HowToDoInJava

# Lambda Expressions in Java

Last Updated: January 9, 2022 | By: Lokesh Gupta | Java 8 | Functional Interface, Java 8, Lambda Expression

Lambda expressions are known to many of us who have worked on advanced languages like Scala. The term "*lambda*" has its origin in Lambda calculus that uses the Greek letter lambda (**λ**) to denote a *function abstraction*.

Lambda expressions were introduced to Java as part of Java 8 release.

Table Of Contents



## 1. What are Lambda Expressions?

In general programming language, a Lambda expression (or function) is an **anonymous function**, i.e., a **function without any name or identifier, and with a list of formal parameters and a body**. An arrow (–>) is used to separate the list of parameters and the body.

> In Java, a lambda expression is an expression that represents an **instance of a functional interface**.

Similar to other types in Java, lambda expressions are also typed, and their type is a functional interface type. To infer the type, the compiler looks at the left side of the assignment in a lambda expression.

Note that the lambda expression itself does not contain the information about which functional interface it is implementing. This information is deduced from the context in which expression is used.

## 2. Lambda Expression Example

A typical lambda expression syntax will be like this:

```
(parameters) -> expression
```

For example, the below-given lambda expression takes two parameters and returns their addition. Based on the type of `x` and `y`, the expression will be used differently.

- If the parameters match to `Integer` the expression will add the two numbers.
- If the parameters of type `String` the expression will concat the two strings.

```
(x, y) -> x + y
```

For example, we have the following functional interface *Operator*. It has one method *process()* that takes two parameters and returns a value.

```
@FunctionalInterface
interface Operator<T> {
```

```
    T process(T a, T b);
}
```

We can create lambda expressions for this functional interface in the following manner. Notice we are able to create the method implementations and immediately use them. **We do not need to create a concrete class** *OperatorImpl* that implements *Operator* interface.

```
Operator<Integer> addOperation = (a, b) ->  a + b;
System.out.println(addOperation.process(3, 3));     //Prints 6

Operator<String> appendOperation = (a, b) ->  a + b;
System.out.println(appendOperation.process("3", "3"));  //Prints 33

Operator<Integer> multiplyOperation = (a, b) ->  a * b;
System.out.println(multiplyOperation.process(3, 3));    //Prints 9
```

Two good examples of functional interface types are Consumer and BiConsumer interfaces that are heavily used in Stream API for creating lambda expressions.

## 3. Features of Lambda Expressions

- A lambda expression can have **zero, one or more parameters**.

```
(x, y) -> x + y
(x, y, z) -> x + y + z
```

- The body of the lambda expressions can contain **zero, one or more statements**. If the body of lambda expression has a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in the body then these must be enclosed in curly brackets.

```
(parameters) -> { statements; }
```

- The type of the parameters can be explicitly declared or it can be inferred from the context. In previous example, the type of *addOperation* and *appendOperation* is derived from context.
- Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.

```
() -> expression
```

- When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.

```
a -> return a * a;
```

- A lambda expression cannot have a *throws* clause. It is inferred from the context of its use and its body.
- Lambda expressions cannot be *generic* i.e. they cannot declare type parameters.

## 4. More Examples

We are listing out some code samples which you can read and analyze how a lambda expression can be used in the day-to-day programming.

#### Example 1: Using lambda expression to iterate over a List and perform some action on list items

In the given example, we are iterating over the list and printing all the list elements in the standard output. We can perform any desired operation in place of printing them.

```
List<String> pointList = new ArrayList();

pointList.add("1");
pointList.add("2");

pointList.forEach( p ->  { System.out.println(p); } );
```

**Example 2: Using lambda expression to create and start a Thread in Java**

In given example, we are passing the instance of `Runnable` interface into the `Thread` constructor.

```
new Thread(
    () -> System.out.println("My Runnable");
).start();
```

**Example 3: Using lambda expression for adding an event listener to a GUI component**

```
JButton button =  new JButton("Submit");
button.addActionListener((e) -> {
    System.out.println("Click event triggered !!");
});
```

Above are very basic examples of lambda expressions in java 8. I will be coming up with more useful examples and code samples from time to time.

## 5. Advantages of Lambda Expressions

Lambda expressions enable many benefits of *functional programming* to Java. Like most OOP languages, Java is built around classes and objects and treats only the classes as their first-class citizens. The other important programming entities, such as functions, take the back seat.

But in functional programming, we can define functions, give them reference variables, and pass them as method arguments and much more. JavaScript is a good example of functional programming where we can pass callback methods to Ajax calls and so on.

Note that we were able to do everything prior to Java 8 using anonymous classes that we can do with lambda expressions, but they use a very concise syntax to achieve the same result. Let us see the comparison of the same method implementation using both techniques.

```
//Using lambda expression
Operator<Integer> addOperation = (a, b) -> a + b;

//Using anonymous class
Operator<Integer> addOperation = new Operator<Integer>() {
  @Override
  public Integer process(Integer a, Integer b) {
    return a + b;
  }
};
```

Lambda expression is a very useful feature and has been lacking in Java from the beginning. Now with Java 8, we can also use functional programming concepts with the help of this.

Happy Learning !!

## Recommended Reading:

1. Java 8 Comparator example with lambda
2. Java 7 – Switch Expressions with Strings
3. Java 14 – Enhanced Switch Expressions
4. Truthy and Falsy Expressions
5. AWS Lambda Function Example
6. Java Predicates
7. Java 8 – Date and Time Examples
8. Java 8 method reference example
9. Java 8 Optionals : Complete Reference
0. Java 8 forEach()

Subscribe

* We do not spam !!

## 18 thoughts on “Lambda Expressions in Java”

### Himanshu

March 12, 2020 at 1:55 pm

Genuinely good article specially the example where you explained how lambda is converted to functional interface with example of Runnable.

Reply

### alexandra

February 29, 2020 at 1:49 am

Thank you! I love your way of explaining things! So happy I found your page

Reply

**vinod**

February 18, 2018 at 12:20 am

i want to append “1” to each element. But its not allowing me any reason ?
i simply replace //do more work with p+”1''; but it is thowing compile error.

```
List<String> pointList = new ArrayList();
pointList.add("1");
pointList.add("2");

pointList.forEach(p ->  {
                            System.out.println(p);
                            //Do more work
                            p+"1";
                        }
                );
```

Reply

**Nitin Agrawal**

March 14, 2018 at 6:25 pm

pointList.forEach(p -> {
p=p+”1'';
System.out.println(p);
}
);

Reply

### Nitin Agrawal

March 14, 2018 at 6:31 pm

```
pointList.forEach(p ->  {
                  p=p+"1";
                              System.out.println(p);
                          }
                    );
```

Reply

### Rameez

January 1, 2020 at 3:45 pm

Basically, when u are trying to mutate a value in list while iteration, it won't work as per desired way. So, the best option is a dummy list and play with it like this.

```
List pointList = new ArrayList();
pointList.add("1");
pointList.add("2");
List newList = new ArrayList();
pointList.forEach(e > newList.add(e+1));
pointList = newList;
System.out.println(pointList);
```

Reply

**Laxminarsaiah Ragi**

January 3, 2018 at 12:00 pm

Awesome boss, excellent.

Reply

**AJ**

May 26, 2016 at 1:57 am

Hi Lokesh, I am following HTDIJ for long time and I must say that current presentation of the topics is best so far, easy to access.

Regards, AJ

Reply

**Lokesh Gupta**

May 26, 2016 at 12:05 pm

Thanks for the feedback. It really helps.

Reply

**m.sandyareddy**

April 21, 2015 at 11:14 am

Nice Article

Reply

**yosama bin badden**

October 7, 2014 at 11:25 am

Can someone explain where the lambdas are used in example #3?

Reply

**Lokesh Gupta**

October 7, 2014 at 2:08 pm

You see how name compare function is used.. That's lambda.

Reply

**Laxminarsaiah Ragi**

January 3, 2018 at 12:03 pm

```
Arrays.sort(employees, Employee::nameCompare);//lamda style
```

Reply

**vivek agarwal**

April 23, 2014 at 6:45 am

Hi Lokesh

Just wanted to know if you can also possibly mention what all new api's have been added in Java 8.
For example Iterable.foreach has been added recently.

Reply

**sattymish**

April 2, 2014 at 1:13 pm

while lambda expressions have definitely added to the language feature i.e. doing away with cryptic codes I still have a feeling that they are mere syntactical changes. With my current knowledge of the features of java8 I believe the real action is in streams …. but that is just my opinion 🙂

Reply

**Lokesh Gupta**

April 2, 2014 at 1:36 pm

You are absolutely right about lambdas. They are basically for making code shorter, cleaner and easy to read (once we are in habit of it). But for providing lambda support, they had to introduce function interfaces, and this is a very much game changer in my opinion. This is going to change a lot the way we write APIs/frameworks today.

Reply

**Ram**

April 1, 2014 at 7:58 am

Really nice, simple and clear post

Reply

**sudeep**

March 27, 2014 at 1:40 pm

Awesome work.. Its make simple and useful.. Keep Going…

Reply

## Leave a Comment

Name *

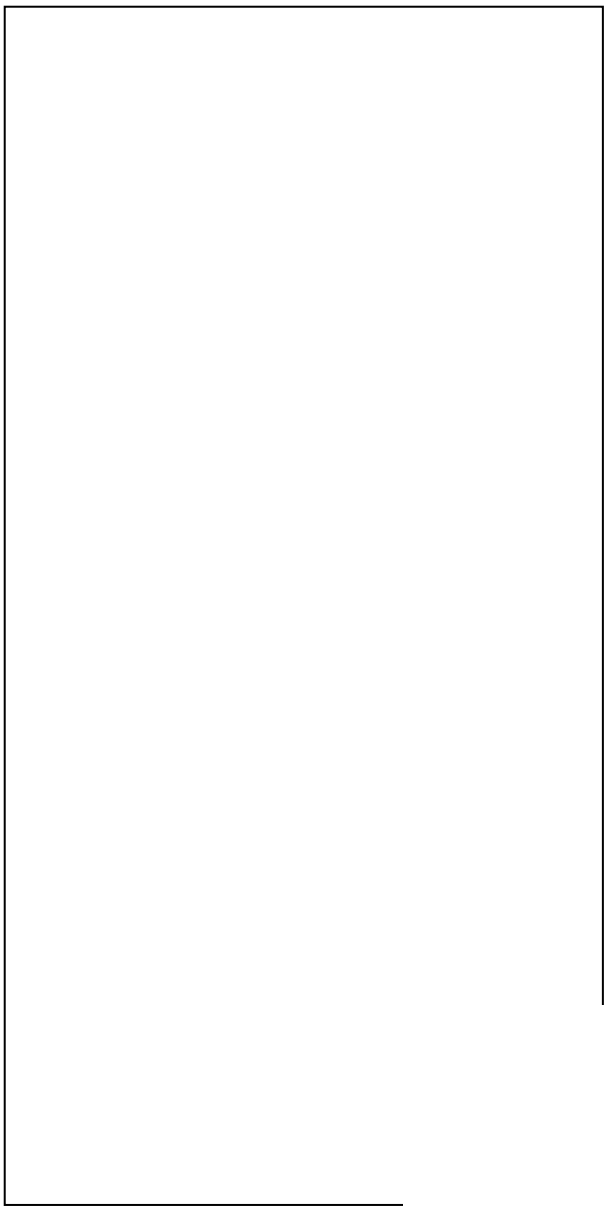
Email *

Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

**Post Comment**

Search …



## HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

**Meta Links**

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

**Blogs**

REST API Tutorial