Lambda Expressions in Java

Alan Biju
@itsmeambro

Lambda Expressions

Lambda expressions in Java simplify code, making it concise, readable, and efficient. Introduced in Java 8, they enable functional-style programming, especially when working with functional interfaces.

🔹 What is a Lambda Expression?

A lambda expression is an anonymous function that can be used to implement functional interfaces in Java.

✅ *Syntax:*

```
(parameters) -> { expression/body }
```

📌 *Example: Lambda Expression to Add Two Numbers*

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

🔹 Why Use Lambda Expressions?

✓ Reduces boilerplate code
✓ Enhances readability & maintainability
✓ Enables functional programming
✓ Works well with Streams & Collections

🔹 Basic Example: Replacing Anonymous Class

🔸 *Without Lambda (Anonymous Class)*

```
interface Greet {
  void sayHello();
}

public class LambdaExample {
  public static void main(String[] args) {
      Greet greet = new Greet() {
          public void sayHello() {
              System.out.println("Hello, Lambda!");
          }
      };
      greet.sayHello();
  }
}
```

Alan Biju
@itsmeambro

🔸 ***With Lambda Expression***

```
Greet greet = () -> System.out.println("Hello, Lambda!");
greet.sayHello();
```

✅ Less Code, Same Functionality!

🔹 **Lambda Expressions with Functional Interfaces**

✓ Predicate<T> → Represents a condition (boolean-valued function).

```
Predicate<Integer> isEven = num -> num % 2 == 0;
```

✓ Function<T, R> → Transforms data from one type to another.

```
Function<String, Integer> length = str -> str.length();
```

✓ Consumer<T> → Consumes input without returning anything.

```
Consumer<String> print = str -> System.out.println(str);
```

✓ Supplier<T> → Supplies values without taking input.

```
Supplier<Double> randomValue = () -> Math.random();
```

✓ Comparator<T> → Compares two values for sorting.

```
Comparator<Integer> compare = (a, b) -> a - b;
```

🔹 **Lambda Expressions in Java Collections**

✅ Using Predicate<T> for Filtering in Streams

```
List<Integer> evenNumbers = numbers.stream()
                               .filter(n -> n % 2 == 0)
                               .collect(Collectors.toList());
```

✅ Using Function<T, R> for Transformation

```
List<Integer> nameLengths = names.stream()
                             .map(name -> name.length())
                             .collect(Collectors.toList());
```

✅ Using Consumer<T> for Iteration

```
names.forEach(name -> System.out.println(name));
```

✅ Using Comparator<T> for Sorting

```
Collections.sort(numbers, (a, b) -> a - b);
```

✅ Using BinaryOperator<T> for Reduction (Sum Calculation)

```
int sum = numbers.stream()
             .reduce(0, (a, b) -> a + b);
```

Alan Biju
@itsmeambro

🔹 Method References (::) - Shorter Lambda

📌 ***Instead of:***

```
  list.forEach(name -> System.out.println(name));
```

We can use:

```
  list.forEach(System.out::println);
```

🚀 Summary

✔ Lambda expressions make Java code concise & functional.
✔ Functional interfaces like Predicate<T>, Function<T, R>, Consumer<T>, and Supplier<T> simplify coding.
✔ Work seamlessly with Java Streams & Collections.
✔ Improve performance & readability in modern Java applications.

Alan Biju
@itsmeambro

If you find this helpful, like and share it with your friends

Alan Biju
@itsmeambro