Combining Lambda Expressions ← **Writing and Combining Comparators** → That's the end of the series!

Writing and Combining Comparators

Implementing a Comparator with a Lambda Expression

Thanks to the definition of functional interfaces, the good old `Comparator<T>` interface introduced in JDK 2 became functional. So, implementing a comparator can be done using a lambda expression.

Here is the only abstract method of the `Comparator<T>` interface:

```
@FunctionalInterface
public interface Comparator<T> {

    int compare(T o1, T o2);
}
```

The contract of a comparator is the following:

- If `o1 < o2` then `compare(o1, o2)` should return a negative number
- If `o1 > o2` then `compare(o1, o2)` should return a positive number
- In all cases, `compare(o1, o2)` and `compare(o2, o1)` should have opposite signs.

It is not *strictly* required that in case `o1.equals(o2)` is `true`, the comparison of `o1` and `o2` returns 0.

How can you create a comparator of integers, that would implement the natural order? Well, you can just use the method you saw at the beginning of this tutorial:

```
Comparator<Integer> comparator = (i1, i2) -> Integer.compare(i1, i2);
```

You may have noticed that this lambda expression can also be written with a very nice bound method reference in that way:

```
Comparator<Integer> comparator = Integer::compare;
```

> *Refrain from implementing this comparator with `(i1 - i2)`. Even if this pattern seems to be working, there are corner cases where it will not produce a correct result.*

This pattern can be extended to anything you need to compare, as long as you follow the contract of the comparator.

The `Comparator` API went one step further, by providing a very useful API to create comparators in a much more readable way.

Using a Factory Method to Create a Comparator

Suppose you need to create a comparator to compare strings of characters in a non-natural way: the shortest strings are lesser than the longest strings.

Such a comparator can be written in that way:

```
Comparator<String> comparator =
        (s1, s2) -> Integer.compare(s1.length(), s2.length());
```

You learned in the previous part that it is possible to chain and compose lambda expressions. This code is another example of such a composition. Indeed, you can rewrite it in that way:

```
Function<String, Integer> toLength = String::length;
Comparator<String> comparator =
        (s1, s2) -> Integer.compare(
                toLength.apply(s1),
                toLength.apply(s2));
```

Now you can see that the code of this `Comparator` only depends on the `Function` called `toLength`. So it becomes possible to create a factory method that takes this function as an argument and returns the corresponding `Comparator<String>`.

There is still a constraint of the returned type of the `toLength` function: it has to be comparable. Here it works well because you can always compare integers with their natural order, but you need to keep that in mind.

Such a factory method does exist in the JDK: it has been added to the `Comparator` interface directly. So you can write the previous code in that way:

```java
Comparator<String> comparator = Comparator.comparing(String::length);
```

This `comparing()` method is a static method of the `Comparator` interface. It takes a `Function` as an argument, that should return a type that is an extension of `Comparable`.

Suppose you have a `User` class with a `getName()` getter, and you need to sort a list of users according to their name. The code you need to write is the following:

```java
List<User> users = ...; // this is your list
Comparator<User> byName = Comparator.comparing(User::getName);
users.sort(byName);
```

Chaining Comparators

The company you are working for is currently very happy with the `Comparable<User>` you have delivered. But there is a new requirement in version 2: the `User` class now has a `firstName` and a `lastName`, and you need to produce a new `Comparator` to handle this change.

Writing each comparator follows the same pattern as the previous one:

```java
Comparator<User> byFirstName = Comparator.comparing(User::getFirstName);
Comparator<User> byLastName = Comparator.comparing(User::getLastName);
```

Now what you need is a way to chain them, just as you chained instances of `Predicate` or `Consumer`. The Comparator API gives you a solution to do that:

```java
Comparator<User> byFirstNameThenLastName =
        byFirstName.thenComparing(byLastName);
```

The `thenComparing()` method is a default method of the `Comparator` interface, that takes another comparator as an argument and returns a new comparator. When applied to two users, the comparator first compare these users using the `byFirstName` comparator. If the result is 0, then it will compare them using the `byLastName` comparator. In a nutshell: it works as expected.

The Comparator API went one step further: since `byLastName` only depends on the `User::getLastName` function, an overload of the `thenComparing()` method has been added to the API which takes this function as an argument. So the pattern becomes the following:

```java
Comparator<User> byFirstNameThenLastName =
        Comparator.comparing(User::getFirstName)
                  .thenComparing(User::getLastName);
```

With lambda expressions, method references, chaining, and composition, creating comparators has never been so easy!

Specialized Comparators

Boxing and unboxing or primitive types can also occur with comparators, leading to the same performance hits as in the case of the functional interfaces of the `java.util.function` package. To deal with this problem, specialized versions of the `comparing()` factory method and the `thenComparing()` default method have been added.

You can also create an instance of `Comparator<T>` with:

- `comparingInt(ToIntFunction<T> keyExtractor)`;
- `comparingLong(ToLongFunction<T> keyExtractor)`;
- `comparingDouble(ToDoubleFunction<T> keyExtractor)`.

You use these methods if you need to compare objects using a property that is a primitive type and need to avoid the boxing / unboxing of this primitive type.

There are also corresponding methods to chain `Comparator<T>`:

- `thenComparingInt(ToIntFunction<T> keyExtractor)`;
- `thenComparingLong(ToLongFunction<T> keyExtractor)`;
- `thenComparingDouble(ToDoubleFunction<T> keyExtractor)`.

The idea is the same: using these methods, you can chain the comparison with a comparator built on a specialized function that returns a primitive type, without having any performance hit due to boxing / unboxing.

Comparing Comparable Objects Using Their Natural Order

There are several factory methods worth mentioning in this tutorial, that will help you create simple comparators.

Many classes in the JDK and probably in your application too are implementing a special interface of the JDK: the `Comparable<T>` interface. This interface has one method: `compareTo(T other)` that returns an `int`. This method is used to compare this instance of `T` with `other`, following the contract of the `Comparator<T>` interface.

Many classes of the JDK are already implementing this interface. This is the case for all the wrapper classes of primitive types (`Integer`, `Long` and the like), for the `String` class, and for date and time classes from the Date and Time API.

You can compare the instances of these classes using their natural order, that is, by using this `compareTo()` method. The Comparator API gives you a `Comparator.naturalOrder()` factory class. The comparator it builds does exactly that: it compares any `Comparable` object using its `compareTo()` method.

Having such a factory method can be very useful when you need to chain comparators. Here is an example, where you want to compare string of characters with their length, and then with their natural order (this example uses a static import for the `naturalOrder()` method to further improve readability):

```
Comparator<String> byLengthThenAlphabetically =
        Comparator.comparing(String::length)
                  .thenComparing(naturalOrder());
List<String> strings = Arrays.asList("one", "two", "three", "four", "five");
strings.sort(byLengthThenAlphabetically);
System.out.println(strings);
```

Running this code produces the following result:

```
[one, two, five, four, three]
```

Reversing a Comparator

One major use of comparators is of course to sort lists of objects. The JDK 8 saw the addition of a method on the `List` interface especially for that: `List.sort()`. This method takes a comparator as an argument.

If you need to sort the previous list in reverse order, you can use the nice `reversed()` method from the `Comparator` interface.

```
List<String> strings =
        Arrays.asList("one", "two", "three", "four", "five");
strings.sort(byLengthThenAlphabetically.reversed());
System.out.println(strings);
```

Running this code will produce the following result:

```
[three, four, five, two, one]
```

Dealing with Null Values

Comparing null objects may lead to nasty `NullPointerException` while running your code, something you want to avoid.

Suppose you need to write a null-safe comparator of integers to sort a list of integers. The convention you decide to follow is to push all the null values at the end of your list, meaning that a null value is greater than any other non-null value. And then you want to sort non-null values in their natural order.

Here is the kind of code you may write to implement this behavior:

```
Comparator<Integer> comparator =
        (i1, i2) -> {
            if (i1 == null && i1 != null) {
                return 1;
            } else if (i1 != null && i2 == null) {
                return -1;
            } else {
                return Integer.compare(i1, i2);
            }
        };
```

You can compare this code to the first comparator you wrote at the beginning of this part, and see that readability took a big hit.

Fortunately, there is a much easier way to write this comparator, using another factory method of the `Comparator` interface.

```java
Comparator<Integer> naturalOrder = Comparator.naturalOrder();

Comparator<Integer> naturalOrderNullsLast =
        Comparator.nullsLast(naturalOrder());
```

The `nullsLast()` and its sibling method `nullsFirst()` are factory methods of the `Comparator` interface. Both take a comparator as an argument and do just that: handle the null values for you, pushing them to the end, or putting them first in your sorted list.

Here is an example:

```java
List<String> strings =
        Arrays.asList("one", null, "two", "three", null, null, "four", "five");
Comparator<String> naturalNullsLast =
        Comparator.nullsLast(naturalOrder());
strings.sort(naturalNullsLast);
System.out.println(strings);
```

Running this code produces the following result:

```
[five, four, one, three, two, null, null, null]
```

More Learning

Last update: *February 24, 2023*

Combining Lambda Expressions ← **Writing and Combining Comparators** → That's the end of the series!

Home > Tutorials > Lambda Expressions > Writing and Combining Comparators