Lecture (PDF)

The Hard Parts of OOP

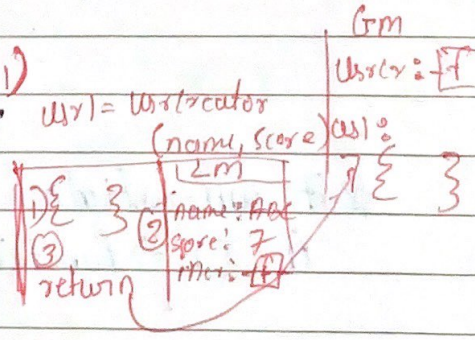
- To organise the code & Add new features & functionality

⇒ We can create a obj to assign or bundle the properties together. or by using (.) to add new property to the obj later on.

Solution 1 :- (Copy of functions on user1)

```
function usrCreator(name, score) {
  const newUsr = { };
  newUsr.name = name;
  newUsr.score = score;
  newUsr.increment = function() {
    this.score ++; };
  return newUsr;
}
```

★ This is not the App to do this every time user created function is created



const usr1 = usrCreator("ABC", 7);
usr1.increment()

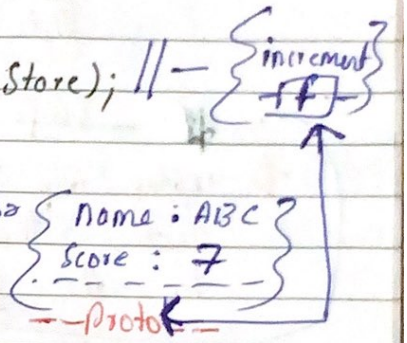
Que:- How New keyword work underHood?

Page No.
Date / /

* Solution 2:-

In this we will have to create a link to the object, in the global memory and we will do that using Object.Create(obj name). this will the the Object __proto__ will link to that object in Global memory.

Ex. function usrCreator(name, score){
  const newUsr = Object.Create(funStore); //
  newUsr.name = name;
  newUsr.score = score;
  return newUsr; //
}



const funStore = {
  increment: function() { this.score++; }
}

const usr1 = UsrCreator('ABC', 7);
usr1.increment(); // it will look through the proto link.

* Solution 3:-

Using new keyword to make the __proto__ bond to the function store object in the global memory

* The new keyword does 3 things:-
  1) create a this: { }
  2) makes the bond to Object.prototype. fun link.
  3) returns the object.

1) The function is both function & obj in JS
The link is ~~no~~ the object part of function store the additional function, using the prototype property of the function object part.

Page No.
Date / /

* When we do the call using obj created & call the method in the obj part of the function, (that has this keyword in it) then there is a rule. Whatever is the left hand side of dot. the this is assigned to that.

Ex. usr1.increment()

this : usr1

* Imp * Bug * Gotcha

* Very imp :- if there is a function in it nested that says this then this is Assigned to the global which is undefined.

to overcome → write [that = this; this++;] In the outer increment function

By Closure

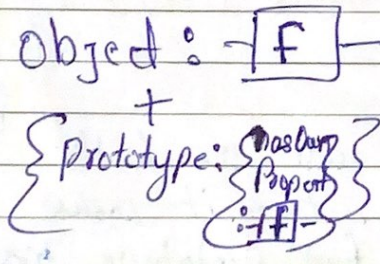
* But if we use the Arrow function as the nested function, the the this is lexically scoped to the object calling that method.

* Solution :- using class keyword,
1) the Above Soln are modified by using the class Keyword, in that the property is stored in the constructor (in this case) & function as the prototype, Property of obj.

✱ Default prototype chain :-

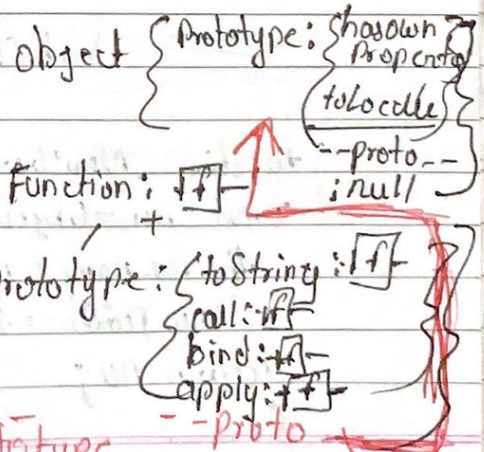
→ When we create a function, it is also a object, which has __proto__, linking to Object.prototype which is created at the runtime by JS, which has a prototype property. obj that has the methods like {toLocaleString(), toString(), hasOwnProperty()}

✱ When we do functionName.prototype.funtionCreated then we interclude the chain, & that [Name] function links its __proto__ to the Object .prototype



✱ When we run JS at runtime JS does two thing it creates a Object.prototype & Function: f + obj. combo, the fun^n obj combo has methods like,

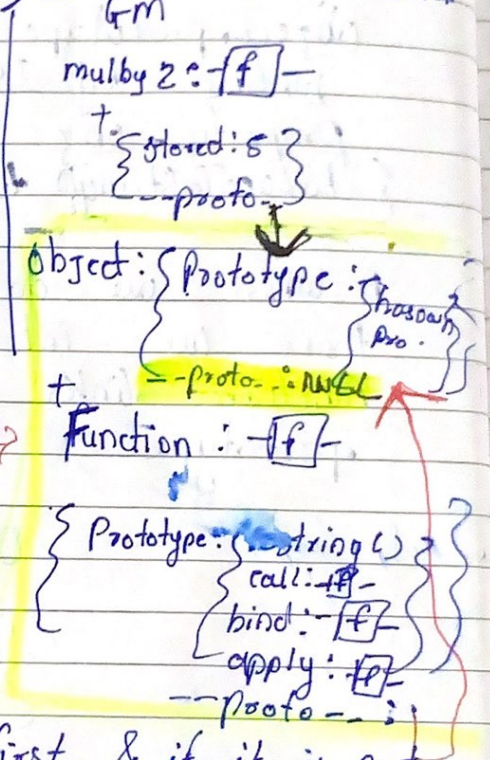
So when we call any thing on function, it checks the fun^n Prototype prop, if it not there it goes up to object.prototype prop.,




```
fun mul (num{ num=5 };
mul.toString() // In fun Prototype
mul.hasOwnProp("num"); // In obj Prototype
```

★ the function is both a function and object combo, so at the runtime; if we have declared a function then the JS at runtime adds (Function & Object) which links to that fun^

```
// function mulby2(num)
    return num*2;

★ mulby2.stored = 5
★ mulby2(5) //10
★ mulby2.stored //5
★ mulby2.prototype
```



this → part is created at runtime by JS,

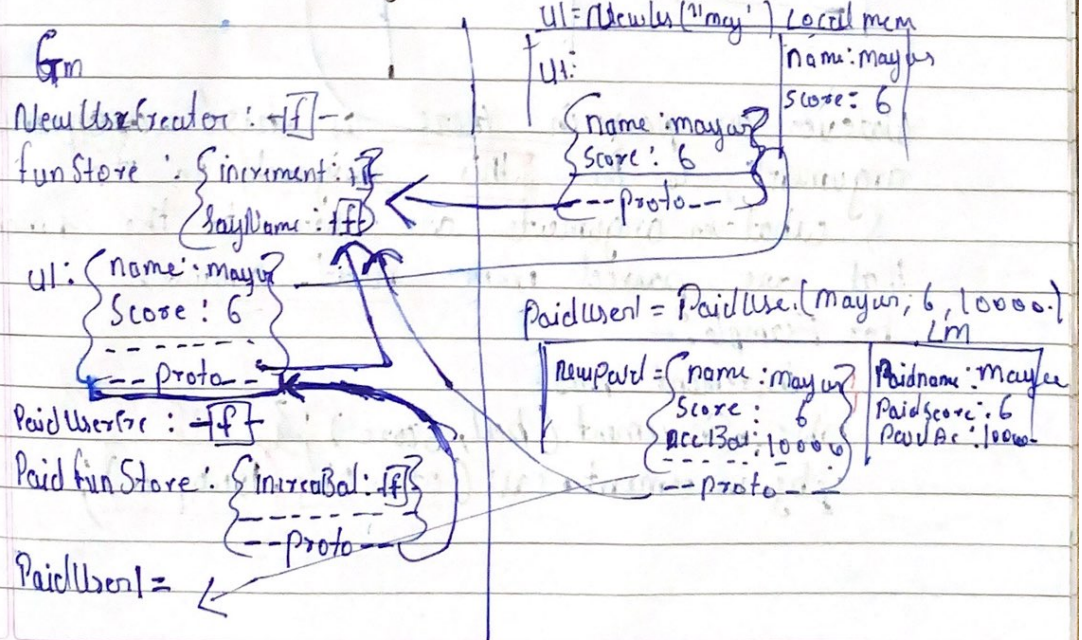
★ When we call the fun^ (.) property, then it looks in to the fun^.prototype first, & if it is not there then it continues to look through its __proto__ property i.e referencing to the Object.prototype

Sab Classing — JS — 801

```
function NewUser(creator Iname, $core) {
  const nu = Object.create(funStore);
  nu.name = name;
  nu.$coren = $core;
  return nu;
}
```

Sol 2 => Subclassing

```
const funStore = {
increment : function() { this.store ++ };
sayName : function() { "Hi!" + this.name };

const u1 = NewUserCreator("mayur", 6);
   u1.increment();
;

function PaidUserCr(PaidName, PaidScore, PaidAccountBal) {
  const NewPaidUser = NewUserCreator(PaidName, PaidScore);
  Object.setPrototypeOf(newPaidUser, PaidfunStore);
  NewPaidUser.AccountBal = PaidAccountBal;
}
const PaidfunStore = {
  increaseBal : function() { this.accountBal++; }
Object.setPrototypeof(PaidfunStore, UserfunStore);
   const PaidUser1 = PaidUserCr("mayur", 6, 100000);
      PaidUser1.increaseBal();
      PaidUser1.sayName();
```



Using Call,

```
const obj = {
  num: 3;
  increment: function() { this.name++ };
}
const otherObj = { num: 10 }
obj.increment();
otherobj.
obj.increment.call(otherobj);
```

Gobal memory

obj : { num: 3 } { increment: f }

otherObj = { num: 10 } { Prototype } --proto--

Function: f

{ Prototype: { call: f, Apply: f, Bind: f }.

obj.increment();

this.num++;	Local mem
obj.num++;	this: obj ①

obj.increment.call(otherobj);

this.num++	this: otherObj
otherobj++;	

Whatever we pass in there as first argument, is the this assigned to it.

& whatever arguments are there to the function that are passed from next commas.

for Example:-

increment has

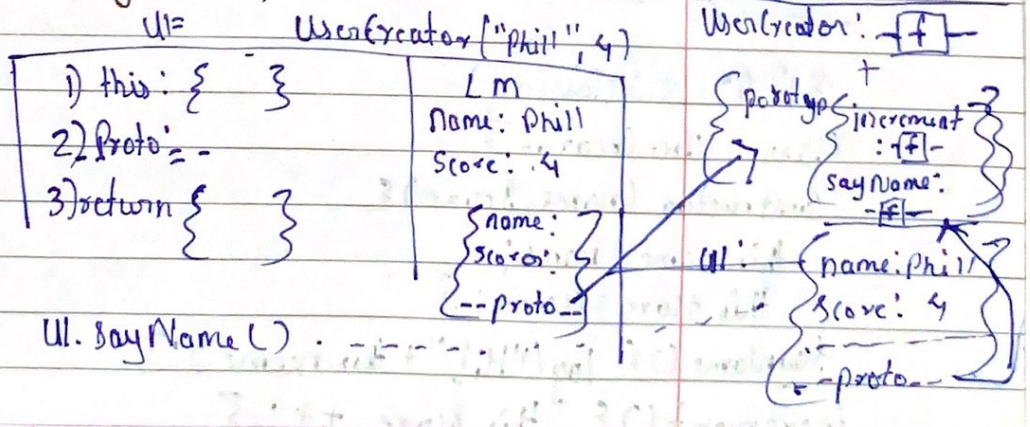
obj = { increment(bal, store) }; arguments

obj.increment.call(otherObj, 1000, 6);

this assigned

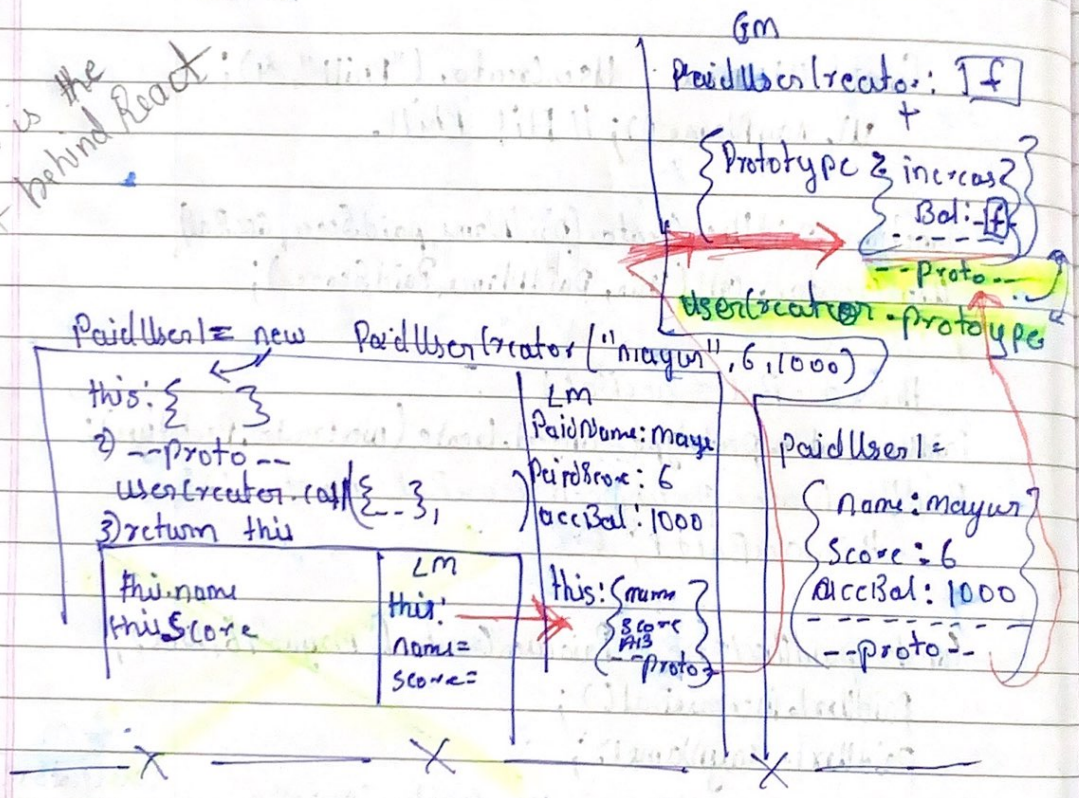
Soln 3 → subclassing

```
function UserCreator(name, score) {
  this.name = name;
  this.score = score; }

UserCreator.prototype.increment = function() {
  this.score++; }
UserCreator.prototype.sayName = function() {
  return "Hi! " + this.name; }

const u1 = new UserCreator("Phill", 4);
u1.sayName(); // Hi! Phill.

function PaidUserCreator(paidName, paidScore, accBal) {
  UserCreator.call(this, paidName, paidScore);
  // UserCreator.apply(this, [paidName, paidScore]);
  this.accBal = accBal; }
PaidUserCreator.prototype = Object.create(UserCreator.prototype);
PaidUserCreator.prototype.increaseBal = function() {
  this.accBal++; }

const paidUsr1 = new PaidUserCreator("Mayur", 6, 1000);
paidUsr1.increaseBal();
paidUsr1.sayName();
```



* Whenever we create with new keyword
1) the empty { } obj with this pointing
2) adds the --proto-- (if we interlude that it will
3) returns the { } point or otherwise it will point to Function.prototype → that links further to Object.prototype & its --proto-- is Null. (No further chain)

This is the concept behind React

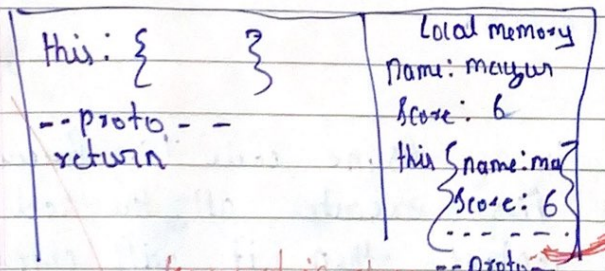


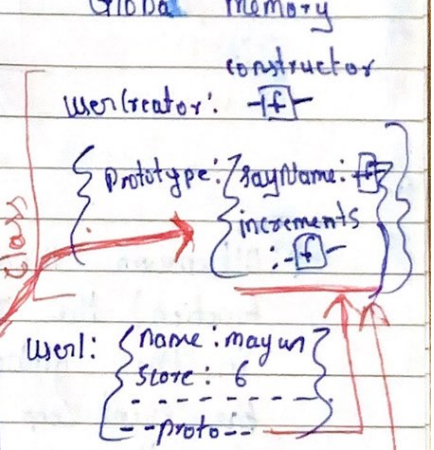
Soln 4:- Subclassing :-

```
Class UserCreator {
constructor (name, score) {
  this.name = name;
  this.score = score; }
sayName() { log("Hi! " + this.name ); }
increment() { this.score ++; }
}
```

```
Const u1 = UserCreator("Mayur", 6);
u1.sayName();

Class PaidUserCreator extends UserCreator {
  constructor(PaidName, PaidScore, AccBal) {
    super(PaidName, PaidScore);
    this.accBal = accBal;
  }
  increaseBal() { this.accBal++ };
}
Const PaidUser1 = new PaidUserCreator("Mayur", 6, 1000);
PaidUser1.increaseBal();
PaidUser1.sayName();
```

User1 = new UserCreator(Mayur, 6)





The new keyword in class makes the link to the Prototype Property of userCreator obj part.

PaidUser1 = new PUC("mayur", 6, 1000);

this: super(mayur, 6)
Reflect.super(userCrea, [May, 6], PUC]
this = new use(mayur, 6)