

What is Functional Programming?

Functional Programming is a programming style where we write programs using functions without changing data.

Java 8 introduced Functional Programming features to make Java:

shorter

cleaner

faster

modern

Functional interface :

A functional interface contains exactly one abstract method, but can have any number of default or static methods. we need Functional interface Because lambda expressions work only with functional interfaces.

@FunctionallInterface annotation is optional but recommended.

predefined Functional interface :

Runnable interface -> run() used in Thread.

Comparable interface -> compareTo(Object o) used in Collection

Comparator interface -> compare(Object o1 , Object o2) used in Collection.

Anonymous class :

An anonymous class is a class without a name that you define and create (instantiate) in one step.

Anonymous classes are useful when you need a class only once and don't want to create a separate

named class file—for example:

to override a method of an existing class

to implement an interface

Lambda Expressions :

Lambda expression is an anonymous function that provides implementation of a functional interface.

A lambda expression is a short way to write a method without method name, without class, and without boilerplate code.

Syntax of Lambda Expression :

(parameters) -> expression

OR

(parameters) -> {

// multiple statements

}

eg:

```
I1 ref1 = ()-> System.out.println("Hello World");
```

```
I2 ref2 = (a,b)-> a+b;
```

forEach() :

The forEach() method is used to iterate through elements of a collection or stream and perform an action on each element.

Syntax :

```
collection.forEach(element -> {
```

// action

});

Method References :

Method reference is just a shortcut for a lambda that only calls an existing method.

If a lambda expression contains only one statement and that statement simply calls an existing method, then a method reference can be used instead of the lambda.

Method reference works only when lambda directly calls ONE existing method.

Syntax:

ClassName::methodName

eg:

```
I2 ref1 = (a,b)-> a+b;  
System.out.println(ref1.add(10, 20));  
  
*instead of this we can sum Integer.sum() method, means we  
can call only method.*  
  
I2 ref2 = Integer::sum;  
System.out.println(ref2.add(10, 23));
```

```
import java.util.Arrays;  
import java.util.List;
```

```
@FunctionalInterface  
interface FI2 {  
    int add(int a, int b);  
}
```

```
public class P11 {  
    public static void main(String[] args) {
```

```

Fl2 obj1 = (a,b) -> a+b;
System.out.println("Using lambda exp :" + obj1.add(20, 30));

Fl2 obj2 = Integer :: sum;
System.out.println("Using method reference : " + obj2.add(10,
20));

List<String> names =
Arrays.asList("Kabir","Devansh","Girish","Nitish","Murali");

System.out.println("\nNAMES: " + names);
System.out.println("using lambda: ");
names.forEach(name -> System.out.println(name));
System.out.println("using method reference: ");
names.forEach(System.out::println);

}
}

```

Stream :

Stream is a Java 8 feature used to process collection data in a functional way without using loops.

- Stream was introduced in Java 8
- Stream does not store data
- It works on Collection data
- Stream processes data one by one
- Stream does not change original collection

Basic Stream Flow : Collection → Stream → Operation → Result

eg: List → stream() → filter() → forEach()

Basic methods of stream :

stream() → Converts a collection into a stream so we can process data in a functional way.

filter() → Selects elements from the stream based on a condition.

map() → Transforms each element in the stream into another form.

forEach() → Performs an action on each element of the stream (commonly used for printing).

collect() → Converts the processed stream back into a collection or another result.

sorted() → Sorts stream elements using natural ordering (uses Comparable).

sorted(Comparator.reverseOrder) → Sorts stream elements using custom sorting logic (uses Comparator).