**Functional Programming**

**─**

Contents

[**1. Introduction to Functional Programming** 2](#_Toc183580337)

[# Pure Functions 2](#_Toc183580338)

[**2. Functional Interface** 4](#_Toc183580339)

[**3. Lambda Expression** 5](#_Toc183580340)

[# Syntax of lambda function 6](#_Toc183580341)

[**4. Interview Questions** 8](#_Toc183580342)

## **1. Introduction to Functional Programming**

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.

Key Principles:

* **Immutability:** You focus on the transformation of data without directly modifying it. New data is produced instead.
* **First-Class Functions:** Functions are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments, or returned from other functions.
* **Declarative Style:** Focus on "what to do" rather than "how to do it." (Just like languages like SQL)
* **Side-Effect Free:** Functions should not modify any state or interact with the outside world.

Real World Example: -

Imagine you're cooking a recipe. In traditional cooking (like traditional programming), you follow a step-by-step process (making it imperative approach): get ingredients, cut the vegetables(modifying their state) , mix them, cook them, and serve.

**Functional programming** is like a different approach to cooking. Instead of following a strict recipe, you focus on the ingredients and the final dish (declarative approach). You prepare all the ingredients beforehand. You might use pre-made sauces, ready-to-eat vegetables, or even a microwave to speed things up.

Benefits: -

* Clean Code: Reduces boilerplate code by leveraging lambda expressions and streams.
* Immutability: Ensures thread safety by avoiding shared mutable state.
* Parallelism: Simplifies parallel operations using parallel streams.

### # Pure Functions

A pure function is a function where the output depends solely on the input parameters and has no side effects (e.g., no modification of global variables or I/O operations).

Characteristics of Pure Functions:

* Deterministic: Same input always produces the same output.
* No Side Effects: Does not alter the state or perform external interactions.

public class PureFunctionExample {

// Pure function  
 public static int add(int a, int b) {  
 return a + b; // Depends only on input  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println(*add*(5, 3)); // Output: 8  
 }  
}

Why Pure Functions Matter:-

* Immutability: Helps maintain an unchangeable state.
* Thread Safety: Pure functions are inherently thread-safe as they don't modify shared data.
* Ease of Testing: Pure functions are predictable and easier to test.

## **2. Functional Interface**

Abstract method

An abstract method is a method that is declared without an implementation. It allows defining behaviors without specifying how they are performed.

Sometimes we may come across situation where we cannot provide implementation to methods in a class, but want to leave implementation to class that extends it. There we use abstract method.

Functional Interface

It is an interface that contains exactly one abstract method. Thus they can have only one functionality to exhibit.

**Key Points:**

* Introduced in Java 8 to facilitate functional programming.
* Lambda expressions and method references rely on functional interfaces.
* Though they may have multiple default and static methods.
* Represented with annotation @FunctionalInterface(acts as compile-time check)

@FunctionalInterface  
interface Calculator {  
 int compute(int a, int b); // Single abstract method  
}  
  
class TempDoubleCalc implements Calculator{  
 public int compute(int a, int b){  
 return 2\*(a+b);  
 }  
}  
  
public class FunctionalInterfaceExample {  
 public static void main(String[] args) {  
 TempDoubleCalc temp = new TempDoubleCalc();  
 System.*out*.println(temp.compute(2,3)); //10  
 }  
}

## **3. Lambda Expression**

A lambda expression is a concise way to represent an anonymous function. It provides an implementation of a functional interface’s single abstract method directly, without using an anonymous class.

Why Lambda Expressions?

* Conciseness: Reduce boilerplate code.
* Readability: Improves code clarity for simple operations.
* Functional Programming: Enables functional style programming in Java by working seamlessly with functional interfaces.

Ex, in previous example of functional interface, to use Calculator interface we first had to create it’s class and then use, that either could be done by anonymous class or better with lambda function

@FunctionalInterface  
interface Calculator {  
 int compute(int a, int b); // Single abstract method  
}  
  
public class FunctionalInterfaceExample {  
 public static void main(String[] args) {  
  
 //With anonymous class  
 Calculator tempC = new Calculator() {  
 @Override  
 public int compute(int a, int b) {  
 return 2\*(a+b);  
 }  
 };  
 System.*out*.println(tempC.compute(2,3)); //10

//With lambda function  
 Calculator TempDoubleCalc = (a,b)-> 2\*(a+b) ;  
 System.*out*.println(TempDoubleCalc.compute(2,3)); //10  
 }  
}

Lambdas are more concise and can only be used for single-method interfaces, while anonymous classes are verbose and can implement multiple methods.

### # Syntax of lambda function

(parameters) -> expression/body

* Parameters: The input arguments (can be zero or more).
* Arrow Token (->): Separates parameters from the body.
* Body: The code to be executed, which can be:
  + A single expression.
  + A block of code enclosed in {}.

1. Lambda with single parameter

name -> System.out.println(name);

1. Lambda with multiple parameter

(a, b) -> a + b;

1. Lambda with no parameter

() -> System.out.println("No parameters!");

1. Lambda with a Single Statement (No Braces Required):

x -> x \* x;

1. Lambda with a Block of Code

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

When and where to use Lambda Expressions?

* Use lambdas for simple, short-lived tasks, such as filtering a collection or setting up an event handler.
* Combine lambdas with functional interfaces like Predicate, Consumer, and Function.
* Use lambdas when passing behavior as arguments.
* Avoid lambdas for **complex logic** requiring multiple lines of code.

## **4. Interview Questions**

**1. Can lambda expressions be used with interfaces that have more than one abstract method? Why or why not?**

No, lambda expressions cannot be used with interfaces that have more than one abstract method because they rely on functional interfaces, which are interfaces with exactly one abstract method. This single abstract method provides the functional target for the lambda expression, enabling Java to determine the intended behavior.

If an interface has multiple abstract methods, there is ambiguity about which method the lambda should implement, violating the principle of functional programming. To ensure compatibility, Java introduced the @FunctionalInterface annotation, which enforces that an interface has only one abstract method, making it suitable for use with lambda expressions.

**1. When should lambdas be avoided?**

* Avoid lambdas for complex logic requiring multiple lines of code.
* Opt for named methods when logic is reused or when lambdas become hard to read.