What is functional programming?

Functional programming is all about maintaining immutable state in a function, returning the same output for the same input. A function is treated as a first class citizen and is free from side-effects.

* Code in declarative style
* Solves the concurrency issues
* Simple, readable and Testable
* Lazy evaluation
* Inside java method (No assignment, No external, same result for an Input)

Why functional programming?

Functional programming avoids the side-effects of functions by maintaining the immutability; it allows coding less as we avoid the boiler plate coding.

Why Object oriented programming?

It Model the complex domains represent real world, where the state is variable and behavior is method.

How to work with Object oriented and functional programming together?

Pure functional programming language make variable immutable, allows to create a new copy of object and set the updated state instead of changing the variable state, avoid the change of state is to avoid concurrency issues (that is what side effects).

How functional programming complements the BigData?

Since functions are independent and side effect free we can go ahead and make execute them in multiple systems. Functional programming having the capacity to process continuous streams, this is the reason functional programming language chosen for BigData projects and web analytics and reducing the concurrency issues.

Problem with synchrozied() in java threads?

Other threads has to wait to get the access.

What is a Lambda Expression?

A lambda expression represents an anonymous function. It comprises of a set of parameters, a lambda operator (->) and a function body.

## Lambda Functions: Syntax

A lambda function consists of optional parameters, the arrow token, and the body:

*LambdaParameters -> LambdaBody*

LambdaParameters are parameters to the lambda function passed within opening parenthesis "(" and closing parenthesis ")". When more than one parameter is passed, they are separated by commas.

To support lambdas, Java has introduced a new operator “->”, also known as lambda operator or arrow operator. This arrow operator is required because we need to syntactically separate the parameter from the body.

LambdaBody can be an expression or a block. The body could consist of single statement (in that case no explicit curly braces defining a block are required); such a lambda body is known as  an "expression lambda." If there are many statements in a lambda body, they need to be in a block of code; such a lambda body is known as "block lambda."

**The compiler performs type inference for lambda expressions:**

The compiler infers the types of the parameters if you do not specify the parameter types in a lambda function definition. When you specify the type of parameters, you need to either specify all or none, or else you will get a compiler error.

You can omit the parenthesis if there is only one parameter. But in this case, you cannot provide the type explicitly. we should leave it to the compiler to infer the type of that single parameter.

The return type of the lambda function is inferred from the body. If any of the code in the lambda returns a value, then all the paths should return a value, or else you will get a compiler error.

Some examples of valid lambda expressions (assuming that relevant functional interfaces are available):

(int x) -> x + x

x -> x % x

() -> 7

(int arg1, int arg2) -> (arg1 + arg2) / (arg1 – arg2)

## Lambda Function—An Example

## In this program, the interface LambdaFunction declares an abstract method named call(); hence it is a functional interface. Inside the main method in the FirstLambda class, a lambda function is assigned to a variable of the functional interface type LambdaFunction.

LambdaFunction lambdaFunction = () -> System.out.println("Hello world");

Here, the expression () -> System.out.println("Hello world") is a lambda expression:

* The syntax () indicates no parameters.
* The arrow operator "->" separates method parameters from the lambda body.
* The statement System.out.println("Hello world") is the body of the lambda expression.

How does the lambda expression relate to the functional interface LambdaFunction? It is through the single abstract method inside the LambdaFunction interface: void call(). **The signature of this abstract method and the lambda expression must match:**

* The lambda expression has () indicating it has no parameters—it matches with the call method that takes no parameters.
* The statement System.out.println("Hello world") is the body of the lambda expression. This body serves as an implementation of the lambda function.
* There is no return statement in this lambda expression body and hence the compiler infers the return type of this expression as the void type—that matches with the return type of the call method.

The next statement lambdaFunction.call(); invokes the lambda function. As a result of this function call, “Hello world” is printed on the console.