Some of method which is not recommended to use in js.

→ eval()
→ arguments
→ for in
→ with
→ delete

- Hidden classes
- Inline caching

// hidden classes

```
function Animal (x, y) {
    this.x = x;
    this.y = y;
}

const obj1 = new Animal(1, 2);
const obj2 = new Animal (3, 4);

obj1.a = 300;
obj1.b = 100;
obj2.b = 30;
obj2.a = 10;
```

Animal is hidden class and have used to create the object

If we change the order then it assume that obj1 & obj2 did not have same class it will confuse. And internally it will slow down the process

// inline caching

```
function finduser (user) {
    return `found ${user.firstName} ${user.lastLastName}`
}
```

Be Positive...

DATE ___/___/___

```
const userData {
    firstName : 'Johson'
    lastName : 'Junior'
}
```

findUser (userData)

It will replace by the function itself

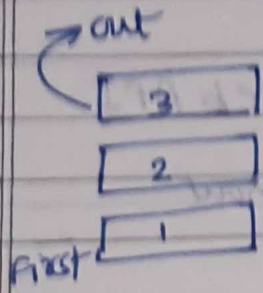
(11) WebAssembly

Be Positive...

(12) Call stack and Memory Heap

DATE / /

first in last out (FILO)



An unordered memory.

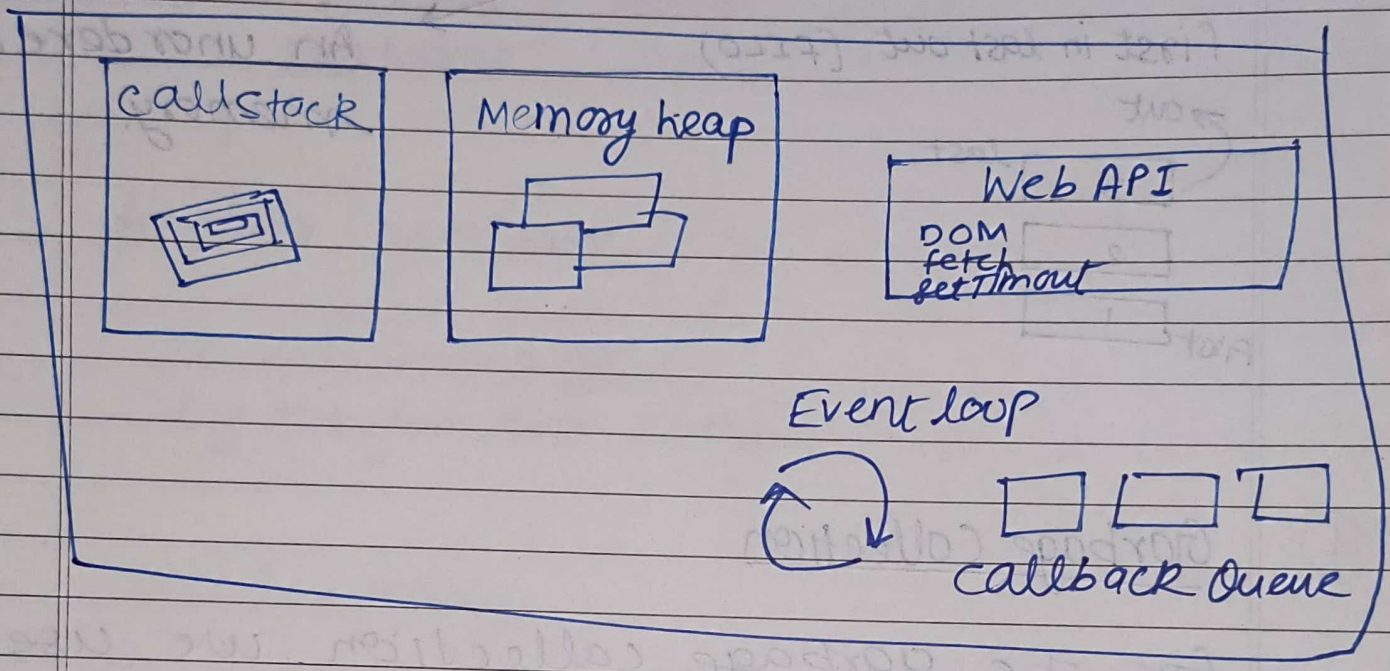
(14) Garbage Collection

for the garbage collection we used mark & sweep algorithm in js

(15)
```
// global variable
var a = 1
var b = 2

// event listeners
var element = document.getElementById(
element.addEventListenet ('click', onClick)

// setInterval

setInterval (() => {
        // reference object
    })
```

(18) DATE / /

JavaScript Runtime

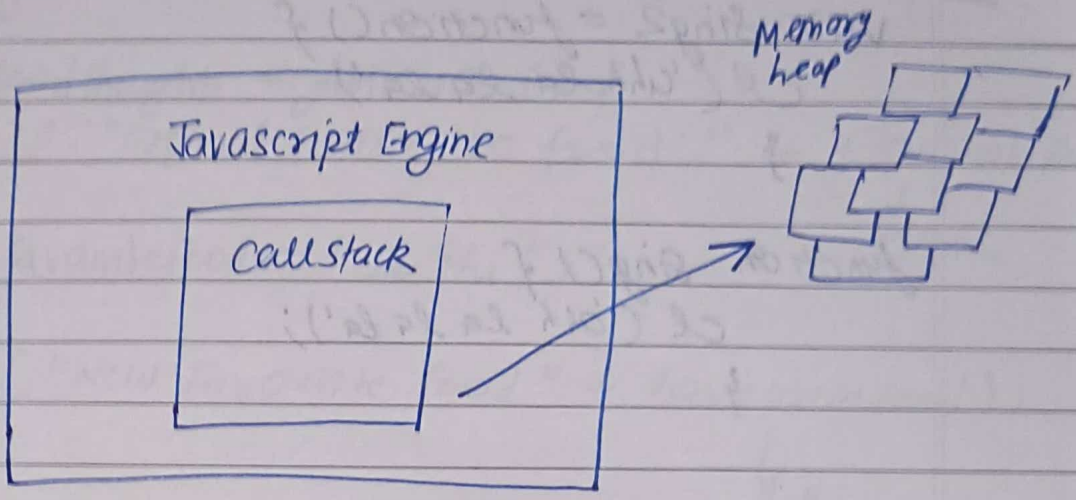


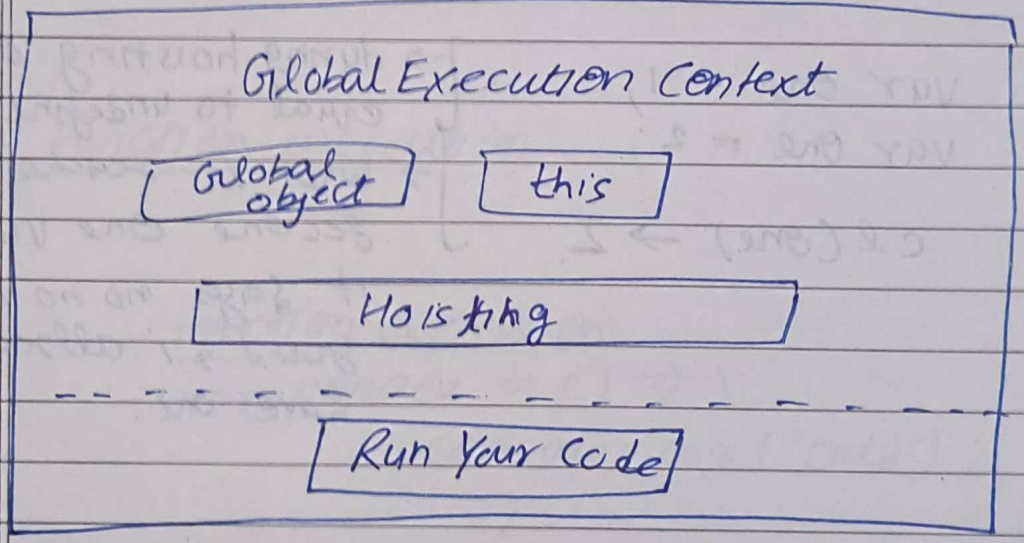
Be Positive...

JavaScript foundation II

DATE / /

Hoisting





eg. → variable (var) are partially hosted.
→ function are hosted

```
cl(teddy)  ——→ undefined
cl(sing2)  ——→ undefined
var teddy = 'bear';

var sing2 = function() {
        cl('uhhh la la la')
    }
```

Be Positive...

```
function sing() {
    cl('uhh la la la') }
```

DATE ___/___/___

eg2 cl(sing2()) TypeError ^ sing2 is not a function

```
var sing2 = function() {
    cl('uhh la la la')
}

function sing() {
    cl('ohh la la la');
}
```

(5)

eg.:
```
var one = 1;
var one = 2;
cl(one) → 2
```

→ during hoisting one is equal to undefined.
→ when it reaches to second one variable it says no no I'm good, i already have one.

eg a() ⟶ bye.

```
function a() {
    cl('hi')
}

function a() {
    cl('bye')
}
```

Be Positive...

DATE ___/___/___

6

```
var favouriteFood = "grapes";

var foodThoughts = function () {
    cl ("original favourite food:" + favouriteFood);

var favouritefood = "sushi";

cl ("New Favourite food" + favouritefood);
};
```

undefine

sushi

8 Function invocation

function expression

```
var canada = () => {
        console.log ('cold');
    }
```

function declaration

```
function india () {
    console.log ('warm');
}
```

// function Invocation / call / Execution

```
canada()
india()
```

TM
B+ be positive
Be Positive...

9

Arguments keyword

DATE ___/___/___

Arguments looks like array but it is not array let see in the example.

```
function marry (person1, person2)
{
    console.log ('arguments', arguments)
    return `${person1} is now married to ${person2}`
}

marry ('Tim', 'Tina')
```

{0: 'Tim', 1: 'Tina'}

* → We can create an array from these object

```
Array.from (arguments) → ['Tina', 'Tim']
```

* → We can also used another approach.

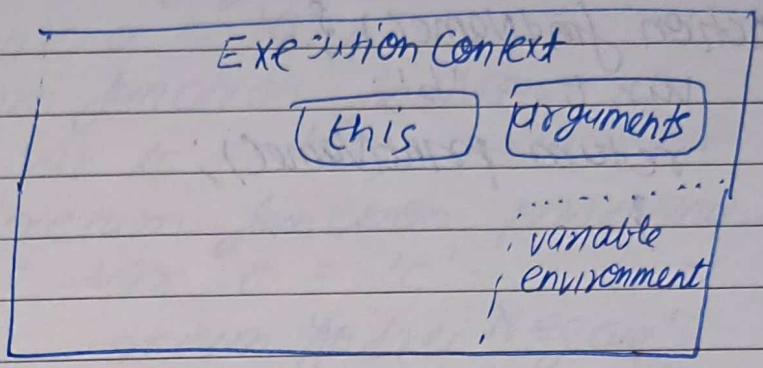
```
function marry2 (...args)
{
    console.log ('argument:', args)
}

marry2('Tim', 'cook');
```

['Tim', 'cook']

Be Positive...

10 Function invocation

DATE / /




```
function two(){
    var isValid; //undefined
    }

function one(){
    var isValid = true; // local env
    two(); // new EC
    }

var isValid = false;
one();

// two()    - undefined
// one()    - true
// global() - false
```

Be Positive...

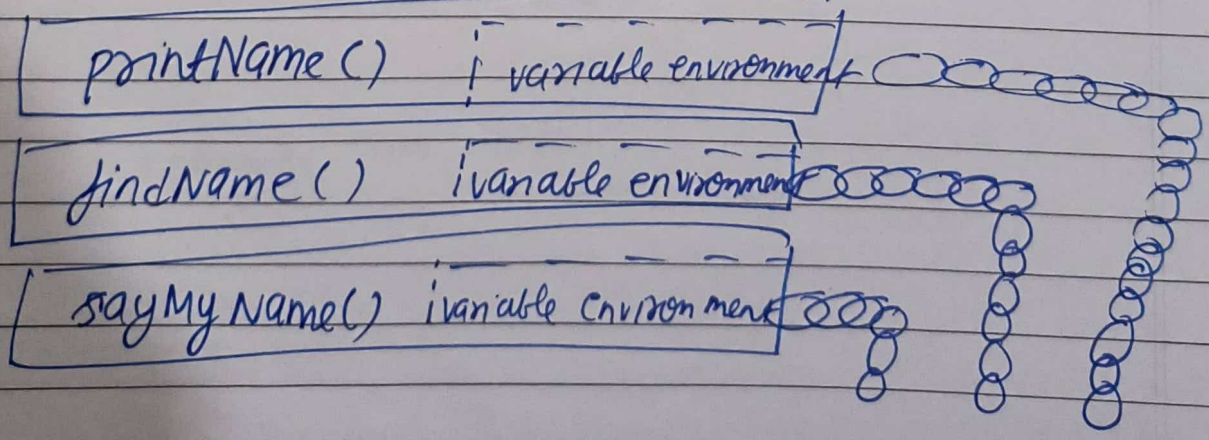
11

DATE / /

```
var x = 5;
function findName() {
    var b = 'b';
    return printName();
}

function printName() {
    var c = 'c';
    return 'Andrei Neogoie'
}

function sayMyName() {
    var a = 'a';
    return findName()
}

sayMyName()
```

* All the function have global access to variable x.

global lexical Environment



Be Positive...

eg 2

```
function sayMyName(){
    var a = 'a';
    return function findName(){
        var b = 'b';          → can access a
        return function printName(){
            var c = 'c';      → can access a, b
            return 'Andrei Negoie'
        }
    }
}
sayMyName()()()
```

13

```
'use strict'
function weird(){
    height = 50;
    return height
}
```

shows error.
if remove the 'use strict' then work fine.

```
var heyhey = function doodle(){
    return 'heyhey'
}

heyhey()  // heyhey
doodle()  // reference error
```

doodle will work inside a function.

DATE / /

14 function scope Vs Block scope.

→ block scoping if we use let or const
→ var to create at global scope.

18

* This is the object that the function is a ~~property~~ of.

eg ~~var a = 5~~

```
function a() {
    console.log(this) → window object
}
a();
```

→ window.a()

* gives methods access to their object
* execute same code for multiple objects.

Be Positive...

19

DATE __/__/__

```
const = {
    name: 'Billy',
    sing() {
        console.log('a', this);
        var anotherfunc = function() {
            console.log('b', this)
        }
        anotherfunc()
    }
}
obj.sing()
```

→ if changed to = () => {

o/p
a → { nam: "Billy", sing: f }
b → window

o/p
a → { name: "Billy", sing: f }
b → { ... }

20 Call(), apply(), bind()

TM
B+ positive
Be Positive...

imp

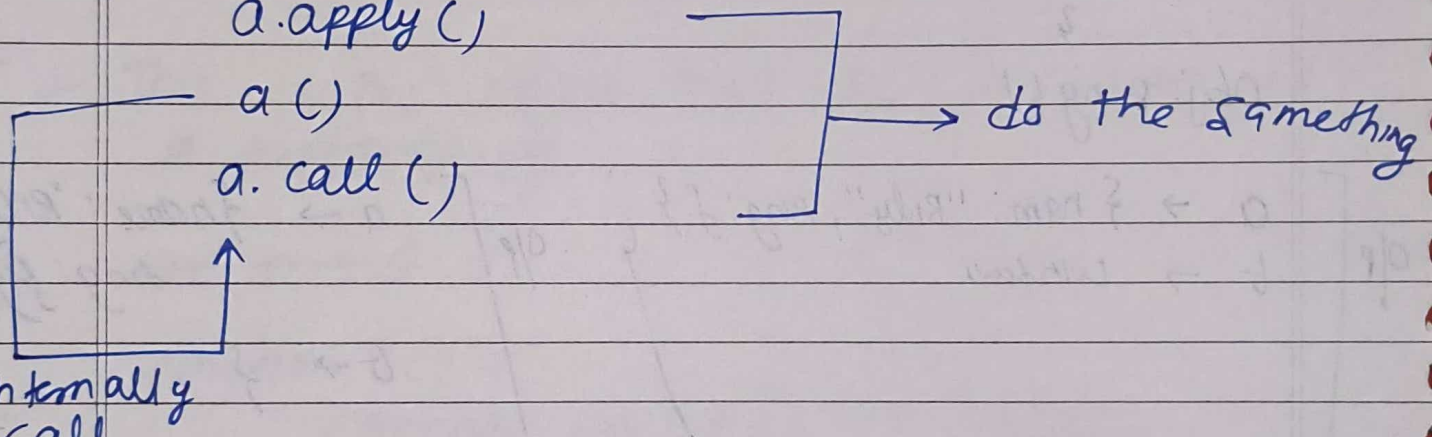
20

Call, apply, bind

DATE / /

→ Call() is alway running internally to — call or invoke the function.

eg
```
function a() {
    console.log('hi');
}
```



```
const wizard = {
    name : 'Merlin',
    health : 50,
    heal() {
        return this.health = 100;
    }
}

const archer = {
    name: 'Robin Hood',
    health : 30
}
```

Be Positive...

DATE / /

Q How can we add the heall method to archer.

Ans wizard.heal.call(archer)

Hey just call heall() function but not - use wizard obj but use archer object.

Suppose

```
wizard = {
    heal (num1, num2) {
        return this.health += num1 + num2;
    }
}
```

Now

take direct parameter

```
wizard.heal.call(archer, 50, 30)
wizard.heal.apply(archer, [50, 30])
```

↳ take array

```
const a = wizard.heal.bind(archer, 100, 30)
a()
```

↓

return function

Be Positive...

22 **Bind() & currying**

DATE __/__/__

```
function multiply (a,b){
    return a*b
}

let multiplyByTwo = multiply.bind(this,2);
console.log(multiplyByTwo(4)) //4

let multiplyByTen = multiply.bind(this,10);
console.log(multiplyByTen(4)) //10
```

* When we are passing incomplete argument and after that providing other argument then this is called currying.

23

```
var b = { name: 'jay'
    say() { console.log(this)
    }
```

b.say()

```
var c = {
    name: 'jay',
    say: { return function(){
        console.log(this)
    }}
}
```

window

c.say()()

```
var d = { name: 'jay',
    say { return () => console.log(this)}
}
```

d.say()()

Be Positive...

4 Types in JavaScript

DATE ___/___/___

2 JavaScript Types

		typeof	Definition
Primitive	1. 5	number	
	2. true	boolean	
	3. 'To be or not to be'	String	
	4. undefined	undefined	absence of definition
	5. null	object	absence of value
	6. Symbol ('just me')	symbol	
Non Primitive	7. { }	Object	
	8. [ ]	object	
	9. function () { }	object	

Object
↑
[ ] function ()

eg

```
true.toString()          // 'true'
```

↓ The internal mechanism

```
Boolean(true).toString()
```

↓

Type Cast.

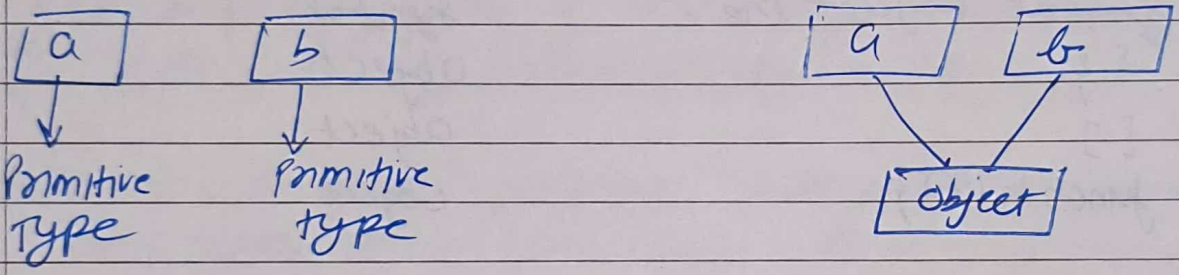
3. Arrays.isArray()

DATE __/__/__

→ Array.isArray([])

↳ to check if it is array or not
return true/false.

4. Pass by value Vs Pass by reference




```
var a = 5
var b = a
  b++

console.log(a)  // 5
console.log(b)  // 6
```

```
let obj1 = {name: 'Yao'
            password: '123'};
let obj2 = obj1;
obj2.password = 'easy'

console.log(obj1.password)
console.log(obj1.password)

// 'easy'
```

Copy object (without reference)

```
let obj = {a: 'a', b: 'b', c: 'c'};
let clone = Object.assign({}, obj);
let clone1 = ~~Object.assign~~ {...obj}
  obj.c = 5
  console.log(clone)
let clone2 = JSON.parse(JSON.stringify(obj))
```

it will take time if object is large.

B+ be positive TM
Be Positive...

DATE / /

Copy arrays (without reference)

```
var c = [1, 2, 3, 4]
var d = [7].concat(c)
d.push(123)
console.log(d)
```

7 Type Coercion

eg: `1 == '1'` true

eg:
```
if (1)
{
}
```

} automatic conversion in data type.

5 The 2 Pillar Closures & Prototypal Inheritance.

DATE ___/___/___

2

Objects
object — Array
Function.

① function one() { }

one()

② const obj = {
two: function() {
return 2;
}
}
obj.two()

③ function three() {
return 3
}
three.call()

3 way to call/invoke function

TM
B+ be positive
Be Positive...

4th way

DATE __/__/__

```
const four = new Function ('return 4
four();   // 4
```

```
const four = new Function ('num', 'return nu
```

Parameter or argument

last argument is body of functio

```
function wcohco(){
    log('wcohco')
}
wcohco.yell = 'ahhh'
const specialObj = {
    yell : 'ahhhh',
    name : 'wohco'
    () : consale.log('wohco')
}
```

Pseudo code or algorithm running inside it

TM
B+ be positive
Be Positive...

3 First Class citizens

DATE / /

→ Function are first class citizens in JS

* assign function

① var stuff = function () { }

* pass function as parameter

② function a(fn) {

fn()

}

* return function.

③ function b() {

return function c() { console.log('bye') }

}

4 Extra Bits function.

eg

```
for (let i = 0; i < 10; i++)
    function() { }
    a()
}

for (let i = 0; i < 10; i++)
    { a()
    }
function a() { }
```

DO not initialize function again & again

Be Positive...

DATE ___/___/___

To avoid parameter reference error

eg

```
function a( ) {
    return param
}
a()
```

param does not exist
give error

```
function a (param=6) {
    return param
}
a()
```

✓

5. Higher order function

1. function
2. function (a, b)
3. HOF

TM
B+ be positive
Be Positive...

6

Exercise HOF

const multiplyBy = (num1) => (num2) => num1 * num2

```
const multiplyBy = function (num1) {
    return function (num2) {
        return num1 * num2
    }
}
```

eg multiplyBy(4)(6)

```
const multiplyByTwo = multiplyBy(2);
const multiplyByFive = multiplyBy(5);

multiplyByTWO(4)
multiplyByTWO(10)
multiplyByFive(6)
```

7

Closures

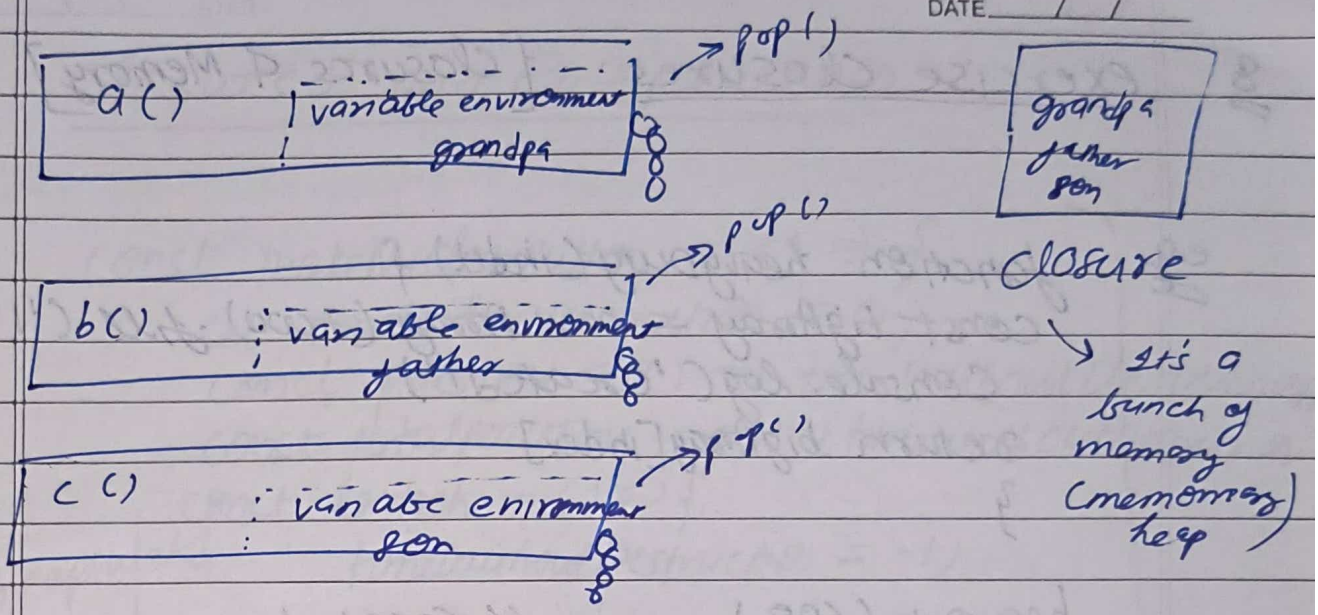
```
function a() {
  let grandpa = 'grandpa';
  return function b() {
    let father = 'father'
    return function c() {
      let son = 'son'
      return `${grandpa} > ${father} > ${son}`
    }
  }
}
```

the function able to access the the above variable due to closures property.

(1) injection token (2) injection function (3) injectable error

Be Positive...

DATE ___/___/___



→ We have read previously about
Lexical Environment ===[[scope]]

→ Why can we use closure
function() + lexicalScope

eg 2:

```
const boo = (string) => (name) => (name2) =>
    log(`${string} ${name} ${name2}`)

boostring = boo('hi');
//5 years
const boostringName = boostring()
```

DATE ___/___/___

8 Exercise Closures [Closures & Memory]

eg

```
function heavyDuty(index) {
  const bigArray = new Array(7000).fill('I')
   console.log('created!')
  return bigArray[index]
}

heavyDuty(688)      // created
heavyDuty(689)      //   "
heavyDuty(690)      //   "
heavyDuty(691)      //   "
```

```
function heavyDuty2() {
   const bigArray = newArray(7000).fill('O')
   consol.log('created Again')
   return function(index) {
   return bigArray[index]
   }
}

const getHeavyDuty = heavyDuty2(); // created again
   getHeavyDuty(688)
      "        (700)
      "        (800)
      "        (900)
      .
```

** Next pattern the global scope variable →

Be Positive...

DATE ___/___/___

10 # Closures and Encapsulation

```
const makeNuclearButton = () => {
    let timeWithoutDestruction = 0;
    const passTime = () => timeWithoutDestruction++;
    const totalPeaceTime = () => timeWithoutDestruction;
    const launch = () => {
        timeWithoutDestruction = -1;
        return '*';
    };
    setInterval(passTime, 1000);
    return {
        totalPeaceTime: totalPeaceTime
    };
};
const ohno = makeNuclearButton();
ohno.launch()   // error
```

Encapsulated

12 eg run or initialized a function only first time (how can u do this)

```
let view;
function initialize() {
    let called = 0;
    return function() {
        if (called > 0) {
            return;
        } else {
            view = 'Hey';
            called++;
            console.log('View has been set!');
        }
    }
}
```

TM B+ be positive
Be Positive...

DATE / /

```
const startOnce = Initialize();
startOnce();
startOnce();
```
} → called only one time

13

```
const array = [1,2,3,4];

for (var i=0; i<array.length; i++) {
    setTimeout(function(){
        console.log('I am at Index' +i)
    }, 3000)
}
```

O/P

I am at index 4
" " " " 4
" " " " 4
" " " " 4

14

```
const array = [1,2,3,4]
for (let i=0; i<array.length; i++) {
    setTimeout(function(){
        console.log('I am at index' + i)
    }, 3000)
}
```

when you declare local variable using let

I am at Index 0
" " " " 1
" " " " 2
" " " " 3

Be Positive...

DATE ___/___/___

```
const array = [1,2,3,4];
for (var i=0; i< array.length; i++) {
    (function (closureI) {
        setTimeout (function() {
            console.log ('I am at index ' + array[closureI]);
        }, 3000)
    }) (i)
}
```

I am at index 1
" " " " 2
" " " " 3
" " " " 4
" " " " 5

15 Jan

Closures Review

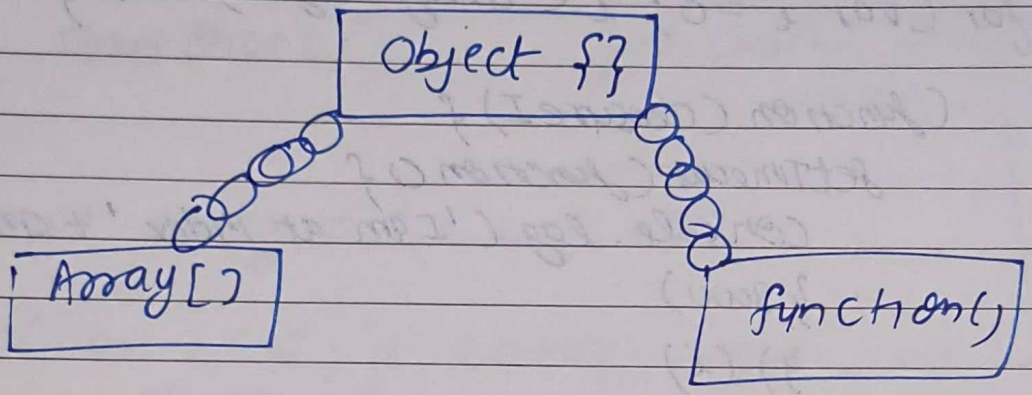
Closure is the combination of a function & the lexical environment from which it was declared & we are able to do this because function is 1st class citizen & we have lexical scope

function + Lexical scope

∴ Closures allow a function to access variable from an enclosing scope or outer scope environment even after it leaves the scope in which it was declared.

DATE ___/___/___

16 Prototypal Inheritance



function & Array access the property of objects using prototypal inheritance.

eg const array = []

array.__proto__

→ gives [] array method.

array.__proto__.....__proto__

→ gives {} methods

17

DATE / /

17

```
let dragon = {
    name: 'Tanya',
    fire: true,
    fight() {
        return 5
    },
    sing() {
        return 'I am ${this.name}, the breather of
                fire';
    }
}

dragon.fight() // 5

let lizard = {
    name: 'Kiki',
    fight() {
        return 1
    }
}

const singLizard = dragon.sing.bind(lizard)
console.log(singLizard())
// I am Kiki, the breather of fire
```

we pass the sing() in lizard using bind

++ Another approach

```
lizard.__proto__ = dargon
lizard.sing() // I am Kiki, the breather of fire.
lizard.fire // true
```

dragon.isPrototypeOf(lizard) // true.

TM B+ be positive
Be Positive...

18 # Prototypal Inheritance

DATE ___/___/___

previous example continued

```
lizard.__proto__ = dragon;

for (let prop in lizard) {
    console.log(prop)
}
```

name
fight
fire
sing

```
for (let prop in lizard) {
    if (lizard.hasOwnProperty(prop)) {
        console.log(prop)
    }
}
```

→ name, fight

19

```
const obj = {}
obj.__proto__
```

▶ It will give objects

```
obj.__proto__.__proto__
```

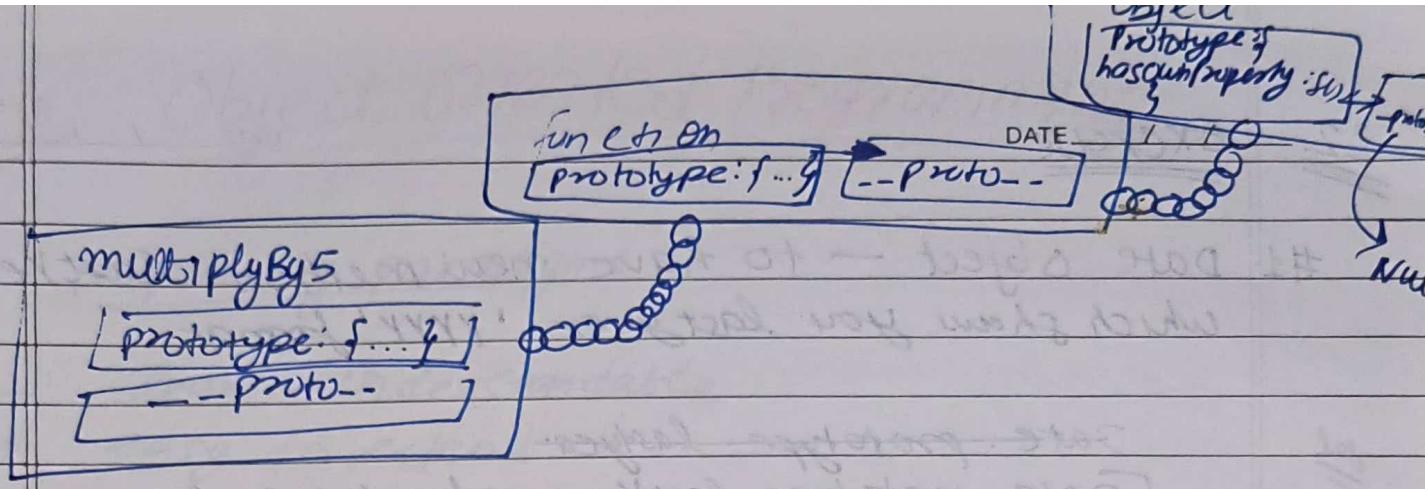
▶ null

* `obj.hasOwnProperty('hasOwnProperty')` // false.

*
```
function a() {}
a.hasOwnProperty('apply')   // false
                ('call')    // false
                ('bind')    // false
```

Why false

Be Positive...



eg
```
function multiplyBy5(num) {
    return num*5
}
> multiplyBy5.__proto__
>
```

The right way to use ↓ Prototypal Inheritance

20

```
let human = {
    mortal: true
}
let socrates = Object.create(human)
console.log(socrates) // empty object
socrates.age = 45
console.log(socrates.mortal) // true
console.log(human.isPrototypeOf(socrates)) // true
```

21

Only function have the prototype property

22 Exercise

DATE ___/___/___

#1 Date object — to have new method lastYear(), which shows you last year 'YYYY' format

sol

~~Date.prototype.lastyear~~

```
Date.prototype.lastYear = function() {
    return this.getFullYear() - 1;
}

new Date('1900-10-10').lastYear()
```

++ if i changed to arrow function

```
Date.prototype.lastYear = () => {
    return this.getFullYear() - 1;
}
```

[Shows error]

#2 Modify .map() to print '☒' at the end of each item

sol

```
Array.prototype.map = function () {
    let arr = [];
    for (let i = 0; i < this.length; i++)
    {
        arr.push ((this[i] + '☒'))
    }
    return arr
}
console.log ([1,2,3].map())
// 1☒, 2☒, 3☒
```

6 Object Oriented Programming

DATE 1 /

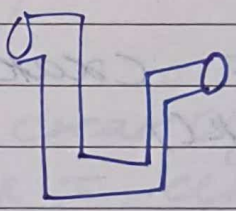
1. Programming paradigms :-

→ Clear + Understandable
→ easy to extend
→ easy to maintain
→ memory efficient
→ dry

2. JavaScript = Scheme — Java

function — object

FP — OOP



4.

```
const ely = {
    name: 'Sally',
    weapon: 'bow',
    attack() {
        return `attack with ${this.weapon}`
    }
}
ely.attack()
```

level 1

TM
B+ be positive
Be Positive...

DATE / /

// factory function

level 2

```
function createEly(name, weapon)
    {
    return {
        name: name,
        weapon: weapon,
        attack() {
            return 'attack with' + weapon
            }
        }
    }
```

if property & key are same then do this

```
return {name, weapon, attack()}
```

```
const peter = createEly('peter', 'stones')
peter.attack()

const sam = createEly('sam', 'fire')
sam.attack()
```

* see attack function, if there are 100 object then we are creating 100 function - location.

Be Positive...

DATE / /

5 OOP2 Object.create()

level 3

```
const elfFunctions = {
    attack() {
        return 'attack with ' + this.weapon
    }
}

function createElf(name, weapon) {
    return {
        name,
        weapon
    }
}

const peter = createElf('Peter', 'stones')
peter.attack = elfFunctions.attack
peter.attack()

const peters = createElf('Peters', 'fires')
peters.attack = elfFunctions.attack
peters.attack()
```

→ still we are repeating same thing again & again

→ see in the next.

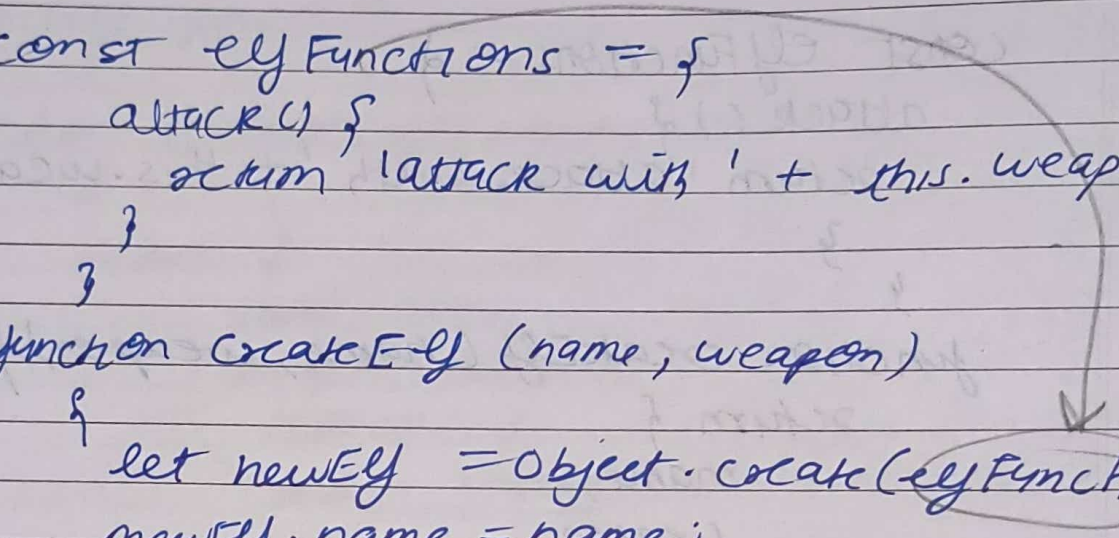
Be Positive...

DATE ___/___/___

// Using Object.create()

level 1

```
const elyFunctions = {
    attack() {
        return 'attack with ' + this.weapon
    }
}
function createEly (name, weapon)
    {
    let newEly = Object.create(elyFunctions)
    newEly.name = name;
    newEly.weapon = weapon;
    return newEly;
    }
const peter = createEly ('Peter', 'stones')
peter.attack()
```



6 what we do when we didn't have Object.create

We used construction function

```
function Ely (name, weapon) {
    this.name = name;
    this.weapon = weapon;
    }
Ely.prototype.attack = function () {
    return 'attack with ' + this.weapon }
const peter = new Ely ('Peter', 'stones')
peter.name // Peter
```

alternate

```
const ELF1 = new Function ('name', 'weapon',
    'this.name = name;
    this.weapon = weapon;')
```

TM
B+ be positive
Be Positive...

DATE ___/___/___

```
const sarh = new Elf1 ('sarh', 'firework')
sarh.name // sarh
```

* ⇒ arrow function is lexical scope
⇒ regular function is dynamically scoped (work on the who is calling.)

7 More construction function

```
function Ely (name, weapon)
    log (this); // {}
this.name = name;
this.weapon = weapon;
    log (this); // { name: 'Peter', weapon: 'ston
}
```

Before we have empty object

→ After we have all data

```
Ely.prototype.attack = function () {
    return 'attack with' + this.weapon
};
```

```
const peter = new Ely ('Peter', 'stones')
log (peter.__proto__); // Ely {attack: [Functi
```

we have created function in prototype which - found in __proto__

* Function has access to prototype & object has access to __proto__

TM
B+ be positive
Be Positive...

Continued

DATE / /

```
Elf.prototype.build = function() {
    const self = this;
    function building() {
        return self.name + 'builds a house';
    }
    return building()
}
```

8 Funny thing About JS

```
var a = new Number(5)
typeof a // object

var b = 5
typeof b // number

a === b // false
```

Date
Number
Boolean
:
} All these have inbuild prototype method

B+ TM Be Positive...

Nobody like Prototype

#9 DATE __/__/__

ES6 class : class allow to write - inside it.

```
class Ely {
    constructor(name, weapon) {
        this.name = name;
        this.weapon = weapon;
    }
    attack() {
        return 'attack with ' + this.weapon
    }
}
```

Instance happen when you call a class and excate an object.

```
const peter = new Ely('Peter', 'stones');
log(peter instanceof Ely) // true.
```

#10 Object.excate() Vs class : every developer goes according to their convenience

TM
Be Positive...

11 this → 4 ways

DATE ___/___/___

// new binding this

```
function Person (name, age) {
    this.name = name;
    this.age = age;
}
const person1 = new Person ('xavier', 55)
person1
```

// implicit binding

```
const person = {
    name : 'karen',
    age : 40,
    hi () {
    console.log ('hi' + this.name)
    }
}
```

// explicit binding

```
const person3 = {
    name : 'karen',
    age : 40,
    hi () {
    console.log ('hi' + this.setTimeout)
    }.bind (window)
}
person3.hi()
```

B+ be positive TM
Be Positive...

DATE ___/___/___

// arrow function

```
const person4 = {
    name: 'Karen',
    age: 40,
    hi: function () {
        var inner = () => {
            console.log('hi' + this.name)
        }
        return inner()
    }
}
person4.hi()
```

Arrow function is used that is why this refer to person object

12 Inheritance

```
class Ely {
    constructor(name, weapon) {
        this.name = name;
        this.weapon = weapon;
    }
    attack() {
        return 'attack with ' + this.weapon;
    }
}
const fiona = new Ely('Fiona', 'ninja stars')
const ogre = {...fiona}
```

ogre.__proto__ = undefined; has object value but did not have prototype function ie

Boolean(...) === ogre // false not have attack method.

DATE ___/___/___

```
class character {
    constructor (name, weapon) {
        this.name = name;
        this.weapon = weapon;
    }
    attack() {
        return 'attack with ' + this.weapon;
    }
}

class Elf extends character {
    constructor (name, weapon, type) {
        super (name, weapon)
        this.type = type
    }
}

const dalby = new Elf ('Dolby', 'cloth', 'house');
dalby.attack(); // attack with cloth
```

14 Public Vs Private

In JS we didn't have something like this concept.

> we can declar variable with - abc] But nothing done my JS comp

B+ be positive
Be Positive...

7 Functional Programming

DATE ___/___/___

4 Pure function

// no side effect
// I/P → output

Not pure

```
const array = [1,2,3]
function mutateArray(arr) {
    arr.pop()
}
```

Not pure

```
function mutateArray2(arr) {
    arr.forEach(it => {
        it.push(1)
    })
}
```

Pure function

```
function removeLastItem(arr) {
    const newArray = [].concat(arr);
    newArray.pop()
    return newArray
}
```

7 Idempotent : a function return same o/p for respective I/P

```
function notGood(t) {
    return Math.random(t)
}
```

TM
B+ be positive
Be Positive...

DATE / /

8 Imperative Vs Declarative

↓ writing all instruction

↓ only set the task

```
for(let i=0; i<1000; i++)
  { console.log(i)
  }
```

```
[1, 2, 3].forEach(it =>
  console.log(item))
```

9 Immutability (Not to change data)

```
const obj = {name: 'Andrei'}

function clone(obj) {
    return {...obj};
  }

function updateName(obj) {
    const obj2 = clone(obj);
    obj2.name = 'Nana'
    return obj2
  }

const updatedObj = updateName(obj)
```

TM B+ be positive
Be Positive...

DATE ___/___/___

10 Higher Order Function and closures

HOF

```
const hof = (fn) => fn(5);
hof(function a(x){return x})
```

Closure

```
const closure = function(){
    let count = 55
    return function getCounter(){
        return count;
    }
}
const getCounter = closure();
getCounter(); //55
```

11 Currying

example

```
Const curriedMultiply = (a) => (b) => a*b;
curriedMultiply(5)(3)
```

PartialApplication

```
const multiply = (a,b,c) => a*b*c;
const partialMultiplyBy5 = multiply.bind(null,5);
partialMultiplyBy5(4,10)   //200
```

TM
B+ be positive
Be Positive...

DATE ___/___/___

14 MCI Memorization

```
function addTo80(n){
console.log('long time')
  return n+80;
}
```

```
let cache ={};
function memoizedAddTo80(n){
  if(n in cache)
  {return cache[n];
  }else{
  console.log('long time');
    cache[n] =5+80;
    return cache[n];
}}
```

make in returning time & function

```
console.log('1', memoizedAddTo80(5))
console.log('2', memoizedAddTo80(5));
```

// long time
85
85

Be Positive...

Ramda Library

DATE ___/___/___

16 Compose

```
const compose = (f, g) => (data) => f(g(data))

const multiplyBy3 = (num) => num * 3;
const makePositive = (num) => Math.abs(num);

const multiplyBy3AndAbsolute = compose(
        multiplyBy3, makePositive)

multiplyBy3AndAbsolute(-50)
```

Pipe | only difference in Pipe & compose is to interchange the function flow:

∴ compose : left → right
∴ pipe : right → left

```
const pipe = (f, g) => (data) => g(f(data))
```

17 Arity : No. of Argument function can take

→ one or two parameter.

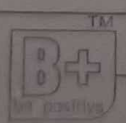
TM
Be Positive...

DATE ___/___/___

19 Implement a cart feature

1. Add item to cart
2. Add 3% tax to item in cart
3. Buy item : cart → purchase
4. empty cart

```
const user = {
  name: 'Kim',
  active: true,
  cart: [],
  purchase: []
}

const history1 = [];
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(addItemToCart,
              applyTaxToItems,
              buyItem,
              emptyUserCart,
              )(user, {name: 'laptop', price: 60});

function addItemToCart(user, item) {
  history1.push(user)
  const updatedCart = user.cart.concat(item)
  return Object.assign({}, user, {cart: updatedCart});
}
```



DATE ___/___/___

```
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 itemInCart = user.cart;
    return Object.assign({} user, {purchases: itemInCart});
    }

function emptyUserCart(user){
      history1.push(user)
    return Object.assign({}, user, {cart: []});
    }
```

TM
B+
positive
Be Positive...

8

OOP vs FP

DATE / /

(1) Composition

// what it has

```
function getAttack (character) {
    return Object.assign({}, character, {attackFn: () => {}})
}

function Elf (name, weapon, type) {
    let elf = {
        name,
        weapon,
        type
    }
    return getAttack (elf)
}
```

(3)

FP (Disadvantage)

- → many operations on fixed data
- → Stateless
- → pure
- → declarative

OOP (Disadvantage)

- → few operation on common data
- → stateful
- → side effect
- → imperative

Be Positive...