

[← \(\)Previous](#)

Java 8 - Lambdas, Method References and Composition

[Next → \(java8_default_methods.jsp\)](#)

Introduction

The introduction of functional programming in Java8 has caused quite a buzz amongst java enthusiasts. It is probably the biggest change in Java since the introduction of generics. Java has lacked this feature which has been quite popular in other languages such as Lisp, Clojure, Haskell, R etc. In this slightly length..ish tutorial we cover the following:



1. Introduction to functional programming in Java.
2. Lambda Expressions in Java - Introduction and examples
3. Lambda Expressions - Functional Interfaces
4. Method References
5. Constructor References
6. Composition

Introduction to Functional programming (in java)

Developers in Java are familiar with Object oriented programming. The 'hero' there is an 'object'. Everything happens through an object. You can 'call' the method on an object, or change the state of an object. You can pass an object to a method or return an object from a method. In Functional programming, the 'hero' is the function. You dont pass a value but a function around. The function may take in a value and return an argument. In a sense, you are passing a *behaviour* around instead of state. From a java perspective we dont have to know more about functional programming, because here Lambda expressions are the drivers of functional programming and we look at them next.

Lambda Expressions in Java

The biggest addition in Java8 has been lambda expressions. The advantages of lambda expression are:

1. It makes the code more readable.
2. It makes coding faster, especially in Collections, which we shall see later in another tutorial
3. It makes parallel processing easier (with respect to Collections)

The widely used example to introduce lambdas is in java AWT. Here's how you would write a typical ActionListener for button.

```
1 Button button = new Button();
2 button.addActionListener(new ActionListener() {
3     @Override
4     public void actionPerformed(ActionEvent arg0) {
5         System.out.println("Do Something");
```

```

6 |     }
7 | });

```

Here's the lambda way.

```

1 | button.addActionListener(event->System.out.println("Do Something"));

```

What you see in the bracket is a lambda expression. There are three parts to a lambda expression.

- The first part on the left are the arguments to the lambda expression. Lambda expression may take 0, 1 or more arguments. The type of the arguments may be defined or may be inferred depending on the context. In the above expression we did not specify the type of the event, but the compiler inferred it based on the Context.
- The second part is the symbol '->'. Look closely at this symbol, very closely ... it means... nothing! It just separates the left from the right.
- The third part is the body. It could be a single statement, or a block of statements and it may return results.

So what's the big deal? just that it makes the code easier to read and faster to develop. Wait till you see Collections and then you would understand the benefit of it. There are different ways to write the lambda expression. Lets look at them:

1. Empty brackets are used when there are no arguments.

```

1 | ()->System.out.println("No Arguments, only behavior")

```

2. You can have a single argument and ignore the brackets.

```

1 | event->System.out.println("One Argument, No Bracket for
2 |     argument")

```

- 3.

```

1 | () -> {
2 |     System.out.println("First statement");
3 |     System.out.println("Second Statement");
4 | };

```

4. This expression takes in two arguments enclosed in bracket but without the type specified

```

1 | (x, y) -> x + y;

```

we will look at more examples later in this tutorial.

Lambda Expressions - Functional Interfaces

Lambda expressions have a type and are assignable. So when we say

```

1 | ()->System.out.println("")

```

This expression has a type and the type is a *Functional Interface*. We can therefore assign a lambda expression to a functional interface, and any expression that is assignable to a functional interface can be used as a lambda expression. Before we look at examples, lets understand what a functional interface is. It is an Interface that has only one abstract method. (Note that it may have other methods that have an implementation, the so called default

methods, we will look at it in the next tutorial, but for now, just remember that java8 allows interfaces to have methods with an implementation.) java already had some functional interfaces. for example, the *Runnable* interface with exact one method *run* is a functional interface. We can assign a lambda expression that does not take an argument and does not return a value to a Runnable Interface

```
1 Runnable someFunction = () -> System.out.println("test");
```

Remember : Lambda Expression should be assignable to a Functional Interface.

Functional Interfaces can be annotated with `@FunctionalInterface` . For these interfaces the compiler checks explicitly whether it has only one unimplemented method. It is not necessary for a functional interface to be annotated with this annotation. Lets look at some of the functional interfaces added in Java8. They are mainly in the **java.util.Function** package. Lets look at some of the interfaces.

Name	Method	Example/Explanation
Function<T,R>	R apply(T t); The lambda expression is the body of the apply() method. In other words, whenever apply() is called , the lambda expression is evaluated. Note that the type of the input parameter and the return	<pre> 1 Function<String, String> function = x -> x.toUpperCase(); 2 Function<String, String> function2 = x -> x.toLowerCase(); 3 convertString(function);// prints STRANGE 4 convertString(function2);// prints strange 5 public static void convertString(Function<String, String> function){ 6 System.out.println(function.apply("StRaNgE")); 7 } 8 </pre>

values is
not
required
since its
inferred
from the
signature
of the
apply
method.

Supplier<T>

T get();

```

1 Supplier<String> supplier1 = () -> "String1";
2 Supplier<String> supplier2 = () -> "String2";
3 printSuppliedString(supplier1);
4 printSuppliedString(supplier2);
5 public static void printSuppliedString(Supplier<String> supplier){
6     System.out.println(supplier.get());
7 }
8

```

Consumer<T>

void
accept(T
t);

```

1 Consumer<String> function = x -> System.out.println(x);
2 Consumer<String> function2 = x -> System.out.println(x.toLowerCase());
3 consumeString(function, "StringA");// prints StringA
4 consumeString(function2, "StringA");// prints stringa
5 public static void consumeString(Consumer<String> consumer, String x) {
6     consumer.accept(x);
7 }
8
9

```

interface
Predicate<T>

boolean
test(T t);
Use this if
you want
to write a
lambda
expression
that
returns a
boolean.

```

1 Predicate<Double> function = x -> x > 10;
2 Predicate<Double> function2 = x -> x < -10;
3 System.out.println(function.test(new Double(9)));// prints false
4 System.out.println(function2.test(new Double(-20)));// prints true
5 public static void testValue(Predicate<Double> predicate, Double d){
6     predicate.test(d);
7 }
8

```

IntConsumer	void accept(int value);	This function accepts a primitive int and does not return any value
BiFunction<T, U, R>	R apply(T t, U u);	This function accepts two arguments and returns a value.
BinaryOperator<T> extends BiFunction<T,T,T>	R apply(T t, U u);	This function is a special case of BiFunction where both the input parameters and the return is of the same type.

Method References

To increase readability java8 has come up with method references. You can access a method (lambda expression) using the :: notation. The only condition that the methods need to follow is that they should be assignable to any FunctionalInterface as described above. There are four kinds of method references, we look at them in below.

Remember: Method References should be assignable to a FunctionalInterface

The Example uses the following class :

```

1  public class Example {
2
3      public int add(int a, int b) {
4          return a + b;
5      }
6
7      public static int mul(int a, int b) {
8          return a * b;
9      }
10
11     public String lower(String a) {
12         return a.toLowerCase();
13     }
14
15     public void printDate(Date date) {
16         System.out.println(date);
17     }
18
19     public void oper(IntBinaryOperator operator, int a, int b) {
20         System.out.println(operator.applyAsInt(a, b));

```

```

21     }
22
23     public void operS(Function<String, String> stringOperator, String a) {
24         System.out.println(stringOperator.apply(a));
25     }
26
27     public GregorianCalendar operC(Supplier<GregorianCalendar> supplier) {
28         return supplier.get();
29     }
30
31 }

```

Type	Lambda Form	Method Reference	Comments
Referencing static methods using Class Name	<code>ex.oper((a, b) -> Example.mul(a, b), 1, 1);</code>	<code>ex.oper(Example::mul, 1, 2);</code>	The mul or multiply method is a static method of the Example class.
Referencing Instance methods using Object Instance	<code>ex.oper((a, b) -> ex.add(a, b), 1, 2);</code>	<code>ex.oper(ex::add, 1, 2);</code>	The add method is called using an instance of the example.
Referencing Instance methods using Class Name	<code>ex.operS(s->s.toLowerCase(), "STRING");</code>	<code>ex.operS(String::toLowerCase, "STRING");</code>	This can be a little confusing. So we are calling a non static method but using a class name. Actually the instance of the class is passed when the method is called.
Referencing the constructor	<code>ex.operC(()->{ return new GregorianCalendar();})</code>	<code>ex.operC(GregorianCalendar::new);</code>	A shortcut to a constructor. Remember that <code>GregorianCalendar::new</code> is assignable to a

Lambda Expressions - Composition

Composition allows applying lambda expressions one after another. There are two methods:

Function compose(Function before) - The before function is applied first and then the calling function


Function andThen(Function after) - The after function is applied after the calling function

Lets look at an example. This example creates a lambda expression for the Math functions. We can then apply one math function after another by using the *compose* and *andThen* methods. We can create a *exp(log(sin))* or *log(exp(sin))* etc .


```
1      package com.studytrails.java8.lambdas;
2
3      import java.util.function.Function;
4
5      public class ExampleCompose {
6
7          public static void main(String[] args) {
8              ExampleCompose ex = new ExampleCompose();
9              Function<Double , Double> sin = d -> ex.sin(d);
10             Function<Double , Double> log = d -> ex.log(d);
11             Function<Double , Double> exp = d -> ex.exp(d);
12             ExampleCompose compose = new ExampleCompose();
13             System.out.println(compose.calculate(sin.compose(log), 0.8));
14             // prints log:sin:-0.22
15             System.out.println(compose.calculate(sin.andThen(log), 0.8));
16             // prints sin:log:-0.33
17             System.out.println(compose.calculate(sin.compose(log).andThen(exp), 0.8));
18             //log:sin:exp:0.80
19             System.out.println(compose.calculate(sin.compose(log).compose(exp), 0.8));
20             //exp:log:sin:0.71
21             System.out.println(compose.calculate(sin.andThen(log).compose(exp), 0.8));
22             //exp:sin:log:-0.23
23             System.out.println(compose.calculate(sin.andThen(log).andThen(exp), 0.8));
24             //sin:log:exp:0.71
25
26         }
27
28         public Double calculate(Function<Double , Double> operator, Double d) {
29             return operator.apply(d);
30         }
31
32         public Double sin(Double d) {
33             System.out.print("sin:");
```

```
34     return Math.sin(d);
35 }
36
37 public Double log(Double d) {
38     System.out.print("log:");
39     return Math.log(d);
40 }
41
42 public Double exp(Double d) {
43     System.out.print("exp:");
44     return Math.exp(d);
45 }
46
47 }
48
```

This wraps up our first article on Lambdas. This article covered the basics of lambdas, Method and constructor references and composition. In the next article we look at Streams and how Java8 revolutionizes the way Collections work. (coming soon...)

 ()Previous

Java 8 - Lambdas, Method References and Composition

Next  (java8_default_methods.jsp)

2 Comments **StudyTrails****1** Login ▾♥ Recommend 4  Share

Sort by Best ▾



Join the discussion...

**Oleg** • 5 months ago

Function<double ,="" double<="" sin="d" -=""> ex.sin(d);

Syntax mistake detected :)

^ | ▾ • Reply • Share ›

**Mithil Shah - StudyTrails** → Oleg • 5 months ago

Thanks for pointing that out. fixed it.

^ | ▾ • Reply • Share ›

 Subscribe Add Disqus to your site Add Disqus Add PrivacySitemap (http://www.studytrails.com/sitemap_index.xml)

© Copyright StudyTrails 2012-2014