**Functional Programming**

**What is state in programming?**The remembered information is called **state** of program.  
eg:-   
  
var name = “mohan”;   
var count = 20;

**What is shared state?**  
Global variable or object shared by multiple function.   
  
Good practise is don’t share data(state) between function pass data(state) between the functions.

**What is pure function**function receive the input and perform the action on received input and return the result. With out modifying the global value or data.   
  
function add(a, b){  
 return a + b;  
}  
  
**Avoid data mutable**const num = 20;  
num = 30; // error Assignment to constant variable

But not true for object

const myarr = [5,3,4,2,6];  
console.log(myarr); //[5, 3, 4, 2, 6]  
myarr.sort();   
console.log(myarr) //[2, 3, 4, 5, 6]

but here “const” array got sorted because we are not reassigning any new value to myarr we are modifying the existing object ie objects in javascript are mutable.

**How to avoid object to be mutable?**use **Object.freez(myarr);**

const myarr =[5,3,4,2,6]  
var sortarr = function(val){

return val.sort();

}

console.log(sortarr(myarr)); //[2, 3, 4, 5, 6]

console.log(myarr); //[2, 3, 4, 5, 6]

**Original array got sorted here so object are not passed by value, Object are passed by reference.**

Best way to avoid object mutation is to clone object.   
  
**Method1:**- clone object using **Object.assign({}, obj1)**  
  
const obj = {

name:"mohan",  
height:6,  
address:{  
 state:"kar",  
 country:"ind"  
}

}

var newobj = Object.assign({}, obj);  
obj.height = 6.2;  
obj.address.state = "andhra";

console.log(obj); // {"name":"mohan","height":6.2,"address":{"state":"andhra","country":"ind"}}  
console.log(newobj); //{"name":"mohan","height":6,"address":{"state":"andhra","country":"ind"}}  
  
for newobj ie cloned object height property is not changed but **inner object called state got changed, so this is not deep cloning, its just shallow cloning.**

**Method 2**:- Deep cloning using **JSON.parse(JSON.stringify(obj));**

var newobj = JSON.parse(JSON.stringify(obj))  
obj.height = 6.2;  
obj.address.state = "andhra";

console.log(JSON.stringify(obj));   
// {"name":"mohan","height":6.2,"address":{"state":"andhra","country":"ind"}}

console.log(JSON.stringify(newobj)); //{"name":"mohan","height":6,"address":{"state":"kar","country":"ind"}}

This is one of the methods for deep cloning, JSON.stringify() convert object to json string, later JSON.parse() convert string to new object in memory, so after parse and stringify it breaks the reference of given object and create new object in memory. Same apply for Array as well.

For cloning create a pure function, call this when it required.

var cloneObj = function(obj){  
 return JSON.parse(JSON.stringify(obj));  
}

Best practise

1. Function should receive input
2. Function should return a value
3. Function should perform only single task

Arrow functions

var sum = function(num1, num2){  
 return num1 + num2;  
}

//passing 2 parameter and return a value  
var sum = (num1, num2) => num1+num2;

//pass single parameter and return a value  
var fan = num => num\*num;

//no parameter but return a value  
var fan = () => 100

**Pipe and Composed**

**Compose function example**

var str = 'Innovation distinguishes between a leader and a follower.';

const trim = str => str.replace(/^\s\*|\s\*$/g, '');

const noPunct = str => str.replace(/[?.,!]/g,'');

const capitalize = str => str.toUpperCase();

const breakout = str => str.split(" ");

const noArticles = str => (str !== "A" && str !== "AN" && str !== "THE");

const filterArticles = arr => arr.filter(noArticles);

var compose = function(...arg){

return function(x){

return arg.reduceRight(function(val1,val2){

//val1 gives value and val2 holds passed fucntions

console.log(val1+" "+val2)

return val2(val1)

},x)

}

}

const result = compose(

filterArticles,

breakout,

capitalize,

noPunct,

trim

)

console.log(result(str)); //["INNOVATION", "DISTINGUISHES", "BETWEEN", "LEADER", "AND", "FOLLOWER"]  
  
  
**pipe function example**

var str = 'Innovation distinguishes between a leader and a follower.';

const trim = str => str.replace(/^\s\*|\s\*$/g, '');

const noPunct = str => str.replace(/[?.,!]/g,'');

const capitalize = str => str.toUpperCase();

const breakout = str => str.split(" ");

const noArticles = str => (str !== "A" && str !== "AN" && str !== "THE");

const filterArticles = arr => arr.filter(noArticles);

var compose = function(...arg){

return function(x){

return arg.reduce(function(val1,val2){

//val1 gives value and val2 holds passed functions

console.log(val1+" "+val2)

return val2(val1)

},x)

}

}

const result = compose(

trim,

noPunct,

capitalize,

breakout,

filterArticles

)

console.log(result(str)); // ["INNOVATION", "DISTINGUISHES", "BETWEEN", "LEADER", "AND", "FOLLOWER"]

**Arity**

const users = [{name: "James",score: 30,tries: 1}, {name: "Mary", score: 110,tries: 4}, {name: "Henry",score: 80,tries: 3}];

//Modifies Data

var storeUser = function(arr, user) {

return arr.map(function(val) {

if (val.name.toLowerCase() === user.name.toLowerCase()) {

return user;

} else {

return val;

}

});

};

//Pure Functions

const cloneObj = function(obj) {

return JSON.parse(JSON.stringify(obj));

};

var getUser = function(arr, name) {

return arr.reduce(function(obj, val) {

if (val.name.toLowerCase() === name.toLowerCase()) {

return val;

}

}, null);

};

var updateScore = function(newAmt, user) {

if (user) {

user.score += newAmt;

return user;

}

};

var updateTries = function(user) {

if (user) {

user.tries++;

return user;

}

};

//bind() method binds the value to the function and this value will be permeant value, below example updateScore() function first parameter is 30 for any call. “this” value for bind() is null.

const partGetUser = getUser.bind(null, users);

const partUpdateScore30 = updateScore.bind(null, 30);

/\*const usr = getUser(users, "Henry");

const usr1 = updateScore(cloneObj(usr), 30);

const usr2 = updateTries(cloneObj(usr1));

const newArray = storeUser(users, usr2);\*/

const pipe = function(...fns) {

return function(x) {

return fns.reduce(function(v, f) {

return f(v);

}, x);

}

};

const compose = function(...fns) {

return function(x) {

return fns.reduceRight(function(v, f) {

return f(v);

}, x);

}

};

const updateUser = pipe(

partGetUser,

cloneObj,

partUpdateScore30,

updateTries);

const newestUser = updateUser("Henry");

**Curring:-**

var doublenum = function(num){

return num\*num;

}

var addnum = function(n1, n2,n3,n4){

return n1+n2+n3+n4;

}

var createarr = function(num1,num2){

return [num1,num2]

}

var pipe = function(...args){

return function(val){

return args.reduce(function(p, f){

return f(p)

},val)

}

}

function curry(fn,arity = fn.length) {

//fn holds function name "addnum" and "createrr"

//arity holds lenght of the arguments to be recived eg: addnum.lenght gives 4

return (function nextCurried(prevArgs){

return function curried(nextArg){

var args = [ ...prevArgs, nextArg ];

if (args.length >= arity) {

return fn( ...args );

}

else {

return nextCurried( args );

}

};

})( [] );

}

var result = pipe(

doublenum,

curry(addnum)(1)(2)(3),

curry(createarr)(30)

);

console.log(result(5)) //[30,31]

/\*

first it recive [1]

second [1,2]

third [1,2,3]

fourth it return curried(){} function it self

so pipe can pass 25 to curried(){} function and if so the final call would be

curried(1,2,3,25){} and that calles fn( ...args );

"return (function nextCurried(prevArgs)"===> mainly used to return a curried() function

prevArgs===> holdes current array

nextArg===> holdes new argument passed

step1:- if above line executes fn recives addnum and arity holds 4

step2:- nextCurried will be invoded with empty array for the first time "prevArgs" holds empty array

step3:- and finally returns cirried() function to the caller

so the first call

flow1:- curry(addnum) recive function curried(nextArg){}

flow2:- curried() function invoked by (1)

flow3:- again curry(addnum)(1) recive function curried(nextArg){}

flow4:- curried() function invoked by (2)

flow5:- again curry(addnum)(1)(2) recive function curried(nextArg){} so...on....

nextArg in side curried() recive the value passed by (1)(2)...etc

\*/

 Imperative versus Declarative Programming

Imperative programming, we define everything using loops and conditions.

Declarative programming, we declare a programme it move by its own control

Declarative code, we develop Imperative code once and we are making it declarative to use effectively and easy to track or trace the flow and understating too.

var result = pipe(doublenum,  
curry(addnum)(1)(2)(3),  
curry(createarr)(30)  
);

we created pipe() and curry() functions we are combining both to get require result, we are not rewriting same pipe() or curry function every time.

Ie declare your code for any input called Imperative, reusing same login or rebuilding same logic called Imperative.

**How Factorial of a number works**

//console.log(5 \* 4 \* 3 \* 2 \* 1)

var factorial = function(num){

if(num === 1) {return 1;}

return num \* factorial(num-1)

}

console.log(factorial(5))

/\*

how this works

=============

factorial(5); is called function memory stack will be created below.

========================stack 5=========================

var factorial = function(1){

if(num === 1) {return 1;} // return 1

return 2 \* factorial(1)

}

// output 1

========================stack 4=========================

var factorial = function(2){

if(num === 1) {return 1;}

return 2 \* factorial(1)

}

// 2 \* 1 return 2

========================stack 3========================

var factorial = function(3){

if(num === 1) {return 1;}

return 3 \* factorial(2)

}

// 3 \* 2 return 6

=========================stack 2=======================

var factorial = function(4){

if(num === 1) {return 1;}

return 4 \* factorial(3)

}

// 4 \* 6 return 24

=====================stack 1===========================

var factorial = function(5){

if(num === 1) {return 1;}

return 5 \* factorial(4)

}

// 5 \* 24 return 120

for 5 iteration 5 function stack will be created, control execute top stack function

ie stack 5 than stack 4.....stack 1 and return the value

\*/