x

Java 8 – Powerful Comparison with Lambdas

Last modified: April 2, 2020

by baeldung (https://www.baeldung.com/author/baeldung/)

Java (https://www.baeldung.com/category/java/) •

Java 8 (https://www.baeldung.com/tag/java-8/)

1. Overview

In this tutorial, we're going to take a first look at the **Lambda support in Java 8 – specifically at how to leverage it to write the *Comparator* and sort a Collection**.

This article is part of the "Java – Back to Basic" series (/java-tutorial) here on Baeldung.

Further reading:

The Java 8 Stream API Tutorial (https://www.baeldung.com/java-8-streams)

The article is an example-heavy introduction of the possibilities and operations offered by the Java 8 Stream API.

Read more (https://www.baeldung.com/java-8-streams) →

Guide to Java 8's Collectors (https://www.baeldung.com/java-8-collectors)

The article discusses Java 8 Collectors, showing examples of built-in collectors, as well as showing how to build custom collector.

Read more (https://www.baeldung.com/java-8-collectors) →

Lamb
(https:

Tips and best practices on using Java 8 lambdas and functional interfaces.

Read more (https://www.baeldung.com/java-8-lambda-expressions-tips) →

First, let's define a simple entity class:

```
public class Human {
    private String name;
    private int age;

    // standard constructors, getters/setters, equals and hashcode
}
```

2. Basic Sort Without Lambdas

Before Java 8, sorting a collection would involve **creating an anonymous inner class for the *Comparator*** used in the sort:

```
new Comparator<Human>() {
    @Override
    public int compare(Human h1, Human h2) {
        return h1.getName().compareTo(h2.getName());
    }
}
```

This would simply be used to sort the *List* of *Human* entities:

```
@Test
public void givenPreLambda_whenSortingEntitiesByName_thenCorrectlySorted() {
    List<Human> humans = Lists.newArrayList(
      new Human("Sarah", 10),
      new Human("Jack", 12)
    );

    Collections.sort(humans, new Comparator<Human>() {
        @Override
        public int compare(Human h1, Human h2) {
            return h1.getName().compareTo(h2.getName());
        }
    });
    Assert.assertThat(humans.get(0), equalTo(new Human("Jack", 12)));
}
```

3. Basic Sort With Lambda Support

With the introduction of Lambdas, we can now bypass the anonymous inner class and achieve the same result with **simple, functional semantics**:

```
(final Human h1, final Human h2) -> h1.getName().compareTo(h2.getName());
```

Similarly – we can now test the behavior just as before:

```
@Test
public
    L

      new Human("Jack", 12)
    );

    humans.sort(
      (Human h1, Human h2) -> h1.getName().compareTo(h2.getName()));

    assertThat(humans.get(0), equalTo(new Human("Jack", 12)));
}
```

Notice that we're also using **the new *sort* API added to *java.util.List* in Java 8** – instead of the old *Collections.sort* API.

4. Basic Sorting With No Type Definitions

We can further simplify the expression by not specifying the type definitions – **the compiler is capable of inferring these** on its own:

```
(h1, h2) -> h1.getName().compareTo(h2.getName())
```

And again, the test remains very similar:

```
@Test
public void
  givenLambdaShortForm_whenSortingEntitiesByName_thenCorrectlySorted() {

    List<Human> humans = Lists.newArrayList(
      new Human("Sarah", 10),
      new Human("Jack", 12)
    );

    humans.sort((h1, h2) -> h1.getName().compareTo(h2.getName()));

    assertThat(humans.get(0), equalTo(new Human("Jack", 12)));
}
```

5. Sort Using Reference to Static Method

Next, we're going to perform the sort using a Lambda Expression with a reference to a static method.

First, we're going to define the method *compareByNameThenAge* – with the exact same signature as the *compare* method in a *Comparator<Human>* object:

```
public
    if

    } e
        return lhs.name.compareTo(rhs.name);
    }
}
```

Now, we're going to call the *humans.sort* method with this reference:

```
humans.sort(Human::compareByNameThenAge);
```

The end result is a working sorting of the collection using the static method as a *Comparator*:

```
@Test
public void
  givenMethodDefinition_whenSortingEntitiesByNameThenAge_thenCorrectlySorted() {

    List<Human> humans = Lists.newArrayList(
      new Human("Sarah", 10),
      new Human("Jack", 12)
    );

    humans.sort(Human::compareByNameThenAge);
    Assert.assertThat(humans.get(0), equalTo(new Human("Jack", 12)));
}
```

6. Sort Extracted Comparators

We can also avoid defining even the comparison logic itself by using an **instance method reference** and the *Comparator.comparing* method – which extracts and creates a *Comparable* based on that function.

We're going to use the getter *getName()* to build the Lambda expression and sort the list by name:

```
@Test
public void
  givenInstanceMethod_whenSortingEntitiesByName_thenCorrectlySorted() {

    List<Human> humans = Lists.newArrayList(
      new Human("Sarah", 10),
      new Human("Jack", 12)
    );

    Collections.sort(
      humans, Comparator.comparing(Human::getName));
    assertThat(humans.get(0), equalTo(new Human("Jack", 12)));
}
```

7. Reverse Sort

JDK 8 has also introduced a helper method for **reversing the comparator** – we can make quick use of that to reverse our sort:

```
@Test
public
    L

      new Human("Jack", 12)
    );

    Comparator<Human> comparator
      = (h1, h2) -> h1.getName().compareTo(h2.getName());

    humans.sort(comparator.reversed());

    Assert.assertThat(humans.get(0), equalTo(new Human("Sarah", 10)));
}
```

8. Sort With Multiple Conditions

The comparison lambda expressions need not be this simple – we can write **more complex expressions as well** – for example sorting the entities first by name, and then by age:

```
@Test
public void whenSortingEntitiesByNameThenAge_thenCorrectlySorted() {
    List<Human> humans = Lists.newArrayList(
      new Human("Sarah", 12),
      new Human("Sarah", 10),
      new Human("Zack", 12)
    );

    humans.sort((lhs, rhs) -> {
        if (lhs.getName().equals(rhs.getName())) {
            return lhs.getAge() - rhs.getAge();
        } else {
            return lhs.getName().compareTo(rhs.getName());
        }
    });
    Assert.assertThat(humans.get(0), equalTo(new Human("Sarah", 10)));
}
```

9. Sort With Multiple Conditions – Composition

The same comparison logic – first sorting by name and then, secondarily, by age – can also be implemented by the new composition support for *Comparator*.

Starting with JDK 8, we can now chain together multiple comparators to build more complex comparison logic:

```
@Test
public void
  givenComposition_whenSortingEntitiesByNameThenAge_thenCorrectlySorted() {

    List<Human> humans = Lists.newArrayList(
      new Human("Sarah", 12),
      new Human("Sarah", 10),
      new Human("Zack", 12)
    );

    humans.sort(
      Comparator.comparing(Human::getName).thenComparing(Human::getAge)
    );

    Assert.assertThat(humans.get(0), equalTo(new Human("Sarah", 10)));
}
```

10. Sort

We can alsc

We can sort the stream using natural ordering as well as ordering provided by a *Comparator.* For this, we have two overloaded variants of the *sorted()* API:

- *sorted()* **–** sorts the elements of a *Stream* using natural ordering; the element class must implement the *Comparable* interface.
- *sorted(Comparator<? super T> comparator)* **–** sorts the elements based on a *Comparator* instance

Let's see an example of how to **use the *sorted()* method with natural ordering**:

```
@Test
public final void
  givenStreamNaturalOrdering_whenSortingEntitiesByName_thenCorrectlySorted() {
    List<String> letters = Lists.newArrayList("B", "A", "C");

    List<String> sortedLetters = letters.stream().sorted().collect(Collectors.toList());
    assertThat(sortedLetters.get(0), equalTo("A"));
}
```

Now let's see how we can **use a custom *Comparator* with the *sorted()* API**:

```
@Test
public final void
  givenStreamCustomOrdering_whenSortingEntitiesByName_thenCorrectlySorted() {
    List<Human> humans = Lists.newArrayList(new Human("Sarah", 10), new Human("Jack", 12));
    Comparator<Human> nameComparator = (h1, h2) -> h1.getName().compareTo(h2.getName());

    List<Human> sortedHumans =
      humans.stream().sorted(nameComparator).collect(Collectors.toList());
    assertThat(sortedHumans.get(0), equalTo(new Human("Jack", 12)));
}
```

We can simplify the above example even further if we **use the *Comparator.comparing()* method**:

```
@Test
public final void
  givenStreamComparatorOrdering_whenSortingEntitiesByName_thenCorrectlySorted() {
    List<Human> humans = Lists.newArrayList(new Human("Sarah", 10), new Human("Jack", 12));

    List<Human> sortedHumans = humans.stream()
      .sorted(Comparator.comparing(Human::getName))
      .collect(Collectors.toList());

    assertThat(sortedHumans.get(0), equalTo(new Human("Jack", 12)));
}
```

11. Sorting a List in Reverse With *Stream.sorted()*

We can also use *Stream.sorted()* to sort a collection in reverse.

First, let's se **r() to sort a list in the re**

```
@Test
public final void
  givenStreamNaturalOrdering_whenSortingEntitiesByNameReversed_thenCorrectlySorted() {
    List<String> letters = Lists.newArrayList("B", "A", "C");

    List<String> reverseSortedLetters = letters.stream()
      .sorted(Comparator.reverseOrder())
      .collect(Collectors.toList());

    assertThat(reverseSortedLetters.get(0), equalTo("C"));
}
```

Now, let's see how we can **use the *sorted()* method and a custom *Comparator***.

```
@Test
public final void
  givenStreamCustomOrdering_whenSortingEntitiesByNameReversed_thenCorrectlySorted() {
    List<Human> humans = Lists.newArrayList(new Human("Sarah", 10), new Human("Jack", 12));
    Comparator<Human> reverseNameComparator =
      (h1, h2) -> h2.getName().compareTo(h1.getName());

    List<Human> reverseSortedHumans = humans.stream().sorted(reverseNameComparator)
      .collect(Collectors.toList());
    assertThat(reverseSortedHumans.get(0), equalTo(new Human("Sarah", 10)));
}
```

Note that the invocation of *compareTo* is flipped, which is what is doing the reversing.

Finally, let's simplify the above example by **using the *Comparator.comparing()* method**:

```
@Test
public final void
  givenStreamComparatorOrdering_whenSortingEntitiesByNameReversed_thenCorrectlySorted() {
    List<Human> humans = Lists.newArrayList(new Human("Sarah", 10), new Human("Jack", 12));

    List<Human> reverseSortedHumans = humans.stream()
      .sorted(Comparator.comparing(Human::getName, Comparator.reverseOrder()))
      .collect(Collectors.toList());

    assertThat(reverseSortedHumans.get(0), equalTo(new Human("Sarah", 10)));
}
```

12. Null Values

So far, we implemented our *Comparator*s in a way that they can't sort collections containing *null* values. That is, if the collection contains at least one *null* element, then the *sort* method throws a *NullPointerException*:

```
@Test(expected = NullPointerException.class)
public void givenANullElement_whenSortingEntitiesByName_thenThrowsNPE() {
    List<Human> humans = Lists.newArrayList(null, new Human("Jack", 12));

    humans.sort((h1, h2) -> h1.getName().compareTo(h2.getName()));
}
```

The simplest solution is to handle the *null* values manually in our *Comparator* implementation:

```
@Test
public
    L

    humans.sort((h1, h2) -> {
        if (h1 == null) {
            return h2 == null ? 0 : 1;
        }
        else if (h2 == null) {
            return -1;
        }
        return h1.getName().compareTo(h2.getName());
    });

    Assert.assertNotNull(humans.get(0));
    Assert.assertNull(humans.get(1));
    Assert.assertNull(humans.get(2));
}
```

Here we're pushing all *null* elements towards the end of the collection. To do that, the comparator considers *null* to be greater than non-null values. When both are *null*, they are considered equal.

Additionally, **we can pass any *Comparator* that is not null-safe into the *Comparator.nullsLast() (https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#nullsLast-java.util.Comparator-)* method and achieve the same result**:

```
@Test
public void givenANullElement_whenSortingEntitiesByName_thenMovesTheNullToLast() {
    List<Human> humans = Lists.newArrayList(null, new Human("Jack", 12), null);

    humans.sort(Comparator.nullsLast(Comparator.comparing(Human::getName)));

    Assert.assertNotNull(humans.get(0));
    Assert.assertNull(humans.get(1));
    Assert.assertNull(humans.get(2));
}
```

Similarly, we can use *Comparator.nullsFirst() (https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#nullsFirst-java.util.Comparator-)* to move the *null* elements towards the start of the collection:

```
@Test
public void givenANullElement_whenSortingEntitiesByName_thenMovesTheNullToStart() {
    List<Human> humans = Lists.newArrayList(null, new Human("Jack", 12), null);

    humans.sort(Comparator.nullsFirst(Comparator.comparing(Human::getName)));

    Assert.assertNull(humans.get(0));
    Assert.assertNull(humans.get(1));
    Assert.assertNotNull(humans.get(2));
}
```

It's highly recommended to use the *nullsFirst()* or *nullsLast()* decorators, as they're more flexible and, above all, more readable.

13. Conclusion

This article illustrated the various and exciting ways that a **List can be sorted using Java 8 Lambda Expressions** – moving right past syntactic sugar and into real and powerful functional semantics.

The implementation of all these examples and code snippets **can be found in the GitHub project (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-lambdas)** – this is an Eclipse-based project, so it should be easy to import and run as it is.

X

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE (/ls-course-end)

Learning to "Build your API **with Spring**"?

Enter your email address

>> Get the eBook

25 COMMENTS Oldest

avoid a dependence for this simple example.

Thanks and congratulations about your article.

+ 2 −

Eugen Paraschiv (https://www.baeldung.com/) 6 years ago

Reply to Michel Graciano

Hey Michael – nice catch – I'll go ahead and update the article first thing tomorrow.
Cheers,
Eugen.

+ 0 −

Ashutosh 5 years ago

Reply to Eugen Paraschiv

Can you provide some examples how to use Iterables & Predicates

+ 0 −

Eugen Paraschiv (https://www.baeldung.com/) 5 years ago

Reply to Ashutosh

Iterables and Predicates for sorting, or just in general?

+ 0 −

Richard Langlois 5 years ago

Cool stuff ! Can't wait to use Java 8 in my next project…

+ 0 −

Eugen Paraschiv (https://www.baeldung.com/) 5 years ago

Reply to Richard Langlois

Yeah, I've been using it for a few months and I keep finding better and better ways to improve the code I write by using stuff like the new Stream API, or Optional. Cool stuff indeed. Cheers,
Eugen.

+ 0 −

mikenhill 4 years ago

As the code stands, the following line: "Assert.assertThat(humans.get(0), equalTo(new Human("Jack", 12)));"

Will compare two different object references which are not equal. Would it be better to use:

Assert.assertThat(humans.get(0).getName(), equalTo(new Human("Jack", 12).getName()));

+ 0 −

Eugen Paraschiv (https://www.baeldung.com/) 4 years ago

Reply to mikenhill

Yeah – you can definitely use the names. In this case, it doesn't really matter because the objects are equal according to the implementation of *equals* in *User* (which looks at *age* and *name*), but generally, if it's not safe to use the full value, you can and should certainly use the fields you're interested in. Cheers,
Eugen.

+ 0 −

Carlos Mollapaza 4 years ago

WHERE IS Lists.newArrayList

+ 0 −

X

Eugen Paraschiv (https://www.baeldung.com/) 4 years ago

Reply to Carlos Mollapaza

Hey Carlos – it's where it belongs – in Guava ☺
Joking aside – why would this be relevant for sorting? I'd be happy to add it in if it is. Cheers,
Eugen.

+ 0 −

Phạm Công Quân 4 years ago

His very good article, you have a video of this tutorial is not so, if you ask for links, thank you very much

+ 0 −

Eugen Paraschiv (https://www.baeldung.com/) 4 years ago

Reply to Phạm Công Quân

Hey Phạm, no video of this one. Only a handful of my writeups also have videos. Cheers,
Eugen.

+ 0 −

Phạm Công Quân 4 years ago

Reply to Eugen Paraschiv

ok, thanks

+ 0 −

Load More Comments

Comments are closed on this article!

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT-SIDE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES