

# Java



In Java, **lambda expressions** (or lambda functions) were introduced in **Java 8** as a way to enable functional programming, simplify code, and allow more concise implementations of interfaces with a single method (commonly known as **functional interfaces**). A lambda expression represents a block of code that can be passed around and executed later, making it a powerful tool for working with functions as first-class citizens in Java.

## Key Characteristics of Lambda Functions in Java:

- **Concise:** Lambda expressions allow you to write functions in a more compact way without the need for creating anonymous inner classes.
- **Functional Programming:** They enable functional programming features in Java, allowing methods to be treated as first-class objects.
- **Improved Readability:** Lambda expressions reduce boilerplate code, especially when working with collections and streams.

## Syntax of a Lambda Expression

The syntax of a lambda expression consists of three parts:

1. **Parameters:** Input parameters, similar to method parameters.
2. **Arrow token (`->`):** Separates the parameters from the body of the lambda.
3. **Body:** The code or expression that defines what the lambda does.

### Basic Syntax:

```
java
```

```
(parameters) -> expression or (parameters) -> { statements }
```

### Example:

```
java
```

```
(int a, int b) -> a + b
```

This is a lambda expression that takes two integers (`a` and `b`) as parameters and returns their sum.

## Examples of Lambda Expressions

### 1. Simple Lambda with One Parameter:

```
java
```

```
(x) -> x * 2
```

This lambda expression takes one parameter `x` and returns its double.

## 2. Lambda with Multiple Parameters:

```
java
```

```
(a, b) -> a + b
```

Here, two parameters `a` and `b` are passed, and the lambda returns their sum.

## 3. Lambda with No Parameters:

```
java
```

```
() -> System.out.println("Hello, World!")
```

This lambda expression takes no parameters and simply prints "Hello, World!" when executed.

## 4. Lambda with a Block of Statements:

```
java
```

```
(int x, int y) -> {  
    int sum = x + y;  
    return sum;  
}
```

In this case, the lambda has a block of code that sums the two integers and returns the result.

# Functional Interface and Lambda Expressions

Lambda expressions are typically used to implement **functional interfaces**. A functional interface is an interface that has exactly **one abstract method** (also known as a single abstract method or SAM). The `@FunctionalInterface` annotation is used to declare such interfaces explicitly.

## Example of a Functional Interface:

```
java
```

```
@FunctionalInterface  
interface MyFunctionalInterface {  
    void display(String message);  
}
```

A lambda expression can be used to provide the implementation of this functional interface:

java

```
MyFunctionalInterface myFunc = (message) -> System.out.println(message);  
myFunc.display("Hello, Lambda!");
```

In this example, `myFunc` is a lambda that implements the `display()` method of `MyFunctionalInterface`.

## Benefits of Lambda Expressions

1. **Conciseness:** They reduce the amount of boilerplate code by eliminating the need for anonymous classes.
2. **Readability:** Lambda expressions make code more readable, especially in scenarios like filtering or mapping data.
3. **Functional Programming:** They enable functional programming in Java, allowing functions to be passed around as arguments and returned as results.
4. **Parallel Processing:** They work seamlessly with Java Streams, making it easier to perform parallel operations on collections.

## Conclusion

Lambda expressions bring **functional programming** capabilities to Java, allowing more concise and flexible code. They improve **readability**, reduce boilerplate, and work effectively with **functional interfaces**, collections, and streams. By leveraging lambda expressions, Java developers can write more efficient, modular, and expressive code, especially when working with operations like filtering, mapping, and data processing.

```

2
3 @FunctionalInterface
4 interface MyLambda
5 {
6     public void display();
7 }
8
9 public class LamdaDemo
10 {
11     public static void main(String[] args)
12     {
13         MyLambda m =
14             () ->
15             {
16                 System.out.println("Hello World");
17             };
18
19         m.display();
20     }
21 }
22
23
24

```

--> Java mein, Lambda Expression ek concise tareeka hai jo anonymous functions (jo ki bina naam ke functions hote hain) ko represent karne mai help karta hai .

--> Java 8 mein introduce hui, Lambda Expressions functional interfaces ke saath kaam karne ke liye syntax ko simplify karte hain.

--> Lambda expression ka anadar sirf ek hi method allowed hai.

--> Lambda expressions aksar functional interfaces ke saath istemal hote hain jismai sirf ek hi single abstract method hoti hain.

--> Lambda Expression ka basic syntax hota hai (parameters) -> { statements; }.

```
Start Page x LamdaDemo.java x
Source History
5 {
6     public void display();
7 }
8
9
10 class Demo
11 {
12     int temp=10;
13
14     public void method1()
15     {
16         int count=0;
17
18         MyLambda ml=()->{
19             int x=0;
20             System.out.println("Hi");
21             System.out.println("Bye"++temp);
22         };
23
24         ml.display();
25     }
26 }
27
28
29 public class LamdaDemo
30 {
31     public static void main(String[] args)
```

--> Ham lamda expression ka andar koi bhi variable initialize kar sakta hai and uska use bhi kar sakta hai. Ex: `int x;`

--> ham lamda expression kai bhar aur method ka andar bhi variable ko initialize kar sakta hai and uska use bhi kar sakta hai inside expression.

Ex: `int count=0;`

# Upper dia gaya dono method mai, ham variable ka kuch bhi kar sakta hai bas update nahi kar sakta hai, nahi too fir error aayaga.

# Also, agar hmna expression mai variable ko use kar liya hai and then within same method aagar ham use update karta hai too bhi error hoga.

Aisa iseliya hota hai kyuki expression can access only those variables which are final or effectively final.

--> Inside same class, agar ham koi global variable declare karta hai, too use lamda expression ka andar update kiya ja sakta hai.

Ex: `int temp =0;`

```
interface MyLambda
{
    public void display();
}

class UseLambda
{
    public void callLambda(MyLambda ml)
    {
        ml.display();
    }
}


class Demo
{
    public void method1()
    {
        UseLambda ul=new UseLambda();
        ul.callLambda(()->{System.err.println("Hello");});
    }
}

public class Main
{
    public static void main(String[] args)
    {
        Demo d=new Demo();
        d.method1();
    }
}
```

--> Lambda expression  
kaisa implement hota  
hai wo agla kuch slide  
mai btaya gya hai.

## 1. Interface MyLambda:

java


 Copy code

```
interface MyLambda {  
    public void display();  
}
```

Yeh ek functional interface hai jisme ek hi method `display()` hai.

## 2. Class UseLambda:

java

 Copy code


```
class UseLambda {  
    public void callLambda(MyLambda ml) {  
        ml.display();  
    }  
}
```

Is class mein ek method `callLambda` hai jo `MyLambda` type ka object lekar uske `display` method ko call karta hai.



### 3. Class Demo:

java


 Copy code

```
class Demo {  
    public void method1() {  
        UseLambda ul = new UseLambda();  
        ul.callLambda(() -> { System.err.println("Hello"); });  
    }  
}
```

`Demo` class mein ek method `method1` hai jisme `UseLambda` class ka ek object banaya jata hai, aur uske `callLambda` method ko ek lambda expression ke sath call kiya jata hai. Yeh lambda expression "Hello" ko standard error stream mein print karta hai.

### 4. Main class:

java

 Copy code

```
public class Main {  
    public static void main(String[] args) {  
        Demo d = new Demo();  
        d.method1();  
    }  
}
```

`Main` class mein `main` method ek `Demo` class ka object banata hai aur uske `method1` ko call karta hai, jisse lambda expression ka execution hota hai.