1. Introduction to functional Programing

a. Why use Functional Programming

OOP and Functional Programming:

Both are not opposite. They can run in parallel to help the write code in better way.

Where does OOPS falls?

When we write large number of codes, they make more bugs when it comes to reproduction of bugs. It creates more issue.

Most of the times codes are written in multiple services and it becomes more buggy state.

Functional Programming helps

When we use functional programming, this helps to eradicate the issue.

Functional Programming help to take the many individual component and organizes them to work in coherent way so that the code remains easy to test and modify.

Brings the precision of math into programming

$f(x) = x + 1$

b. What is declarative Programming

Two Styles of Programming:

- Declarative – What is it?
- Imperative – How do we get it?

Example:

House:

Declarative – What does a house consist of?

Imperative – What are the instruction to build a house?

Average of array:

Imperative: How?	Declarative: What
1. Set x to 0	X is the sum of all the number in the array,
2. Add first number to x	Divided by the length of the array
3. Repeat step 3 for all elements	
4. Divide x by length of array	

$X = \sum x/n$

Core Functionality of Functional Programming:

- Immutability Functional Purity First Class Function

c. Immutability in Functional Programming

Immutability means we should treat most of the variable as constant so that it could not change its value.

int x = 3 ➔ x is 3

Example:

Update a salary

- OOP way
  Employee employee1 = new Employee("Ishwar",60000);
  Employee1.setSalary(7000);
- Functional Programming Way
  Employee employee1 = new Employee("Ishwar",6000);
  Employee updatedEmployee =new Employee("Ishwar",7000);

Functional Programming freezes the state.

We can treat individual component rather than focusing on entire state change.

d. Functional Purity

Always return the same output as given input.

Impure Function: Have an internal or external state change.

```
public class Person {
        private int age;
        public int getAge(){ return age};
        }


person.getAge();        //30
person.setAge(40);
person.getAge();        //40
```

Pure Function Example:

```
int add(int a , int b) {
        return a + b;
}
```

Does that means we can't use member variables?

Example:

```
public class Person {
        private String name;
        private int age;
        public String toString(){
                return “Name :”+ name+ “Age :”+age;
        }
```

If we don’t write any setter method, it means we are not changing the variable. Then the function is the pure function but if we write setter method then it’s an impure function.

None of our functions should refer to any kind of state; they should all be pure.

e. First class function

In OOP function and variable are treated as different.

We can never treat array of functions, take function as argument, return function as argument.

But in Functional Way: We can do all these with the help of Function<T,R> interface.

2. First Class Functions

a. The Functional Interface

We must treat function same as string, integer etc.

In Functional Programming, we are allowed to assign functions to variables.

JAVA Functional Interface allow to use function as a variable.

b. Lambda Expression

Besides creating references to other class’s methods, the function interface allows us to create new functions from scratch.

This we can do with the help of lambda expression.

Lambda expression is a shortened syntax to define new functions without defining these methods as a function to any given class.

Example:

```
Function<Integer, Integer> myFunction = (Integer someArgs) -> someArgs*2+1;
```

if one line no need to put return type.

The type can be gussed with the help of type of function

```
Function<Integer, Integer> myFunction = (someArgs) -> someArgs*2+1;
```

Function<String, Integer> stringLength = (inputString)-> inputString.length();

If only one argument we are allowed to drop the parenthesis.

Function<String, Integer> stringLength = inputString -> inputString.length();

c. BiFunction and beyond

Can we use the Function interface to define functions with different numbers of arguments?

BiFunction<T, U, R>

We can also define our own function with custom input parameters and even No argument function.

d. Functions as data

Assign the definition of function dynamically at runtime.

e. Passing function as arguments.

We can pass function as an argument to the other function.

f. Returning function.

A function is always considered as a black box. In same way it is treated as function creator or function factory.

g. Closure

Closure means when we define a function that return other function, that returned function still has the access to the internal scope of the function that returned it.

h. High Order Function

Function returning function

Example : Divide by zero () -> () ->

3. Working with Streams in JAVA

a. Map in JAVA

Map is used when we have a list of data and we want to convert each of the individual elements in the list to some other form.

Example:

double all the numbers in the list

convert a list of inch measurements into a list of centimeter measurements

Procedural way:

```
for (int i =0;i< list.size-1;i++){
        int result = list.get(i)*2;  newlist.add(result);       }
```

Functional Way:

Function<Integer,Integer> doubles = (x) -> x*2;

list.stream.map(doubles).collect(Collectors.toList());

Benefit: It does not mutate the original list. It create one more pipeline of the result.

b. Filters in JAVA

If we want to find all numbers based on some criteria.

Example:

Number are even

List of employee salary > some amount.

For filter we pass Boolean value.

Predicate<T> is a function which return Boolean

Predicate<Integer> isEven = (x) -> x%2==0;

List<Integer> evenNumbers = list.stream.filter(isEven).collect(Collectors.toList());

```
Function<Integer, Predicate<String>> createLengthTest = (minLength) -> {
        return (str) -> str.length() > minLength;
    };
Predicate<String> longerThan3 = createLengthTest.apply(3);
```

c. reduce in JAVA

SYNTAX : .reduce(sum) => BinaryOperator<T>

Accumulator (acc)

(acc,x) -> acc+x

.reduce(startingValue,func)

reduce(func) -> Optional<T>

.reduce(startingValue,func) -> T

d. collect in JAVA

It is similar to reduce method. it is flexible it can return any type.

List<Integer> numbers.stream.reduce(….) -> must return an integer

List<Integer> numbers.stream.collect(….) ->can return any type

myList.stream.collect(myCollector) → Collector< T,A,R>

Collectors.toList

Collectors.toSet

Collector.joining(“, ”)

Collectors.counting

Collectors.groupingBy

Collectors.partioningBy

e. Parallel Stream

Parallel stream process the data in parallel

Increase performance

4.Advance Concept

a.Partial Application

partial application is when we take a function that has some number of arguments, and we fix some of those arguments to a set value. This function with fixed arguments can then be called from anywhere else in the code, and it will be as if we had called the original function with all of its arguments.

Partial Application is used to configure more general functions into specific functions.