HowToDoInJava

Guide to Sorting in Java

Last Updated: April 23, 2022 By: Lokesh Gupta Java Sorting Guides, Java Sorting

Learn to sort a Java `Set`, `List` and `Map` of primitive types and custom objects using Comparator, Comparable and new lambda expressions. We will learn to sort in ascending and descending order as well.

Quick Reference

```
//Sorting an array
Arrays.sort( arrayOfItems );
Arrays.sort( arrayOfItems, Collections.reverseOrder() );
Arrays.sort(arrayOfItems, 2, 6);
Arrays.parallelSort(arrayOfItems);

//Sorting a List
Collections.sort(numbersList);
Collections.sort(numbersList, Collections.reverseOrder());

//Sorting a Set
Set to List -> Sort -> List to Set
Collections.sort(numbersList);

//Sorting a Map
TreeMap<Integer, String> treeMap = new TreeMap<>(map);

unsortedeMap.entrySet()
    .stream()
    .sorted(Map.Entry.comparingByValue())
    .forEachOrdered(x -> sortedMap.put(x.getKey(), x.getValue()));

//Java 8 Lambda
Comparator<Employee> nameSorter = (a, b) -> a.getName().compareToIgnoreCase(b.getName());
Collections.sort(list, nameSorter);

Collections.sort(list, Comparator.comparing(Employee::getName));

//Group By Sorting
Collections.sort(list, Comparator
                       .comparing(Employee::getName)
                       .thenComparing(Employee::getDob));
```

Table Of Contents

1. Sorting a List of Objects

To sort a list of objects, we have two popular approaches i.e. **Comparable** and **Comparator** interfaces. In the upcoming examples, we will sort a collection of `Employee` objects in different ways.

```
public class Employee implements Comparable<Employee> {

    private Long id;
    private String name;
    private LocalDate dob;
}
```

1.1. Comparable Interface

`Comparable` interface enables the **natural ordering of the classes** that implements it. This makes the classes comparable to its other instances.

A class implementing `Comparable` interface must override `compareTo()` method in which it can specify the comparison logic between two instances of the same class.

- **Lists** (and **arrays**) of objects that implement *Comparable* interface can be sorted automatically by `Collections.sort()` and `Arrays.sort()` APIs.
- Objects that implement this interface will be automatically sorted when put in a **sorted map** (as keys) or **sorted set** (as elements).
- It is strongly recommended (though not required) that natural orderings be consistent with `equals()` method. Virtually all Java core classes that implement `Comparable` have natural orderings that are consistent with `equals()`.

Natural Ordering

```
ArrayList<Employee> list = new ArrayList<>();

//add employees..

Collections.sort(list);
```

To sort the list in **reversed order**, the best way is to use the *Comparator.reversed()* API that imposes the reverse ordering.

Natural Ordering

```
ArrayList<Employee> list = new ArrayList<>();

//add employees..

Collections.sort(list, Comparator.reversed());
```

1.2. Comparator Interface

When not seeking the natural ordering, we can take the help of Comparator interface to apply **custom sorting** behavior.

`Comparator` does not require modifying the source code of the class. We can create the comparison logic in a separate class which implements `Comparator` interface and override its `compare()` method.

During sorting, we pass an instance of this comparator to `sort()` method along with the list of custom objects.

For example, we want to sort the list of employees by their first name, while the natural sorting has been implemented by `id` field. So, to sort on name field, we must write the custom sorting logic using *Comparator* interface.

NameSorter.java

```
import java.util.Comparator;

public class NameSorter implements Comparator&lt;Employee&gt;
{
    @Override
    public int compare(Employee e1, Employee e2)
    {
        return e1.getName().compareToIgnoreCase( e2.getName() );
    }
}
```

Notice the use of *NameSorter* in *sort()* method as the second argument in the given example.

```
ArrayList<Employee> list = new ArrayList<>();

//add employees to list

Collections.sort(list, new NameSorter());
```

To do the **reverse sorting**, we just need to call the *reversed()* method on the *Comparator* instance.

```
ArrayList<Employee> list = new ArrayList<>();

//add employees to list

Collections.sort(list, new NameSorter().reversed());
```

1.3. Sorting with Lambda Expressions

Lambda expressions help in writing `Comparator` implementations on the fly. We do not need to create a separate class to provide the one-time comparison logic.

```
Comparator<Employee> nameSorter = (a, b) -> a.getName().compareToIgnoreCase(b.getName());

Collections.sort(list, nameSorter);
```

1.4. Group By Sorting

To apply *SQL style sorting* on a collection of objects on different fields (**group by sort**), we can use **multiple comparators** in a chain. This **chaining of comparators** can be created using *Comparator.comparing()* and *Comparator.thenComparing()* methods.

For example, we can sort the list of employees by name and then sort again by their age.

```
ArrayList<Employee> list = new ArrayList<>();

//add employees to list

Collections.sort(list, Comparator
                        .comparing(Employee::getName)
                        .thenComparing(Employee::getDob));
```

2. Sorting an Array

Use `java.util.Arrays.sort()` method to sort a given array in a variety of ways. The *sort()* is an overloaded method that takes all sorts of types as the method argument.

This method implements a Dual-Pivot Quicksort sorting algorithm that offers ***O(n log(n))* performance** on all data sets and is typically faster than traditional (one-pivot) Quicksort implementations.

2.1. Ascending Order

Java program to sort an array of integers in ascending order using *Arrays.sort()* method.

```
//Unsorted array
Integer[] numbers = new Integer[] { 15, 11, ... };

//Sort the array
Arrays.sort(numbers);
```

2.2. Descending Order

Java provides *Collections.reverseOrder()* comparator to reverse the default sorting behavior in one line. We can use this comparator to sort the array in descending order.

Note that all elements in the array must be *mutually comparable* by the specified comparator.

```
//Unsorted array
Integer[] numbers = new Integer[] { 15, 11, ... };

//Sort the array in reverse order
Arrays.sort(numbers, Collections.reverseOrder());
```

2.3. Sorting Array Range

`Arrays.sort()` method is an overloaded method and takes two additional parameters i.e. `fromIndex` (inclusive) and `toIndex` (exclusive).

When provided above arguments, the array will be sorted within the provided range from position `fromIndex` to position `toIndex`.

Given below is an example to sort the array from element 9 to 18. i.e. {9, 55, 47, 18} will be sorted and the rest elements will not be touched.

```
//Unsorted array
Integer[] numbers = new Integer[] { 15, 11, 9, 55, 47, 18, 1123, 520, 366, 420 };

//Sort the array
Arrays.sort(numbers, 2, 6);

//Print array to confirm
System.out.println(Arrays.toString(numbers));
```

Program output.

```
[15, 11, 9, 18, 47, 55, 1123, 520, 366, 420]
```

2.4. Parallel Sorting

Java 8 introduced lots of new APIs for parallel processing data sets and streams. One such API is `Arrays.parallelSort()`.

The `parallelSort()` method breaks the array into multiple sub-arrays and each sub-array is sorted with `Arrays.sort()` in **different threads**. Finally, all sorted sub-arrays are merged into one sorted array.

The output of the `parallelSort()` and `sort()`, both APIs, will be same at last. It's just a matter of leveraging the Java concurrency.

```
Java parallel sort an array

//Parallel sort complete array
Arrays.parallelSort(numbers);

//Parallel sort array range
Arrays.parallelSort(numbers, 2, 6);

//Parallel sort array in reverse order
Arrays.parallelSort(numbers, Collections.reverseOrder());
```

3. Sorting a Set

There is no direct support for sorting the `Set` in Java. To sort a `Set`, follow these steps:

1. Convert `Set` to `List`.
2. Sort `List` using `Collections.sort()` API.
3. Convert `List` back to `Set`.

```
//Unsorted set
HashSet<Integer> numbersSet = new LinkedHashSet<>(); //with Set items

List<Integer> numbersList = new ArrayList<Integer>(numbersSet) ;        //set -> list

//Sort the list
Collections.sort(numbersList);

//sorted set
numbersSet = new LinkedHashSet<>(numbersList);          //list -> set
```

4. Sorting a Map

A `Map` is the collection of key-value pairs. So logically, we can sort the maps in two ways i.e. **sort by key** or **sort by value**.

4.1. Sort by Key

The best and most effective a sort a map by keys is to add all map entries in **TreeMap** object. `TreeMap` stores the keys in sorted order, always.

```
HashMap<Integer, String> map = new HashMap<>();   //Unsorted Map

TreeMap<Integer, String> treeMap = new TreeMap<>(map);   //Sorted by map keys
```

4.2. Sort by Value

In Java 8, `Map.Entry` class has `static` method **comparingByValue()** to help us in sorting the `Map` by values.

The `comparingByValue()` method returns a **Comparator** that compares `Map.Entry` in natural order on values.

```
HashMap<Integer, String> unSortedMap = new HashMap<>(); //Unsorted Map

//LinkedHashMap preserve the ordering of elements in which they are inserted
LinkedHashMap<Integer, String> sortedMap = new LinkedHashMap<>();

unSortedMap.entrySet()
    .stream()
    .sorted(Map.Entry.comparingByValue())
    .forEachOrdered(x -> sortedMap.put(x.getKey(), x.getValue()));
```

5. Summary

In the above-given examples, we learned to sort an Array, List, Map, and Set.

We saw different ways to initialize and use `Comparator` interface including lambda expressions. We also learned to effectively use the *Comparator* interface.

Happy Learning !!

> No

Recommended Reading:

1. Sorting Arrays in Java
2. Sorting with Comparable and Comparator
3. Guide to Sorting using Hibernate
4. Sorting Streams in Java
5. Sorting a Stream by Multiple Fields in Java
6. Java – Sorting Array of Strings in Alphabetical Order
7. Spring boot pagination and sorting example
8. Sorting a Map by Keys in Java
9. Java Sort Map by Values (ascending and descending orders)
0. Java Collections sort()

Leave a Comment

Name *

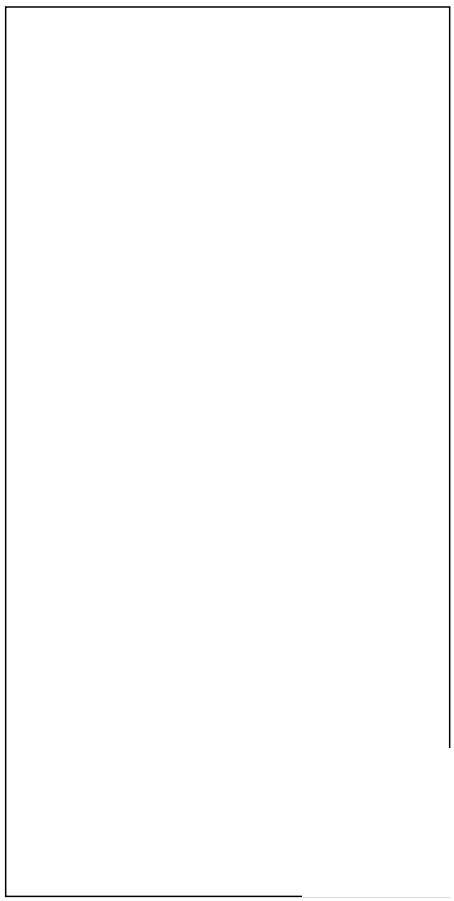
Email *

Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...



- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial