**Functional Programming**

**(small composable parts)**

Data and functions are separated. Functions operate on well-defined data structures like arrays and objects rather than belonging to that data structure like objects.

Clear + understandable

Easy to extend

Easy to maintain

Memory efficient

Dry

Important pillar: **Pure Functions**:

1. The Function should always return same output for the same input
2. The function cannot modify outside of it ( no side effects)

**1 Task**

**Return statement**

**Pure**

**Pure functions No shared state**

**Immutable state**

**composable**

**predictable**

**//not a pure function – side effect**

Function a(){

Console.log(‘hi’);

}

// it has effect on outside world

**map and concat methods can fix this issue**

const array = [1,2,3];

function mutateArray(arr) {

**const newArr = [ ].concat(arr)**

**newArr.pop()**

}

function mutateArray2(arr) {

**arr.map(item => arr.push(1))**

}

//The order of the function calls will matter.

mutateArray(array)

mutateArray2(array)

array

**//Side effects:**

const array = [1,2,3];

function mutateArray(arr) {

arr.pop()

}

function mutateArray2(arr) {

arr.forEach(item => arr.push(1 ))

}

//The order of the function calls will matter.

mutateArray(array)

mutateArray2(array)

array

function a(num1,num2){

return num1 + num2

}

function b(num){

return num \* 2;

}

b(a(3,4)) **// referential transparency**

**b(7) is equal to b(a(3,4))**

**Idempotent:** function always return the same thing

**Imperative vs Declarative:**

**Imperative code**

Tells the machine what to do and how to do

For (let i=0;i<1000;i++){

Console.log(i);

}

**Declarative code**

**Tells the machine what to do and what should happen ( does not tell computer how to do)**

[ ].forEach(item = > console.log(item)

React is declarative // but deep down everything has to be imperative

**Functional programs should be more declarative**

**Immutability:**  not changing the data or state **Higher-Order-Function and Closures**

Due to structural sharing and cheap memory , making a copy of data is efficient .

The data I receive is not mine and I am just going to make a copy and work on it .

**//HOF** const hof = (fn) => fn(5);

hof(function a(x){ return x})

**//Closure**

const closure = function() {

let count = 55;

return function getCounter() {

count++;

return count; // closure can be

}} //used as pure function

const getCounter = closure()

getCounter();getCounter();

HOF takes functions as args or returns functions as result

Closure allows the use of variable that is defined outside the scope

const obj = {name: 'Andrei'}

function clone(obj) {

**return {...obj}; // this is pure**

}

function updateName(obj) {

**const obj2 = clone(obj);**

obj2.name = 'Nana'

return obj2

}

const updatedObj = updateName(obj)

console.log(obj, updatedObj)

**Currying**: change the function from multiple parameters to one **partial application:** process of producing a function parameter at a time //useful to create small utitlity functions with smaller no of parameters

**//Currying**

**const multiply = (a, b) => a \* b**

**const curriedMultiply = (a) => (b) => a \* b**

curriedMultiply(5)(20) (or)

const multiplyBy5 = curriedMultiply(5)

multiplyBy5(20)

**//Partial Application**

const multiply = (a, b, c) => a \* b \* c

**const partialMultiplyBy5 = multiply.bind(null, 5)**

partialMultiplyBy5(10, 20)

// currying

**const curriedMultiply = (a) => (b) =>(c)=> a \* b\*c**

**Memoization:** form of caching, caching the return value based on the parameter (memorization if parameter is not changed)

Memoization is a way to remember a solution to a solved problem so no need to calculate again (if it’s a big one)

**//learn to cache**

function addTo80(n) {

return n + 80;

}

addTo80(5)

let cache = {};

function memoizeAddTo80(n) {

if (n in cache) {

return cache[n];

} else {

console.log('long time');

const answer = n + 80;

cache[n] = answer;

return answer;

}

}// console.log(1, memoizeAddTo80(6))

// // console.log(cache)

// // console.log('-----------')

// console.log(2, memoizeAddTo80(6))

// let's make that better with no global scope. This is closure in javascript so.

function memoizeAddTo80(n) {

let cache = {}; **// placing inside of the function**

return function(n) { **// using closure**

if (n in cache) {

return cache[n];

} else { ***// code efficiency and optimization - memoization***

console.log('long time');

const answer = n + 80;

cache[n] = answer;

return answer;

} } }

const memoized = memoizeAddTo80();

console.log(1, memoized(6))// console.log(cache)// console.log('-----------')

console.log(2, memoized(6))

**Compose and Pipe:**

**Compose:** compose different functions together. It deals with relationship b/w components: selected and assembled in various combinations: right to left

Compose is available through library – **ramda**

Math.abs()

**Creating our own compose function**

**const compose = (f,g) => (data) => f(g(data));**

const multiplyBy3 = (num) => num \* 3;

const makePositive = (num) => Math.abs(num);

const multiplyBy3AndAbsolute = compose(multiplyBy3,makePositive);

console.log(multiplyBy3AndAbsolute(-50));

fn1(fn2(fn3(50)));

**compose(fn1, fn2, fn3)(50) //Right to lext**

**pipe(fn3, fn2, fn1)(50)//left to right**

**const compose = (f, g) => (a) => f(g(a))**

**const pipe = (f, g) => (a) => g(f(a))**

const multiplyBy3AndAbsolute = compose((num) => num\*3, Math.abs)

console.log(multiplyBy3AndAbsolute(-50))

**Pipe: goes from left to right but order remains the same**

**Arity:** no of parameters. Arity should be less to compose (becomes harder)

**FP is useful in distributed computing and parallel programming**

**Amazon Exercise**

const itemsInCart = user.cart;

return Object.assign({}, user, { purchases: itemsInCart });

}

function emptyUserCart(user) {

history1.push(user)

return Object.assign({}, user, { cart: [] });

}

function refundItem() {}

function getUserState() {}

function goBack() {}

function goForward() {}

)(user, {name: 'laptop', price: 50})

function addItemToCart(user, item) {

history1.push(user)

const updatedCart = user.cart.concat(item)

return Object.assign({}, user, {cart: updatedCart});

}

function applyTaxToItems(user) {

history1.push(user)

const {cart} = user;

const taxRate = 1.3;

const updatedCart = cart.map(item => {

return {

name: item.name,

price: item.price\*taxRate

}

})

return Object.assign({}, user, { cart: updatedCart });

}

function buyItem(user) {

history1.push(user)

const itemsInCart = user.cart;

return Object.assign({}, user, { purchases: itemsInCart });

}

function emptyUserCart(user) {

history1.push(user)

return Object.assign({}, user, { cart: [] });

}

function refundItem() {

}

function getUserState() {

}

function goBack() {

}

function goForward() {

}

const user = {

name: 'Kim',

active: true,

cart: [],

purchases: []

}

const history1 = []; // spread operator to get n args

const compose = (f, g) => (...args) => f(g(...args))

const pipe = (f, g) => (...args) => g(f(...args))

const purchaseItem = (...fns) => fns.reduce(compose);

const purchaseItem2 = (...fns) => fns.reduce(pipe);

//purchaseItem2( // reduce - > iterates

// addItemToCart, // through each function

// applyTaxToItems, //…fns – n functions

// buyItem,

// emptyUserCart,

//)(user, {name: 'laptop', price: 60})

purchaseItem(

emptyUserCart,

buyItem,

applyTaxToItems,

addItemToCart

// )(user, {name: 'laptop', price: 50})

function addItemToCart(user, item) {

history1.push(user)

const updatedCart = user.cart.concat(item)

return Object.assign({}, user, {cart: updatedCart});

}

function applyTaxToItems(user) {

history1.push(user)

const {cart} = user;

const taxRate = 1.3;

const updatedCart = cart.map(item => {

return {

name: item.name,

price: item.price\*taxRate

}

})

return Object.assign({}, user, { cart: updatedCart });

}

function buyItem(user) {

history1.push(user)

const itemsInCart = user.cart;

return Object.assign({}, user, { purchases: itemsInCart });

}

function emptyUserCart(user) {

history1.push(user)

return Object.assign({}, user, { cart: [] });

}

function refundItem() {

}

function getUserState() {

}

function goBack() {

}

function goForward() {

}