

The basic idea here is that we don't need to write the entire logic of comparing ourselves. We just need to specify which fields from the object should be used for comparison.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class ComparatorDemo {

    public static void main(String args[]){
        List<Person> personList = new ArrayList<>();
        personList.add(new Person("Jane", 54));
        personList.add(new Person("Dave", 21));
        personList.add(new Person("Carl", 34));
        // Sorting the List using comparing() method of Comparator.
        Collections.sort(personList, Comparator.comparing(Person::getName));

        personList.forEach(System.out::println);
    }
}

class Person {
    String name;
    int age;
    int yearsOfService;

    Person(String name, int age){
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

## b) `thenComparing()` #

What if we need to sort the person object based on the basis of name? If the name is the same then we will need to sort on the basis of age.

is the same then we will need to sort on the basis of age.

In this scenario, we will use `thenComparing()` method. It is a default method that takes in a function and returns a `Comparator`.

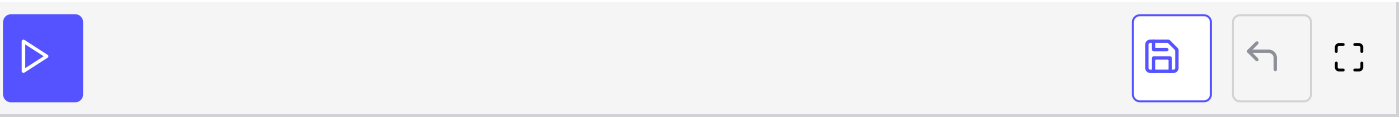
Since this is a non-static method, it cannot be called directly from the `Comparator`. It can be called from the `Comparator` object. The below code will not compile.

```
Collections.sort(personList, Comparator.thenComparing(Person::getAge));
```

Below is the example to sort the person list by name and age.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class ComparatorDemo {

    public static void main(String args[]){
        List<Person> personList = new ArrayList<>();
        personList.add(new Person("Jane",54));
        personList.add(new Person("Dave",21));
        personList.add(new Person("Carl",34));
        personList.add(new Person("Dave",58));
        personList.add(new Person("Carl",12));

        Using thenComparing() method to sort the List on the basis of two criterias.
        Collections.sort(personList, Comparator.comparing(Person::getName).thenComparing(Person::g

        personList.forEach(System.out::println);
    }
}

class Person {
    String name;
    int age;
    int yearsOfService;

    Person(String name, int age){
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person{" +
```

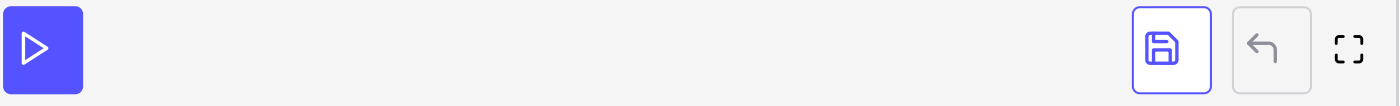
```
                "name='" + name + '\'' +
                ", age=" + age +
                '}';

    }

}
```



## c) `naturalOrder()`

If we don't need to provide our own implementation of the `Comparator` and use the natural order, we can use the `naturalOrder()` method. This is a static method that returns a comparator, which sorts in the natural order.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class ComparatorDemo {

    public static void main(String args[]){
        List<String> fruits = new ArrayList<>();
        fruits.add("guava");
        fruits.add("peach");
        fruits.add("apple");
        fruits.add("banana");

        Collections.sort(fruits, Comparator.naturalOrder());

        fruits.forEach(System.out::println);
    }
}
```

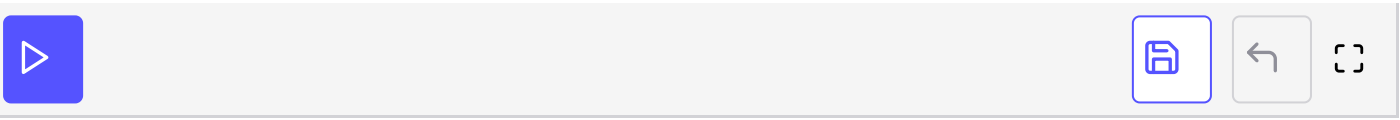


## d) `reverseOrder()`

This is a static method that returns a comparator, which sorts in the reverse order of the natural order.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class ComparatorDemo {

    public static void main(String args[]){
        List<String> fruits = new ArrayList<>();
```

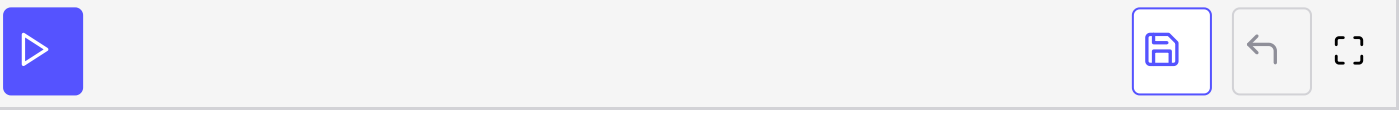
```
        fruits.add("guava");
        fruits.add("peach");
        fruits.add("apple");

        fruits.add("banana");

        // Sorting the List in reverse order.
        Collections.sort(fruits, Comparator.reverseOrder());

        fruits.forEach(System.out::println);
    }
}
```



## e) `nullsFirst` #

`Comparator`'s `nullsFirst()` method takes in a comparator as input and returns a comparator, which considers null values lesser than non-null values.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class ComparatorDemo {

    public static void main(String args[]){
        List<String> fruits = new ArrayList<>();
        fruits.add("guava");
        fruits.add(null);
        fruits.add("apple");
        fruits.add("banana");

        // Sorting the List keeping nulls first.
        Collections.sort(fruits, Comparator.nullsFirst(Comparator.naturalOrder()));

        fruits.forEach(System.out::println);
    }

}
```

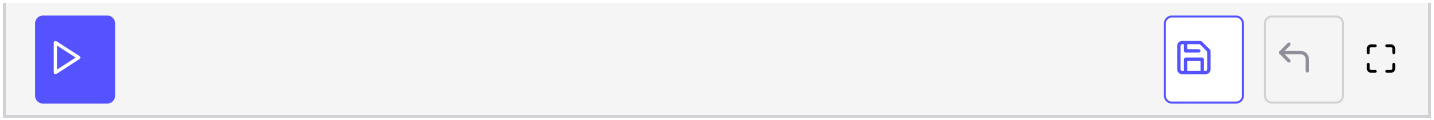


## f) `nullsLast` #

`Comparator`'s `nullsLast()` method takes in a comparator as input and returns a comparator, which considers null values greater than non-null values.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
```

```
public class ComparatorDemo {


    public static void main(String args[]){
        List<String> fruits = new ArrayList<>();
        fruits.add("guava");
        fruits.add(null);
        fruits.add("apple");
        fruits.add("banana");

        // Sorting the List keeping nulls last.
        Collections.sort(fruits, Comparator.nullsLast(Comparator.naturalOrder()));

        fruits.forEach(System.out::println);
    }
}
```



These are the main methods that you should know to write concise and efficient sorting logic using `Comparator`.

---

In the next lesson, we will look at the improvements made in the concurrency API.