

Core Java

Lesson 14—Lambda Expression



Learning Objectives

At the end of this lesson, you should be able to:






- ✓ Define Lambda Expression
- ✓ Understand the steps involved in writing default methods
- ✓ Use lambda expression in functional interfaces
- ✓ Understand Method Call using Method Reference (::) vs Lambda Expression (->)
- ✓ Use Lambda Expression in Streams



Topic 1—What is Lambda Expression?

Topic 1—What is Lambda Expression?

Lambda Expression

-  Lambda expression is the biggest add-on to Java 8. You can write functional programming using this.
-  Using this expression, you can build anonymous functions that let you pass methods as arguments in a simple way.
 -  This reduces the usage of boiler plate code.
 -  It won't have any type of visibility or return type, which makes programmers' life easier.
-  It helps you develop more flexible, generic, and reusable APIs.

Lambda Expression (Contd.)



-> symbol is called lambda expression.

Syntax:

```
variable/parameter/object    ->    body    of    method  
(arg1, arg2, arg3, ... . argn) ->{ body of method }
```



In lambda expression, there is no need to mention the type of variable/parameter/object on the left side.



The compiler predicts the type based on the return value of the expression.

Lambda Expression Example



In the following example, line 7 shows how to write the expression inline without specifying the type of arg1. The compiler automatically gets the type of arg1 and works out which is a modular way of programming.

```
1. interface TestLambda
2. {
3.     public void sayHello (String message);
4. }
5. public class Main {
6. public static void main (String [] args) {
7.     TestLambda lambda = arg1->System.out.println("hello" +arg1);
8.     lambda.sayHello("lambda expression");
9.     lambda.sayHello("world");
10.    lambda.sayHello(", its awesome programming with lambda (->)");
11. }
12. }
```

Output:

hello lambda expression

hello world

hello , its awesome programming with lambda (->)

Topic 2—Default Methods

Topic 2—Default Methods

Writing Default Methods



Prior to Java 8, Java versions did not allow programmers to write a body for any method within the interface.



Java 8 version allows programmers to write the definition for default methods within the interface.



This feature allows programmers to add a functionality to the interface where the backward compatibility has been preserved.



When a class implements such an interface, it doesn't have to implement the functionality for the default methods.

If the class wishes to modify the functionality, it can go ahead and override the functionality.



To give default implementation for any method, you need to add default keyword before the method because all the methods in an interface are public by default.

Syntax:

```
interface MyInterface{    default void mymethod(param-list){ //body
of method    } }
```


Writing Default Methods Example



The following example shows how to write the default method in the interface and create an object to call that default method from the interface.

```
interface DefaultDemoInterface
{
    default void defaultMethod() {
        System.out.println("I am default implementation of a method in interface");
    }
}

public class DefaultDemo implements DefaultDemoInterface {
    public static void main (String[] args) {
        DefaultDemo d = new DefaultDemo();
        d.defaultMethod();
    }
}
```





Output:

I am default implementation of a method in interface

Topic 3—Functional Interfaces

Topic 3—Functional Interfaces

Functional Interfaces

-  In functional interfaces, only one abstract method is similar to Comparable interface's *compareTo* method.
-  In Runnable interface, you can see run() method; in Comparator interface, you can see compare (arg1,arg2) method. These interfaces are termed functional interfaces.
-  Java 8 has added many interfaces to improvise the programming and separation in functionality: Consumer<T>, Function<T,R>, Predicate<T>, Supplier<T>, IntToLongFunction, etc.
-  Interfaces with only one abstract method (such as the ones mentioned above) can be easily represented and implemented using lambda expression.

Functional Interfaces (Contd.)

- The annotation `@FunctionalInterface` in Java 8 can be given above the interface to create your own functional interface. The compiler makes sure that you write only one abstract method in an interface.
- Such interfaces are called Single Abstract Method (SAM) interfaces.

Functional Interfaces (Contd.)

Example

The following example shows how annotation implements the functional interface and how the lambda expression gives a body to such a method and makes a call.

```
@FunctionalInterface
interface MyFunctionalInterface
{
    public abstract void doSomeWork();
}

public class FunctionalInterfaceDemo {
    public static void main(String[] args) {
        carryOutTheWork() -> System.out.print("Carry out some work using lambda expression");
    }
    public static void carryOutTheWork(MyFunctionalInterface mi) {
        mi.doSomeWork();
    }
}
```

Output:

Carry out some work using lambda expression

Functional Interfaces (Contd.)



The implementation of Predicate<T> interface, which has the test() method, helps you evaluate the expression that has been passed. The following example of List<String> shows how to use functional interfaces of Java 8 by using lambda expression.

Example

```
public class FunctionalDemos2
{
    public static void main (String[] args {
        List<String> nameList=Arrays.asList("Sachin", "Samuels", "Zaheer", "Dhoni", "Virat", "Shewag");
        System.out.println("Printing all the names starting with S");
        checkString(nameList, (name->name.startsWith("S")));
        System.out.println("Printing all the names ending with I");
        checkString(nameList, (name->name.endsWith("I")));
    }
    public static void checkString(List<String> nameList,Predicate<String> p){
        for(String s:nameList){
            if(p.test(s)){
                System.out.print(s+" ");
            }
        }
        System.out.println();
    }
}
```

Output:

```
Printing all the names starting with S
Sachin Samuels Shewag
Printing all the names ending with I
Dhoni
```

Topic 4—Method Reference

Topic 4—Method Reference

Method Reference



Method reference operator (::) is the shorthand of the lambda expression. It executes one method easily. It is simpler than lambda.

Syntax:

```
Object :: method
```



You cannot use method reference operator for all methods; it can only be used to replace the single method lambda expression.



To use a method reference operator, you need a lambda expression that has implemented one method. To use a lambda expression, you need a functional interface with one abstract method in it.



Using method reference (::), you can call static methods, instance methods of an existing object, and constructors.

Method Reference Example

The following example shows how to invoke a method for object of a particular type using method reference.

```
Import java.util.Arrays;
public class MethodRefDemo
{
    public static void main (String[] args {
        String[] names = { "Barbara", "James", "Mary", "John", "Patricia", "Robert", "Michael", "Linda" };
        System.out.println("Before Sorting");
        System.out.println("Arrays.toString(names));
        // sorting by ignoring case and using method reference to call
        Arrays.sort(names, String :: compareToIgnoreCase);
        System.out.println("After Sorting");
        System.out.println(Arrays.toString(names));
    }
}
```

Output:

Before Sorting

[Barbara, James, Mary, John, Patricia, Robert, Michael, Linda]

After Sorting

[Barbara, James, John, Linda, Mary, Michael, Patricia, Robert]

Method Reference Example (Contd.)

The following example shows you how to invoke static and non-static methods using method reference.

```
Import java.util.function.IntBinaryOperator;

public class MethodRefDemo2
{
    public static void main (String[] args {
        MethodRefDemo2 m2 = new MethodRefDemo2();
        //Calling a static method using reference operator
        m2.operation(MethodRefDemo2 :: multiply, 22, 22);
        //Calling a non static method using reference operator
        m2.operation(m2 ::add, 150, 25);
    }
    public static int multiply(int x, int y) {
        return x * y;
    }
    public int add(int x, int y){
        return x+y;
    }
    public void operation(IntBinaryOperator operator, int a, int b) {
        System.out.println(operator.applyAsInt(a, b));
    }
}
```

Output:

484

175

Method Reference Example—Lambda Vs. Method Reference

```
class Numbers {  
    // using findNumbers() method  
    List<Integer> list = Arrays.asList(12,5,45,18,33,24,40);  
  
    // Using an anonymous class  
    findNumbers(list, new BiPredicate<Integer, Integer>() {  
        public boolean test(Integer i1, Integer i2) {  
            return Numbers.isMoreThanFifty(i1, i2);  
        }  
    });  
  
    // Using a lambda expression  
    findNumbers(list, (i1, i2) -> Numbers.isMoreThanFifty(i1, i2));  
  
    // Using a method reference  
    findNumbers(list, Numbers::isMoreThanFifty);  
}
```

Method Reference Example—Lambda Vs. Method Reference (Contd.)



By using the Lambda expression, you can invoke the method; by using reference operator, you can just refer to that method.



Using method reference (`::`), you can call static methods, instance methods of an existing object, and constructors.

Topic 5—Stream vs. Lambda Expression

Topic 5—Stream vs. Lambda Expression

Stream vs. Lambda Expression



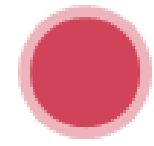
Stream under `java.util.stream` is a feature in java 8 that makes coding simpler.



Streams will categorize the primitives and perform functionalities or logical operations on the categorized primitives.



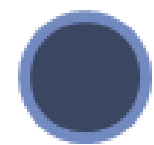
It represents the sequence of values.



Values with stream API allow you to easily perform regular calculations or manipulations of an object.



Stream API provides the simplest way to access data from arrays, files, collections, and various other data sources.



Stream API provides filtering and mapping of the data, especially with collections.

Key Takeaways



- ✓ Lambda expression is the biggest add on to Java 8. You can write functional programming using this.
- ✓ The annotation `@FunctionalInterface` in Java 8 can be given above the interface to create your own functional interface.
- ✓ The implementation of `Predicate<T>` interface, which has the `test()` method, helps you evaluate the expression that has been passed.
- ✓ Method reference operator (`::`) is the shorthand of the lambda expression. It executes one method easily. It is simpler than lambda.
- ✓ You cannot use method reference operator for all methods; it can only be used to replace the single method lambda expression.
- ✓ The stream API provides the simplest way to access data from arrays, files, collections, and various other data sources.



**QUIZ
1**

Which of the following is true about Functional Interface?

- a. It's a interface with no abstract method
- b. It's a interface with SAM (single abstract method)
- c. You can use @FunctionalInterface annotation to create your own functional interface
- d. Functional Interfaces have many abstract methods



QUIZ 1

Which of the following is true about Functional Interface?

- a. It's a interface with no abstract method
- b. It's a interface with SAM (Single Abstract Method)
- c. You can use @FunctionalInterface annotation to create your own functional interface
- d. Functional Interfaces have many abstract methods



The correct answer is **b and c.**

Functional Interface is an interface with single abstract method. The annotation **@FunctionalInterface** was added from java8 to enable users to build their own Functional Interfaces.

QUIZ 2

Lambda expressions will help in:

- a. Reducing boilerplate code
- b. Replacing anonymous classes and interfaces
- c. Writing an expression or body of method without even giving return type
- d. All of the above



**QUIZ
2**

Lambda expressions will help in:

- a. Reducing boilerplate code
- b. Replacing anonymous classes and interfaces
- c. Writing an expression or body of method without even giving return type
- d. All of the above



The correct answer is **d.**

Lambda expressions will help in reducing boilerplate code, replacing anonymous classes and interfaces, and writing an expression or body of method without even giving return type.



Thank You