Java Lambda Expressions

Last Updated : 01 Aug, 2025

Lambda expressions in Java, introduced in Java SE 8. It represents the instances of functional interfaces (interfaces with a single abstract method). They provide a concise way to express instances of single-method interfaces using a block of code.

Key Functionalities of Lambda Expression

Lambda Expressions implement the only abstract function and therefore implement functional interfaces. Lambda expressions are added in Java 8 and provide the following functionalities.

- **Functional Interfaces**: A functional interface is an interface that contains only one abstract method.
- **Code as Data**: Treat functionality as a method argument.
- **Class Independence**: Create functions without defining a class.
- **Pass and Execute**: Pass lambda expressions as objects and execute on demand.

Example: Implementing a Functional Interface with Lambda

The below Java program demonstrates how a lambda expression can be used to implement a user-defined functional interface.

```
interface FuncInterface
{
    // An abstract function
    void abstractFun(int x);


    // A non-abstract (or default) function
    default void normalFun()
```

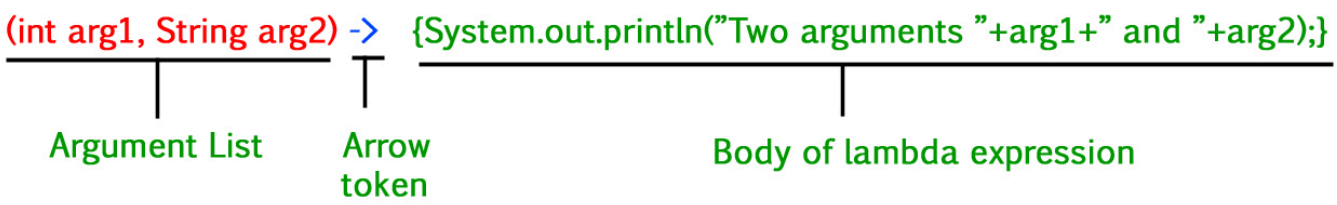
```
    {
        System.out.println("Hello");
    }
}
```

Output

```
10
```

Structure of Lambda Expression

Below diagram demonstrates the structure of Lambda Expression:



Syntax of Lambda Expressions

Java Lambda Expression has the following syntax:

(argument list) -> { body of the expression }

Components:

- **Argument List:** Parameters for the lambda expression
- **Arrow Token (->):** Separates the parameter list and the body
- **Body:** Logic to be executed.

Types of Lambda Parameters

There are three Lambda Expression Parameters are mentioned below:

1. Lambda with Zero Parameter
2. Lambda with Single Parameter
3. Lambda with Multiple Parameters

1. Lambda with Zero Parameter

Syntax:

```
() -> System.out.println("Zero parameter lambda");
```

Example: The below Java program demonstrates a Lambda expression with zero parameter.

```
@FunctionalInterface
interface ZeroParameter {
    void display();
}


public class Geeks {
    public static void main(String[] args)
    {
        // Lambda expression with zero parameters
```

Output

```
This is a zero-parameter lambda expression!
```

2. Lambda with a Single Parameter

Syntax:

```
(p) -> System.out.println("One parameter: " + p);
```

It is not mandatory to use parentheses if the type of that variable can be inferred from the context.

Example: The below Java program demonstrates the use of lambda expression in two different scenarios with an ArrayList.

- We are using lambda expression to iterate through and print all elements of an ArrayList.
- We are using lambda expression with a condition to selectively print even number of elements from an ArrayList.

```
import java.util.ArrayList;


class Test {


    public static void main(String args[])

    {

        // Creating an ArrayList with elements {1, 2, 3, 4}

        ArrayList<Integer> al = new ArrayList<Integer>();

        al.add(1);

        al.add(2);

        al.add(3);

        al.add(4);


        // Using lambda expression to print all elements of al
```

Output

```
Elements of the ArrayList:
1
2
3
4
Even elements of the ArrayList:
2
4
```

Note: In the above example, we are using lambda expression with the **foreach()** method and it internally works with the consumer functional interface. The Consumer interface takes a single paramter and perform an action on it.

3. Lambda Expression with Multiple Parameters

Syntax:

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

Example: The below Java program demonstrates the use of lambda expression to implement functional interface to perform basic arithmetic operations.

```
@FunctionalInterface
interface Functional {

    int operation(int a, int b);

}


public class Test {


    public static void main(String[] args) {
```

Output

```
9
20
```

Note: Lambda expressions are just like functions and they accept parameters just like functions.

Common Built-in Functional Interfaces

- Comparable<T>: int compareTo(T o);
- Comparator<T>: int compare(T o1, T o2);

These are commonly used in sorting and comparisons.

Note: Other commonly used interface include **Predicate<T>,** it is used to test conditions, **Function<T, R>**, it represent a function that take the argument of type T and return a result of type R and **Supplier<T>,** it represent a function that supplies results.

Based on the syntax rules just shown, which of the following is/are NOT valid lambda expressions?

1. () -> {};
2. () -> "geeksforgeeks";
3. () -> { return "geeksforgeeks";};
4. (Integer i) -> {return "geeksforgeeks" + i;}
5. (String s) -> {return "geeksforgeeks";};
6. () -> { return "Hello" }
7. x -> { return x + 1; }
8. (int x, y) -> x + y

Explanation of above lambda expressions:

1. **() -> {}: (valid).** No parameters, no return value (empty body). It is valid
2. **() -> "geeksforgeeks": (valid).** This lambda takes no parameters and returns a String without using a return keyword or braces. This is allowed for

single-expression lambdas.

3. **() -> { return "geeksforgeeks"; }: (valid).** This lambda takes no parameters and returns a value using a code block. Since it uses braces {}, the return keyword is required. It is valid.
4. **(Integer i) -> {return "geeksforgeeks" + i;}: (valid).** This lambda takes one parameter i of type Integer and returns a concatenated string. It uses a code block with return, which is correct. It is valid.
5. **(String s) -> {return "geeksforgeeks";}: (valid).** This lambda takes one parameter s of type String, but does not use it. That's still perfectly valid.
6. **() -> { return "Hello" }: (invalid).** Missing semicolon after the return statement inside the block.
7. **x -> { return x + 1; }: (invalid).** This is missing the type of parameter if used in a context where type inference is not possible
8. **(int x, y) -> x + y: (invalid).** If you specify the type of one parameter, you must specify the type of all parameters.