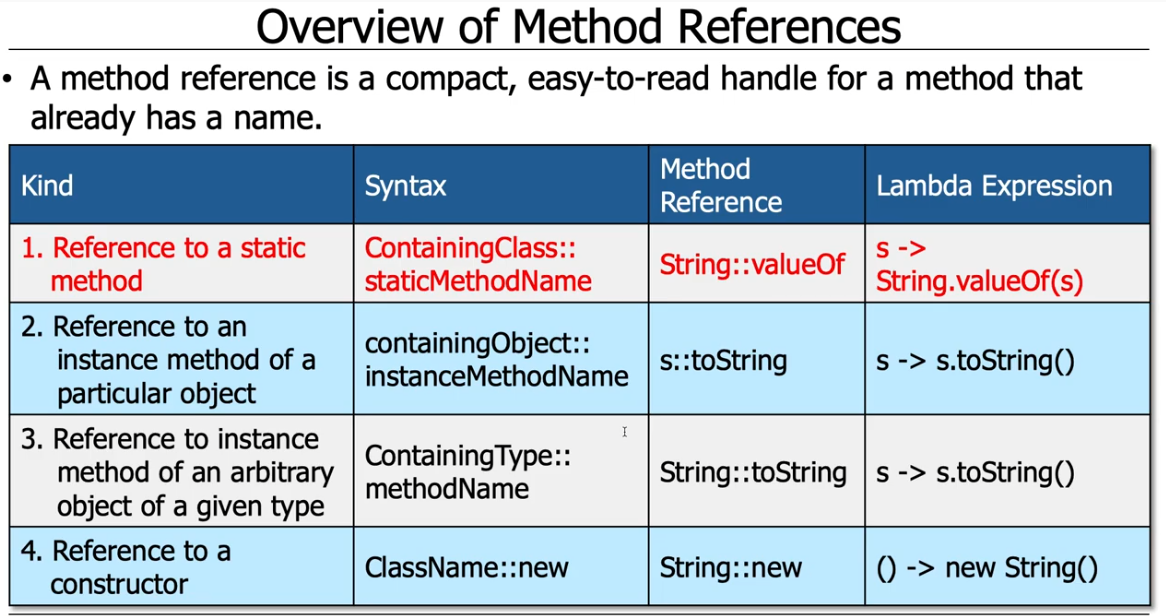
****

<https://codingnconcepts.com/java/double-colon-operator-in-java-8/>

# Double Colon (::) Operator in Java 8

*Method Reference in Java 8*

Ashish Lahoti

Last Modified: April 30, 2020

[Java](https://codingnconcepts.com/categories/java/)

**PAGE CONTENT**

* [**1. Overview**](https://codingnconcepts.com/java/double-colon-operator-in-java-8/#1-overview)
* [**2. Static Method**](https://codingnconcepts.com/java/double-colon-operator-in-java-8/#2-static-method)
* [**3. Instance Method**](https://codingnconcepts.com/java/double-colon-operator-in-java-8/#3-instance-method)
* [**4. Constructor**](https://codingnconcepts.com/java/double-colon-operator-in-java-8/#4-constructor)
* [**5. Super Method**](https://codingnconcepts.com/java/double-colon-operator-in-java-8/#5-super-method)
* [**6. Real World Practical example**](https://codingnconcepts.com/java/double-colon-operator-in-java-8/#6-real-world-practical-example)
* [**7. Conclusion**](https://codingnconcepts.com/java/double-colon-operator-in-java-8/#7-conclusion)

The double colon :: operator is introduced in Java 8 for method reference. It is a shorthand syntax for lambda expression that executes one method. You can write more compact and readable code using double colon operator as compare to anonymous classes and lambda expression. We are going to understand its usage with lots of examples.

## 1. Overview

##### Prerequisite

You should have a basic understanding of ***functional interface***, ***lambda expressions*** and ***streams*** to understand these examples, if not then don’t worry, you will learn those concepts as well along the way.

##### Double colon refers to a method

Double colon :: is basically refers to a single method, and this single method can be a

1. **A Static method** ClassName::staticMethodName  
   e.g. Integer.parseInteger, Double.parseDouble
2. **An Instance method** Object::instanceMethodName  
   e.g. System.out::println, String::toUpperCase
3. **A Constructor** ClassName::new
4. **A Super method** super::parentMethodName

##### Double colon returns a functional interface

Double colon :: always return a functional interface. There are two ways to use this returned functional interface -

1. Use this to **initialize a functional interface** and later execute a function.  
   Here we either use java built-in functional interface like Function, Supplier, Consumer, BiFunction or we create our custom functional interface using @FunctionalInterface annotation.
2. Use this to **replace lambda expression** in streams.

Don’t worry if it’s looking too complicated. Let’s deep dive into examples to understand them better.

## 2. Static Method

###### Syntax

// Lambda expression

(args) -> ClassName.staticMethodName(args)

// Method Reference

ClassName::staticMethodName

#### 2.1 Calculator

Let’s create two **static** methods square and multiply in Calculator class.

class Calculator {

public static double square(double num){

return Math.pow(num, 2);

}

public static double multiply(double num1, double num2) {

return num1\*num2;

}

}

###### 2.1.1 Initialize Functional Interface

Let’s initialize java built-in Function, BiFunction functional interface and apply it to find square and multiply respectively.  
Function interface accepts one argument and returns one argument. Here, it takes one Double arg and return its square.  
BiFunction interface accepts two arguments and returns one argument. Here, it takes two Double args and return its multiplication.

// Initialize functional interface

Function<Double, Double> square = Calculator::square;

BiFunction<Double, Double, Double> multiply = Calculator::multiply;

// Execute function

square.apply(3.0);

multiply.apply(2.5, 5.0)

Output

9.0

12.5

###### 2.1.2 Replace Lambda Expression

We generate square of list of Integer by replacing lambda expression with method reference.

List<Integer> numbers = Arrays.asList(1,2,3);

// Lambda expression

numbers.stream().map(number -> Calculator.square(number)).forEach(number -> System.out.println(number));

// Method Reference

numbers.stream().map(Calculator::square).forEach(System.out::println);

Output

1.0

4.0

9.0

#### 2.2 Integer::parseInt

We know that parseInt is a static method in Integer class.

###### 2.2.1 Initialize Functional Interface

Let’s initialize java built-in Function functional interface and apply it to parse String to Integer.  
Function interface takes one argument and returns one argument. Here in example, it takes a **String** argument and returns **Integer**.

// Initialize functional interface

Function<String, Integer> parseInt = Integer::parseInt;

// Execute function

parseInt.apply("2019");

Output

2019

###### 2.2.2 Replace Lambda Expression

Let’s parse a list of String to **Integer** using its static **parseInt** method. We see that how we can replace lambda expression with method reference resulting in same output.

List<String> years = Arrays.asList("2019", "2020", "2021");

// Lambda Expression

years.stream().map(year -> Integer.parseInt(year)).forEach(year -> System.out.println(year));

// Method Reference

years.stream().map(Integer::parseInt).forEach(System.out::println);

Output

2019

2020

2021

## 3. Instance Method

###### Syntax

// Lambda expression

(args) -> object.instanceMethodName(args)

// Method Reference

object::instanceMethodName

#### 3.1 Calculator

Let’s create two **instance** methods square and multiply in Calculator class.

class Calculator {

public double square(double num){

return Math.pow(num, 2);

}

public double multiply(double num1, double num2) {

return num1\*num2;

}

}

###### 3.1.1 Initialize Functional Interface

Let’s initialize java built-in Function, BiFunction functional interface and apply it to find square and multiply respectively.

// Initialize functional interface

Function<Double, Double> square = new Calculator()::square;

BiFunction<Double, Double, Double> multiply = new Calculator()::multiply;

// Execute function

square.apply(3.0);

multiply.apply(2.5, 5.0)

Output

9.0

12.5

###### 3.1.2 Replace Lambda Expression

We generate square of list of Integer by replacing lambda expression with method reference.

List<Integer> numbers = Arrays.asList(1,2,3);

// Lambda expression

numbers.stream().map(number -> new Calculator().square(number)).forEach(number -> System.out.println(number));

// Method Reference

numbers.stream().map(new Calculator()::square).forEach(System.out::println);

Output

1.0

4.0

9.0

#### 3.2 System.out::println

###### 3.2.1 Initialize Functional Interface

Let’s initialize java built-in Supplier functional interface to print a String.  
Supplier interface accepts one argument and returns nothing. Here in example, it accepts a **String** argument and print it.

// Initialize functional interface

Consumer<String> println = System.out::println;

// Execute function

println.accept("Learning Method Reference a.k.a Colon Operator ::");

Output

Learning Method Reference a.k.a Colon Operator ::

###### 3.2.2 Replace Anonymous Class and Lambda Expression

We will print a list of String and see how double colon :: method reference makes the code more concise and readable as compare to anonymous class and lambda expression.

List<String> languages = Arrays.asList("java", "javascript", "css");

// Anonymous Class

languages.forEach(new Consumer<String>() {

@Override

public void accept(String str) {

System.out.println(str);

}

});

// Lambda expression

languages.forEach(str -> System.out.println(str));

// Method Reference

languages.forEach(System.out::println);

Output

java

javascript

css

###### 3.2.3 Print a list of Integer

Let’s print a list of integer using lambda expression and double colon :: method reference.

List<Integer> numbers = Arrays.asList(1,2,3);

// Lambda expression

numbers.forEach(number->System.out.println(number));

// Method Reference

numbers.forEach(System.out::println);

Output

1

2

3

#### 3.3 String::toUpperCase

###### 3.3.1 Initialize Functional Interface

Let’s initialize java built-in Function functional interface and execute it to change String to UPPERCASE.

// Initialize functional interface

Function<String, String> toUpperCase = String::toUpperCase;

// Execute function

toUpperCase.apply("java");

Output

JAVA

###### 3.3.2 Replace Lambda Expression

Let’s look at the example where we use multiple method references to print the String in uppercase.

List<String> languages = Arrays.asList("java", "javascript", "css");

// Lambda expression

languages.stream().map(str -> str.toUpperCase()).forEach(str -> System.out.println(str));

// Method Reference

languages.stream().map(String::toUpperCase).forEach(System.out::println);

Output

JAVA

JAVASCRIPT

CSS

## 4. Constructor

Double colon :: operator can be used to create an instance by calling constructor.

###### Syntax

ClassName::new

#### 4.1 Create an int[10] array

// Default

int[] array1 = new int[10];

// Method Reference

IntFunction<int[]> arrayMaker = int[]::new;

int[] array2 = arrayMaker.apply(10);

#### 4.2 Create HashMap

// Default

Map map1 = new HashMap();

// Method Reference

Supplier<Map> mapMaker = HashMap::new;

Map map2 = mapMaker.get();

## 5. Super Method

###### Syntax

super::parentMethodName

We create ***square*** and ***multiply*** instance methods in Calculator class and then call those methods in our inherited AdvanceCalculator class using super::parentMethodName to create advance methods ***squareAndAdd*** and ***squareAndMultiply***

class Calculator {

public double square(double num){

return Math.pow(num, 2);

}

public double multiply(double num1, double num2) {

return num1\*num2;

}

}

class AdvanceCalculator extends Calculator {

public double squareAndAdd(double num1, double num2) {

Function<Double, Double> square = super::square;

return square.apply(num1) + square.apply(num2);

}

public double squareAndMultiply(double num1, double num2) {

Function<Double, Double> square = super::square;

BiFunction<Double, Double, Double> multiply = super::multiply;

return multiply.apply(square.apply(num1), square.apply(num2));

}

}

## 6. Real World Practical example

Let’s create a Class Tutorial with properties name, duration and rating.

class Tutorial {

private String name;

private Integer duration;

private Double rating;

public Tutorial(String name) {

this.name = name;

this.duration = 0;

this.rating = 0.0;

}

public Tutorial(String name, Integer duration) {

this.name = name;

this.duration = duration;

this.rating = 0.0;

}

public Tutorial(String name, Integer duration, Double rating) {

this.name = name;

this.duration = duration;

this.rating = rating;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Integer getDuration() {

return duration;

}

public void setDuration(Integer duration) {

this.duration = duration;

}

public Double getRating() {

return rating;

}

public void setRating(Double rating) {

this.rating = rating;

}

@Override

public String toString() {

return "Tutorial[ " + name + "\t- " + duration + "min, rating=" + rating + " ]";

}

public static int compareByRating(Tutorial t1, Tutorial t2) {

return t1.getRating().compareTo(t2.getRating());

}

public static int compareByDuration(Tutorial t1, Tutorial t2) {

return t1.getDuration().compareTo(t2.getDuration());

}

}

Let’s create a list of tutorials objects:-

// List of tutorials

List<Tutorial> tutorials = Arrays.asList(new Tutorial[] {

new Tutorial("Streams in Java 8", 30, 4.2),

new Tutorial("What's new in Java 11", 25, 4.8),

new Tutorial("Core Java Concepts", 45, 3.5)});

#### 6.1 Print the list of tutorials

// Lambda Expression

tutorials.forEach(tutorial -> System.out.println(tutorial));

// Method Reference

tutorials.forEach(System.out::println);

Output

Tutorial[ Streams in Java 8 - 30min, rating=4.2 ]

Tutorial[ What's new in Java 11 - 25min, rating=4.8 ]

Tutorial[ Core Java Concepts - 45min, rating=3.5 ]

#### 6.2 Get list of tutorial names in uppercase

// Lambda Expression

tutorials.stream().map(tutorial -> tutorial.getName()).map(name -> name.toUpperCase()).forEach(s -> System.out.println(s));

// Method Reference

tutorials.stream().map(Tutorial::getName).map(String::toUpperCase).forEach(System.out::println);

Output

STREAMS IN JAVA 8

WHAT'S NEW IN JAVA 11

CORE JAVA CONCEPTS

#### 6.3 Sort Tutorials by Rating

// Lambda Expression

tutorials.stream().sorted((tutorial1, tutorial2) -> Tutorial.compareByRating(tutorial1, tutorial2)).forEach(tutorial -> System.out.println(tutorial));

// Method Reference

tutorials.stream().sorted(Tutorial::compareByRating).forEach(System.out::println);

Output

Tutorial[ Core Java Concepts - 45min, rating=3.5 ]

Tutorial[ Streams in Java 8 - 30min, rating=4.2 ]

Tutorial[ What's new in Java 11 - 25min, rating=4.8 ]

#### 6.4 Sort Tutorials by Duration

// Lambda Expression

tutorials.stream().sorted((tutorial1, tutorial2) -> Tutorial.compareByDuration(tutorial1, tutorial2)).forEach(tutorial -> System.out.println(tutorial));

// Method Reference

tutorials.stream().sorted(Tutorial::compareByDuration).forEach(System.out::println);

Output

Tutorial[ What's new in Java 11 - 25min, rating=4.8 ]

Tutorial[ Streams in Java 8 - 30min, rating=4.2 ]

Tutorial[ Core Java Concepts - 45min, rating=3.5 ]

#### 6.5 Create New Instance of Tutorial

Each constructor method reference returns a Functional Interface. One-arg constructor returns Function and two-arg constructor returns BiFunction functional interface comes default in Java 8 so we are creating TriFunction functional interface on our own to call three-arg constructor.

@FunctionalInterface

interface TriFunction<A, B, C, R> {

R apply(A a, B b, C c);

default <V> TriFunction<A, B, C, V> andThen( Function<? super R, ? extends V> after) {

Objects.requireNonNull(after);

return (A a, B b, C c) -> after.apply(apply(a, b, c));

}

}

// Create an instance from one arg constructor

Function<String, Tutorial> tutorial1 = Tutorial::new;

Tutorial t1 = tutorial1.apply("Tutorial 1");

// Create an instance from two arg constructor

BiFunction<String, Integer, Tutorial> tutorial2 = Tutorial::new;

Tutorial t2 = tutorial2.apply("Tutorial 2", 25);

// Create an instance from three arg constructor

TriFunction<String, Integer, Double, Tutorial> tutorial3 = Tutorial::new;

Tutorial t3 = tutorial3.apply("Tutorial 3", 30, 4.9);

Arrays.asList(t1, t2, t3).forEach(System.out::println);

Output

Tutorial[ Tutorial 1 - 0min, rating=0.0 ]

Tutorial[ Tutorial 2 - 25min, rating=0.0 ]

Tutorial[ Tutorial 3 - 30min, rating=4.9 ]

## 7. Conclusion

In this article, we saw how to use double colon operator introduced in Java 8. It is very useful to keep your code concise and readable specially in streams where you can replace lambda expressions with method reference using double colon operator.

Modern IDEs such as Eclipse (Quick Fix Feature) and IntelliJ IDEA (Intention Feature) provide built in support to convert lambda expression to an equivalent method reference.

**. 1. Overview**

One of the most welcome changes in Java 8 was the introduction of [lambda expressions](https://www.baeldung.com/java-8-lambda-expressions-tips), as these allow us to forego anonymous classes, greatly reducing boilerplate code and improving readability.

**Method references are a special type of lambda expressions**. They're often used to create simple lambda expressions by referencing existing methods.

There are four kinds of method references:

* Static methods
* Instance methods of particular objects
* Instance methods of an arbitrary object of a particular type
* Constructor

**2. Reference to a Static Method**

We'll begin with a very simple example, capitalizing and printing a list of *Strings*:

List<String> messages = Arrays.asList("hello", "baeldung", "readers!");

We can achieve this by leveraging a simple lambda expression calling the [*StringUtils.capitalize()*](https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/StringUtils.html#capitalize-java.lang.String-) method directly:

messages.forEach(word -> StringUtils.capitalize(word));

Or, we can use a method reference to simply refer to the *capitalize* static method:

messages.forEach(StringUtils::capitalize);

**Notice that method references always utilize the *::* operator.**

**3. Reference to an Instance Method of a Particular Object**

To demonstrate this type of method reference, let's consider two classes: