Learn to sort a List of Objects by a field value. Note that if you have millions of records for sorting at a time then a database query is the best way. Otherwise, using either [*Comparable*](https://howtodoinjava.com/java/collections/java-comparable-interface/) or [*Comparator*](https://howtodoinjava.com/java/collections/java-comparator/) interface is a very convenient approach.

Table Of Contents

1. [1. Overview](https://howtodoinjava.com/java/sort/comparable-comparator/#1-overview)
2. [2. Sorting with Comparable for Natural Ordering](https://howtodoinjava.com/java/sort/comparable-comparator/#2-sorting-with-comparable-for-natural-ordering)
   * [2.1. Implementing Comparable Interface](https://howtodoinjava.com/java/sort/comparable-comparator/#21-implementing-comparable-interface)
   * [2.2. Collections.sort() Method](https://howtodoinjava.com/java/sort/comparable-comparator/#22-collectionssort-method)
   * [2.3. Stream.sorted() Method](https://howtodoinjava.com/java/sort/comparable-comparator/#23-streamsorted-method)
3. [3. Sorting with Comparator for Custom Ordering](https://howtodoinjava.com/java/sort/comparable-comparator/#3-sorting-with-comparator-for-custom-ordering)
   * [3.1. Creating Comparator Instances](https://howtodoinjava.com/java/sort/comparable-comparator/#31-creating-comparator-instances)
   * [3.2. Collections.sort()](https://howtodoinjava.com/java/sort/comparable-comparator/#32-collectionssort)
   * [3.3. Stream.sorted()](https://howtodoinjava.com/java/sort/comparable-comparator/#33-streamsorted)
4. [4. hashCode() and equals() Contract](https://howtodoinjava.com/java/sort/comparable-comparator/#4-hashcode-and-equals-contract)
5. [5. Conclusion](https://howtodoinjava.com/java/sort/comparable-comparator/#5-conclusion)

**1. Overview**

In the examples given in this tutorial, we will be using the [record type](https://howtodoinjava.com/java14/java-14-record-type/) *User*. It has four fields: id, firstName, lastName and age. I have chosen these fields purposefully to show different usecases.

Employee.java

**import** java.io.Serializable;

**public** **record** User(Long id, String firstName, String lastName, Integer age)

**implements** Serializable {

**public** User {

**if** (age < 18) {

**throw** **new** IllegalArgumentException("You cannot hire a minor person");

}

}

}

We will be using the given unsorted list and sorting it on different field values.

**private** **static** List<User> getUnsortedUsers() {

**return** Arrays.asList(

**new** User(1L, "A", "Q", Integer.valueOf(24)),

**new** User(4L, "B", "P", Integer.valueOf(22)),

**new** User(2L, "C", "O", Integer.valueOf(27)),

**new** User(3L, "D", "N", Integer.valueOf(29)),

**new** User(5L, "E", "M", Integer.valueOf(25)));

}

Moving on, we will be using the *Comparable* and *Comparator* interfaces for sorting on different field values.

**2. Sorting with Comparable for Natural Ordering**

**2.1. Implementing Comparable Interface**

Comparable interface provides a single method *[compareTo(T o)](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparable.html" \l "compareTo(T)" \t "_blank)* to implement by any class so that two objects of that class can be compared. This method is **used for implementing the natural sorting behavior**.

The *User* record after implementing the *Comparable* interface is as follows. The similar implementation can be done for *class* types as well. The default sorting has been done on the *id* field.

**public** **record** User(Long id, String firstName, String lastName, Integer age)

**implements** Serializable, Comparable<User> {

**public** User {

**if** (age < 18) {

**throw** **new** IllegalArgumentException("You cannot hire a minor person");

}

}

@Override

**public** **int** compareTo(User o) {

**return** **this**.id.intValue() - o.id.intValue();

}

}

**2.2. Collections.sort() Method**

We can pass the List of objects in the *sort()* method that will sort the objects in their natural ordering i.e. by id field.

Collections.sort( list );

Check out the output in the console.

Output

[User[id=1, firstName=A, lastName=Q, age=24],

User[id=2, firstName=C, lastName=O, age=27],

User[id=3, firstName=D, lastName=N, age=29],

User[id=4, firstName=B, lastName=P, age=22],

User[id=5, firstName=E, lastName=M, age=25]]

**2.3. Stream.sorted() Method**

[Java Stream](https://howtodoinjava.com/java/stream/java-streams-by-examples/) API has [sorted()](https://howtodoinjava.com/java8/stream-sorted-method/) method that can sort a stream of items in the natural order. Note that **stream operations do not modify the original collections**, so the objects in the list will be unchanged.

List<User> sortedList = list.stream()

.sorted()

.collect(Collectors.toList());

**3. Sorting with Comparator for Custom Ordering**

**3.1. Creating Comparator Instances**

Let us assume that we want to sort the users list based on some other fields, for example, by firstName or age. We can modify the *User* record because it already implements the natural ordering by id field.

Here comes the Comparator interface to rescue. A **Comparator can be used to define the custom ordering**. To sort on different object fields, we can create multiple *Comparator* implementations.

For example, to sort the users list by firstName, we can create *FirstNameSorter* **class that implements the *Comparator***.

FirstNameSorter.java

**import** java.util.Comparator;

**public** **class** FirstNameSorter **implements** Comparator<User> {

@Override

**public** **int** compare(User o1, User o2) {

**return** o1.firstName().compareTo(o2.firstName());

}

}

Note that we can use the [lambda expression](https://howtodoinjava.com/java8/lambda-expressions/) for creating the **inline *Comparator* instances**, for single-time uses.

Comparator instance using lambda expression

Comparator<User> firstNameSorter = (o1, o2) -> o1.firstName().compareTo(o2.firstName());

We can create **group by sorting effect by combining multiple comparators** using Comparator.thenComparing() method. For example, we can create a complex comparator *fullNameSorter* for sorting a list by *first name* and then by *last name*.

Complex Comparator

Comparator<User> firstNameSorter = (o1, o2) -> o1.firstName().compareTo(o2.firstName());

Comparator<User> lastNameSorter = (o1, o2) -> o1.lastName().compareTo(o2.lastName());

Comparator<User> fullNameSorter = firstNameSorter.thenComparing(lastNameSorter);

One more type of Comparator is worth discussing that is used for **reverse ordering**. We can get this reverse comparator by calling *reversed()* method on any comparator instance.

Reverse Sorting

Comparator<User> reverseSorter = firstNameSorter.reversed();

Similar way, we can create as many comparators as needed in the applications.

**3.2. Collections.sort()**

To sort using *Collection.sort()* method, pass two method arguments. The first argument is the unsorted list and the second argument is the Comparator instance.

List<User> list = getUnsortedUsers();

Comparator<User> firstNameSorter

= (o1, o2) -> o1.firstName().compareTo(o2.firstName());

Collections.sort(list, firstNameSorter);

**3.3. Stream.sorted()**

To sort the stream items using comparator instance, we can pass the comparator as method argument to the *sorted()* method.

List<User> list = getUnsortedUsers();

Comparator<User> firstNameSorter

= (o1, o2) -> o1.firstName().compareTo(o2.firstName());

List<User> sortedList = list.stream()

.sorted(firstNameSorter)

.collect(Collectors.toList());

**4. hashCode() and equals() Contract**

If we have overridden equals() method in the User class, always remember to honor the [contract between hashCode() and equals()](https://howtodoinjava.com/java/basics/java-hashcode-equals-methods/) methods.

If two objects are equal using equals() method then compareTo() method should return zero.

As a general practice, always use the same fields in both methods. If we are using id field in the *equals()* method then use the id field in *compareTo()* method also. An example implementation is given as follows:

**import** java.io.Serializable;

**import** java.util.Objects;

**public** **record** User(Long id, String firstName, String lastName, Integer age)

**implements** Serializable, Comparable<User> {

**public** User {

**if** (age < 18) {

**throw** **new** IllegalArgumentException("You cannot hire a minor person");

}

}

@Override

**public** **int** compareTo(User o) {

**return** **this**.id.intValue() - o.id.intValue();

}

@Override

**public** **int** hashCode() {

**return** Objects.hash(id);

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** true;

**if** (obj == **null**)

**return** false;

**if** (getClass() != obj.getClass())

**return** false;

User other = (User) obj;

**return** Objects.equals(id, other.id);

}

}

**5. Conclusion**

In this *Java Comparable and Comparator* tutorial, we learned to implement both interfaces in different ways for different usecases. We also saw the use of both interfaces in Java Stream API.

Finally, we understood how to correctly override *hashCode()* and *equals()* method on objects to keep sorting functioning properly.

Happy Learning !!