Functional Programming – 2

There are different ways, in which we can construct our webpage or project; some sort of pattern is to be followed. In functional programming, everything is achieved using a simpler and concise functions. There are generally 4 pillars or principles that come into object-oriented programming. But functional programming has different set of principles.

Principles:

1. Pure Functions
2. Function Composition
3. Avoiding Shared State
4. Avoid Mutating State
5. Avoid Side Effects
6. Referential Transparency
7. Higher Order Functions
8. First Class Functions

The last 3 are supported by the language itself.

Pure Function:

These functions have two main properties. First, they always produce the same output for same arguments irrespective of anything else.

Secondly, they have no side-effects i.e. they do modify any argument or global variables or output something.

Later property is called immutability. The pure functions only result is the value it returns. They are deterministic.

Programs done using functional programming are easy to debug because pure functions have no side effect or hidden I/O. Pure functions also make it easier to write parallel/concurrent applications.

When the code is written in this style, a smart compiler can do many things – it can parallelize the instructions, wait to evaluate results when need them, and memorize the results since the results never change as long as the input doesn’t change.

Ex: Impure Function

```javascript
// Impure Function

const name = 'Alice'
const sayHi = () => {
    console.log('hi, I am ${name}')
}

sayHi()
```

So, the pure function way of doing the same thing is,

```javascript
// Impure Function

const name = 'Alice'
const sayHi = () => {
    console.log(`hi, I am ${name}`)
}

sayHi()


// Pure Function
const sayHello = (nameStr) => {
    console.log(`Hi, I am ${nameStr}`)
}

sayHello(name)
```

Pure function depends on what you passed inside the function.

Ex: 02

```javascript
const PI = 3.14
const radius = 12

// impure
const area = (parameterRadius) => {
    console.log(PI * parameterRadius * parameterRadius)
}

area(radius)
```

This is an impure function, because it is dependent on the external value and not dep endent on a value that is defined inside the function. To make it into a pure function,

```
const calArea = (paramRadius, pi) => {
    console.log(pi * paramRadius * paramRadius)
}

calArea(radius, 3.14)
```

PI value is reoccuring. So, the precision is not being defined. but in pure function, we are passing the value of PI and accordingly the calArea() will give us the result.

Avoid Mutating State

```
const person = {name: 'Alice', age: 12}
console.log(person)

const mutateObj = (personObj) => {
    personObj.name = "Yash"
    personObj.age = 10
}

mutateObj(person)
```

Everything is passed inside the function. The mutateObj() is changing the values in the person object. Since in pure function values are not modified or changed, this does not come under impure function.

To make it into a pure function,

```
const pureMutate = (personObj) => {
    const newPerson = {...personObj}
    newPerson.name = 'Yash'
    newPerson.age = 10
    return newPerson
}

const nowMutateObj = pureMutate(person)
console.log(person)
console.log(pureMutate)
```

The first log gives the information that we defined first and then the second log, it is showing the new values that we have changed in pureMutate() function. The property values are not changed directly on the external object, but it is creating a new object which is passed as a parameter and in that object the values are changed and returning the values.

Avoid Side Effects:

```
let areaVolume = [0, 0] //12, 23

const area = (length, breadth) => {
    const ar = length * breadth;

    areaVolume[0] = ar

}

const volume = (length, breadth, height) => {
    const vol = length * breadth * height;

    areaVolume[1] = vol
}

function displayAreaAndVolume (arVol) {
    console.log(`Area is ${arVol[0]} & Volume is ${arVol[1]}`)
}

area(10,10)
volume(10,10,10)
displayAreaAndVolume(areaVolume)
```

The area function is impure. We can return a new array, but we are avoiding the mutation property. Same way in the volume as well, we are avoiding the mutation property.

We have changed the first index with the second index value. So, areaVolume[1] = ar and areaVolume[0] = vol. Doing this, we are creating a side effect which we are not sure about.

But if we return the area and volume from the respective functions, and in the global element, if we write,

```
let areaVolume = [0, 0] //12, 23

const area = (length, breadth) => {
    const ar = length * breadth;

    return ar

}
```

```
const volume = (length, breadth, height) => {
    const vol = length * breadth * height;

    return vol
}

function displayAreaAndVolume (arVol) {
    console.log(`Area is ${arVol[0]} & Volume is ${arVol[1]}`)
}

areaVolume[0] = area(10,10)
areaVolume[1] = volume(10,10,10)
displayAreaAndVolume(areaVolume)
```

Avoiding Shared States:

```
const PI = 3.14
const radius = 12

const areaCircle = (parameterRadius) => {
    console.log(PI * parameterRadius * parameterRadius)
}

const volSphere = (passedRadius) => {
    console.log(PI * (passedRadius**3) * (4/3))
}
```

We are using the same external values and the two functions are sharing the values. So, if we change the value of one variable, it would change the result of both the functions. So to do it a pure way,

```
const calArea = (parameterRadius, pi) => {
    console.log(pi * parameterRadius * parameterRadius)
}

const volSphere = (passedRadius, pi) => {
    console.log(pi * (passedRadius**3) * (4/3))
}
```

“How to become a developer”

We want to change the font to lowercase. One way to achieve this is,

```
const str = "How to become a developer"

const toSlug = (string ="") => {
    const result = string.split(' ')
    console.log(result)
}

toSlug(str)
```

When we call this function, the string.split will split all the words based on the space so the output would be an array with all the words. Since this entire thing is returing an array, I’ll be able to map over it.

```
const str = "How to become a developer"

const toSlug = (string ="") => {
    const result = string.split(' ').map(item => item.toLocaleLowerCase
())
    console.log(result)
}

toSlug(str)
```

Split returns a new array. Split is a pure function, so it will not make any changes. So we will map over that new array and covert everything into a lowercase words. And finally, we need to join everything. So,

```
const str = "How to become a developer"

const toSlug = (string ="") => {
    const result = string.split(' ').map(item => item.toLocaleLowerCase
()).join('-')
    console.log(result)
}

toSlug(str)
```

We are able to convert the string into a lowercase. In function composition we give an input.

$G(x) = x + 3$

F(y) = y – 2

The same logic applies here as well.

```
toSlug(str)

const toNewSlug = () => {
    join('-')(
        map(toLowerCase)(
            split(str)
        )
    )
}
```

This will not work, but it is only to get an idea of how f(g(x)) applies to our javascript here.

Function Composition

Ex: 03

```
function greeter(name) {
    return () => {
        console.log(`Hi my name is ${name}`)
    }
}

const yashGreeter = greeterMaker('Yash')

const akashGreeter = greeterMaker('Akash')
```

in the output we get a function. Now, basically greeterMaker() is composing functions internally. So, if we call yashGreeter() and akashGreeter(), then the output is

Hi my name is Yash

Hi my name is Akash

We have created a function based on a parameter. It is also making use of closures also. The function internally knows