Object Orientated Programing

Why OOP? - an enormously popular paradigm for structuring our complex code.

→ easy to add features & functionality

→ Easy for us & other developers - A clear structure

→ performant. (efficient in terms of memory)

we need to organise our code as it gets more complex

---

1 let's suppose we're building a quiz game with users

Some of the users

name : phil
Score : 4

Name : Julia
Score : 5

functionality ⇒ ~~to~~ ability to increase score.

what would be the best way to store this data & functionality? ⇒ Objects

Objects - Store functions with their associated data!

1st method of creating object

```
const user1 = {
    name: "phil",
    score: 4
    increment: function() {
        user1.score++;
    }
};
user1.increment();
```

This is the principle of encapsulation
↓
put appropriate functionality & appropriate data & bundle them up

---

2nd method of creating object - creating user2 using 'dot notation'

```
const user2 = { }   // Create an empty object
user2.name = "Julia";   // assign properties to that object
user2.score = 5;
user2.increment = function() {
    user2.score++;
};
```

3rd method using Object.create.

```
const user3 = Object.create(null);
user3.name = "Eva"
user3.increment = function() {
    user3.score++;
};
```

this will give us some extra features if we pass any obj instead of null.

→ our code is getting repetitive, we're breaking our DRY principle & suppose we have millions of users, what could we do.

4 th method wrapping repeated code in a function.

```
function usercreator (name, score) {
        const newUser = { };
        newUser.name = name;
        newUser.score = score;
        newUser.increment = function () {
            newUser.score++;
        };
        return newUser;
}
const user1 = usercreator ("Phil", 4);
const user2 = usercreator ("Julia", 5);
```

* user1.increment();

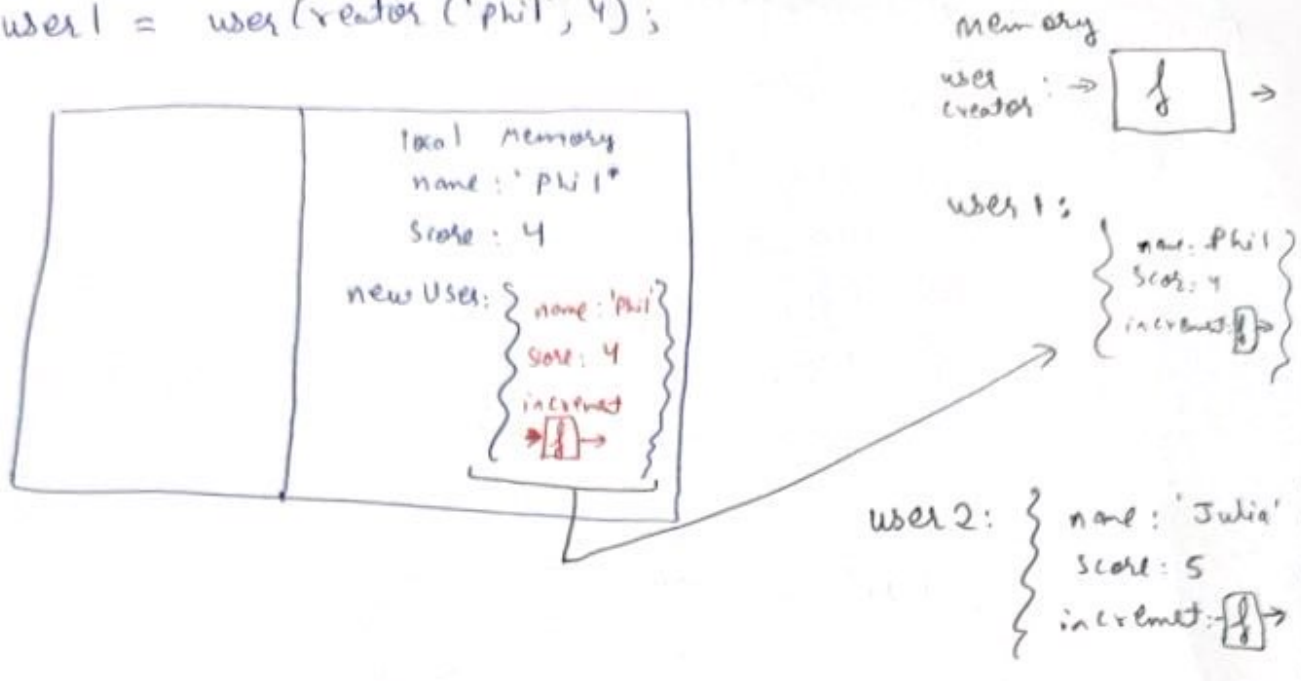
---

in both cases a copy of increment function on user1 & on user2

* issues in this approach

1. easy to proceed with this approch but difficult to add features.
2. fundamentally wrong because of memory issues.

user1 = user(reator ('phil', 4);



---

* increment method is stored individualy.

  copies of identical functions on every single object.

---

* instead of storing increment fn individually & we should have it some where else

---

Problem - Each time we create a new user we make space in our computer's memory for all our data & functions. But our functions are just copies.
is there a better way?

Solution - Store the increment function in just one object & have the interpreter, if it doesn't find the function on user1, look up to that object to check if it's there.

how to make this link? (proto chain)

```
const functionStore = {
    increment : function() {this.score++;},
    login : function() { console.log ("you are logged in")}

};

const user1 = {
    name : "phil",
    score : 4

}
user1.name // name is a property of user1 object
user1.increment // Error! increment is not!
```

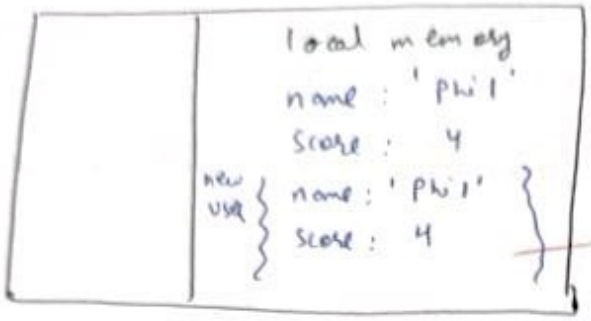
link user1 & functionStore so the interpreter, on not finding .increment, makes sure to check up in functionStore

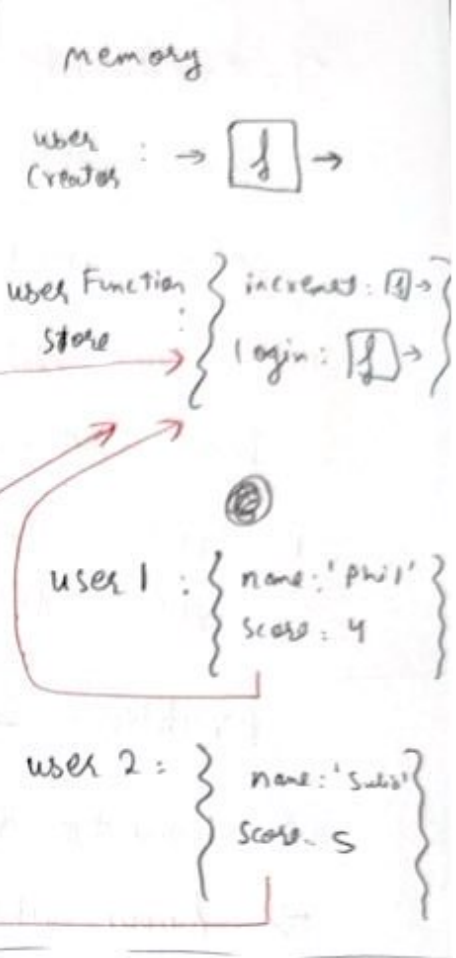
where it would find it.

* Object.Create

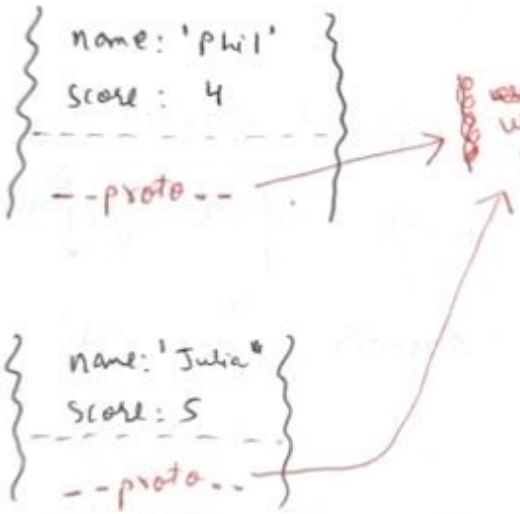
```
function userCreator (name, score) {
    const newUser = Object.create (userFunctionStore);
    newUser.name = name;
    newUser.score = score;
    return newUser;
};

const userFunctionStore = {
    increment: function() { this.score++; }
    login: function() { console.log("you're logged in"); }
}

const user1 = userCreator ("phil", 4);
const user2 = userCreator ("Julia, 5);
user1.increment();
```

user1 = user Creator ('phil', 4);





* user1.increment();



--proto-- ⟶ obj
(happened because of Object.create(obj))

Introducing the keyword that automates the hard work: new

```
const user1 = new User(creator("phil", 4));
```

when we call the constructor function with new in front we automate 2 things.

1) create a new user object.
2) return the new user object.

---

problems → ?

→ how to refer the auto-created object? (this)

→ know where to put our single copies of functions?

Interlude - functions are both objects & functions.

```
function multiplyBy2(num){
    return num * 2;
}

multiplyBy2.stored = 5;
multiplyBy2(3); // works
multiplyBy2.prototype = { };
```

Memory

multiplyBy2 : → [f] → + { stored = 5, prototype : { } }

when I call any function with new keyword, it makes me my object automatically bond with fn.prototype.

* next Solution

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

UserCreator.prototype.increment = function(){
    this.score++;
};

UserCreator.prototype.login = function(){
    console.
}

const user1 = new UserCreator("Eva", 9);

user1.increment();
```

memory

UserCreator: → [f] →

+

Prototype: { increment: [f], login: [f] }

user1 =

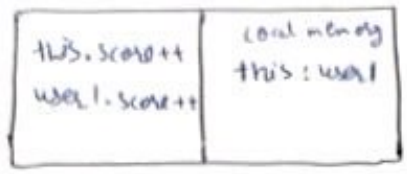
1. this: { }	name: 'Eva'
2. --proto--	score: 9
3. return	this: { name: 'Eva', score: 9, --proto-- }

red → new

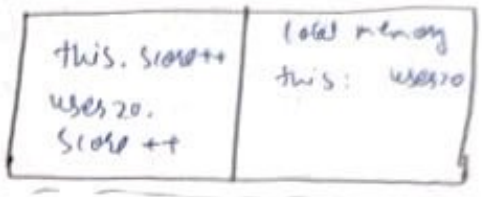
data & functionality bundled together — encapsulation

→ no strict encapsulation, means all of the properties are public by default.

user1.increment();



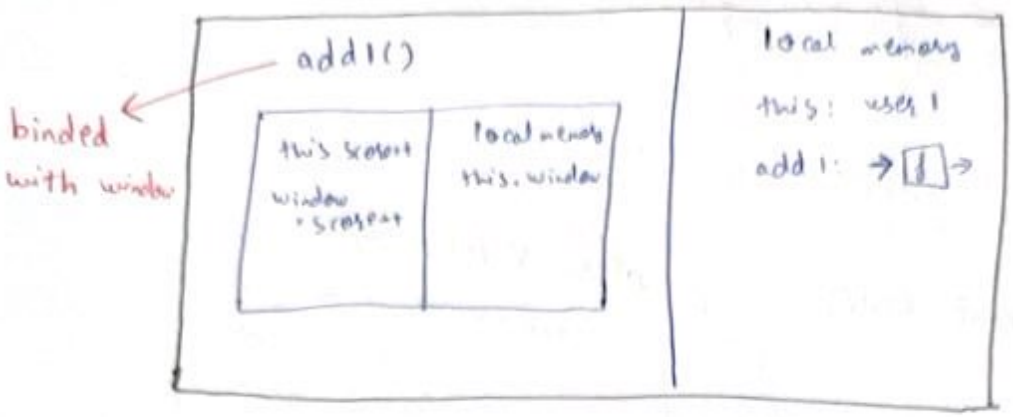
user2.increment();



what if we want to organise our code inside one of our shared functions — perhaps by defining a new inner function.

```
function User(reator (name, score) {
        this.name = name;
        this.score = score;
}
User(reator.prototype.increment = function() {
        function add1() {
                this.score++;
        }
        add1();
};
User(reator.prototype.login = function() {
        console.log("login");
};
const user1 = new User(reator ("Eva", 9);
        user1.increment();
```

user1. increment() :



① st (use call, bind & apply to fix this)

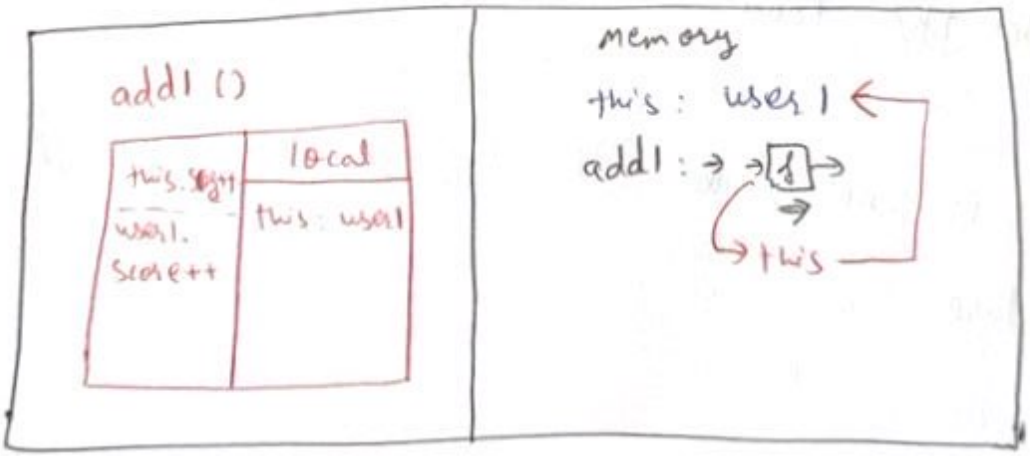
or

② nd (use arrow function inside ~~[illegible]~~ object methods)

```
User(reator. prototype. increment = function () {
    const add1 =() => { this . score++}
    add1();
} ;
```

---

user1. increment();



* class → → [f] → + {prototype}

constructor

The class 'Syntactic Sugar' :-

```
function User(creator(name, score){
    this.name = name;
    this.score = score;
}
User(creator.prototype.increment = function(){
    this.score;
};
User(creator.prototype. login = function(){
    console.log("login");
}

const user1 = new User(creator("Eva", 9);
user1.increment();
```

```
class User(creator {
    constructor (name, score){
        this.name = name;
        this.score = score;
    }
    increment(){
        this.score++;
    }
    login(){
        console.log("login");
    }
}
const user1 = new User(creator("Eva", 9
user1.increment();
```

under the hood | Syntactic Sugar

→ J.S. is not an OOP language it's prototypical in nature

Benefits of class in J.S.

- Emerging as a new standard
- feels more like style of other languages

Problem -

- 99% of developers have no idea how it works & therefore fail interviews.

J.S. uses this proto link to give objects, functions & arrays a bunch of bonus functionality. All objects by default have --proto--.

```
const obj = {
    num : 3
}
```

→ obj.num // 3

~~obj.hasOwnProperty~~

→ obj.hasOwnProperty("num") // ? where's this method ?

→ Object.prototype // { hasOwnProperty : Function }

→ with Object.create we override the default --proto-- reference to Object.prototype & replace with functionStore

→ but functionStore is an object so it has a --proto-- reference to Object.prototype - we just intercede in the chain

obj. num // 3

obj. hasOwnProperty('num')

memory

obj: { num : 3 }
{ --proto-- }
→ Object.prototype

Function: [f]
+
{ prototype: { --proto-- } }

Object: → [f] →
+
{ prototype: → { hasOwnProperty : f } }

* Arrays & functions are also objects so they get access to all the functions in Object.prototype but also more goodies.

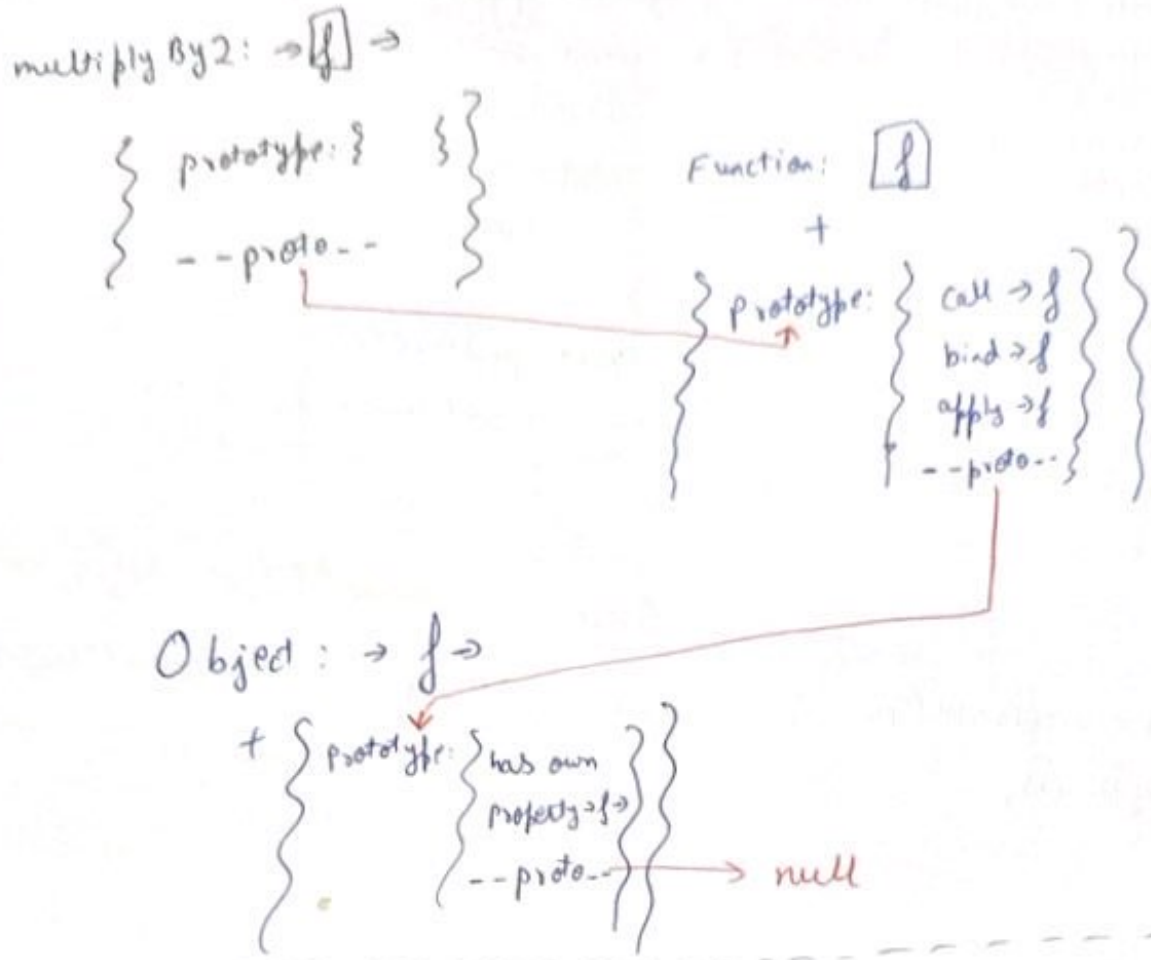
```
function multiplyBy2(num){
    return num * 2;
}
```

multiplyBy2.call() // where is this method?

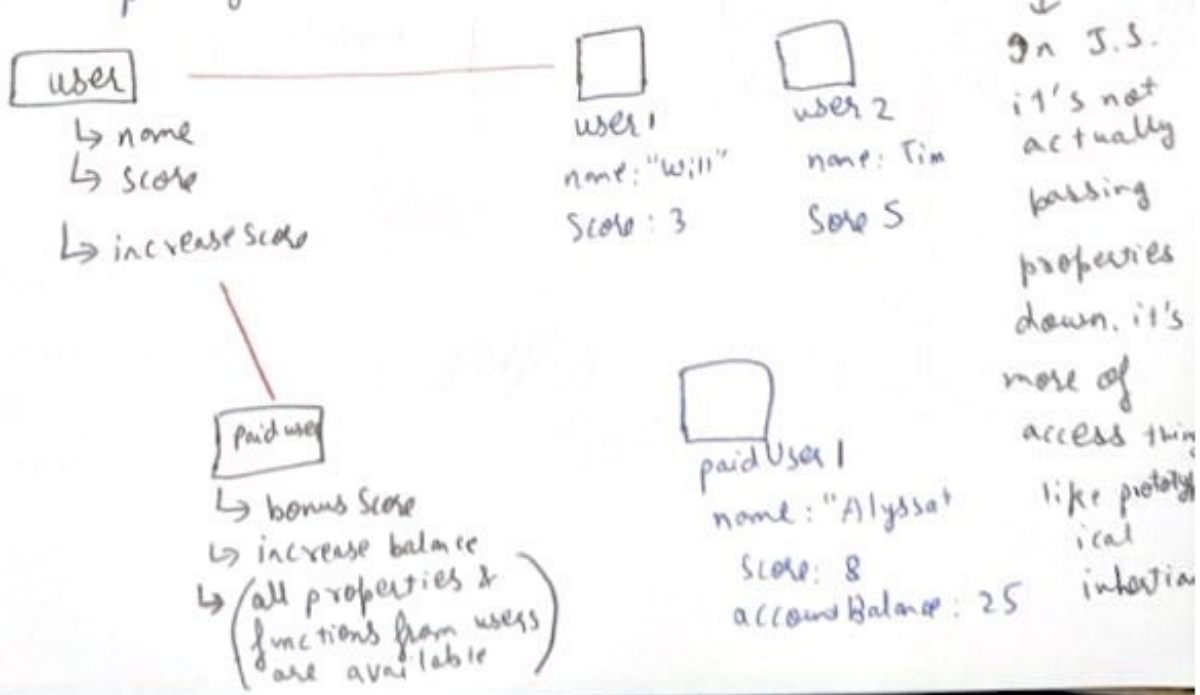
Function.prototype // { call → f , bind → f }

multiplyBy2.hasOwnProperty("score") // where's this function?

Function.prototype.--proto-- // Object.prototype
{ hasOwnProperty : f }



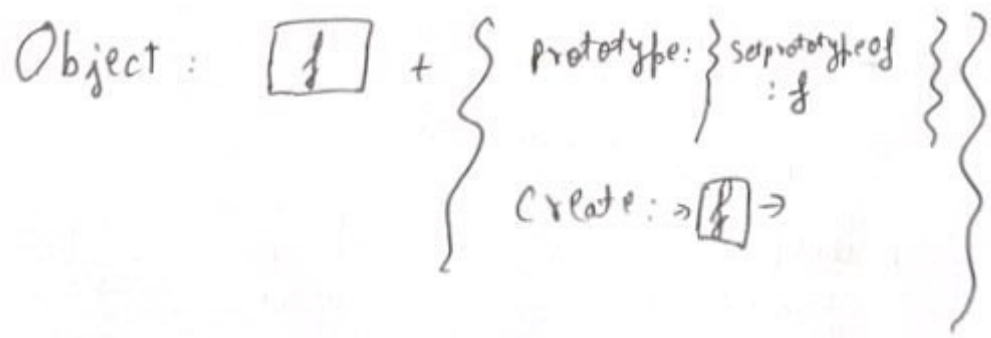
Subclassing – A core of an OOP approach is inheritance – passing knowledge down.



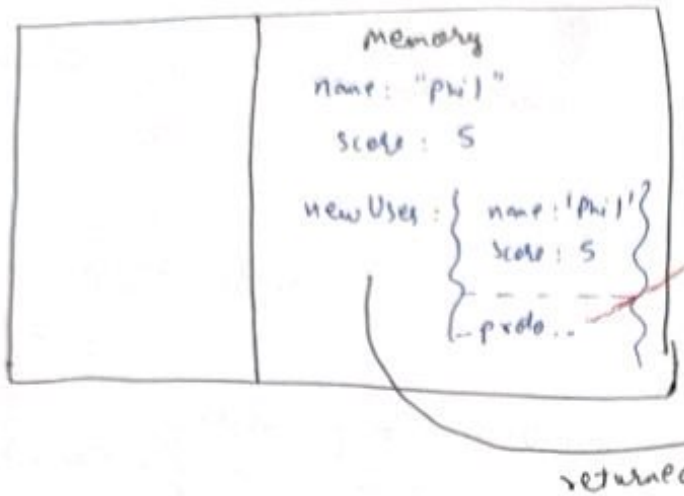
Sub Classing in factory function approach

```
function user(creator (name, score)){
    const newUser = Object.create(userFns);
    newUser.name = name;
    newUser.score = score;
    return newUser;
}

userFns = {
    SayName: function(){
        console.log("I am" + this.name);
    }
    increment : function(){
        this.score++;
    }
}

const user1 = user(creator("phil", 5);
user1.sayName();
```

```
function paidUser(creator( paidName, paidScore, accountBalance){
    const ~~newUser~~ newPaidUser = user(creator(paidName, paidScore);
    Object.setPrototypeOf (newPaidUser, paidUserFns);
    newPaidUser.accountBalance = accountBalance;
    return newPaidUser;
}

const paidUserFns = {
    increaseBalance : function (){
        this.accountBalance++;
    }
};
Object.setPrototypeOf (paidUserFns, userFns);

const paidUser1 = paidUser(creator("Aly", 8, 25);
paidUser1.increaseBalance();
paidUser1.sayName();
```

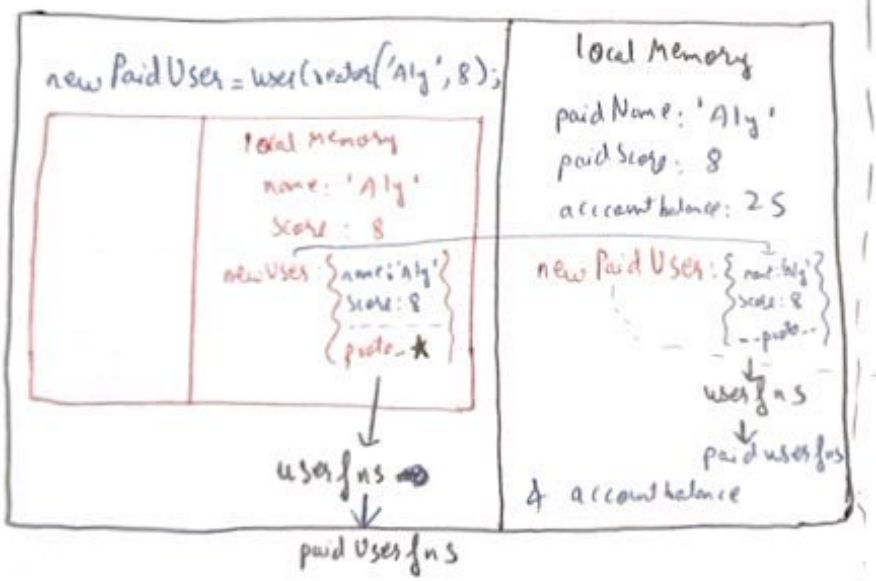


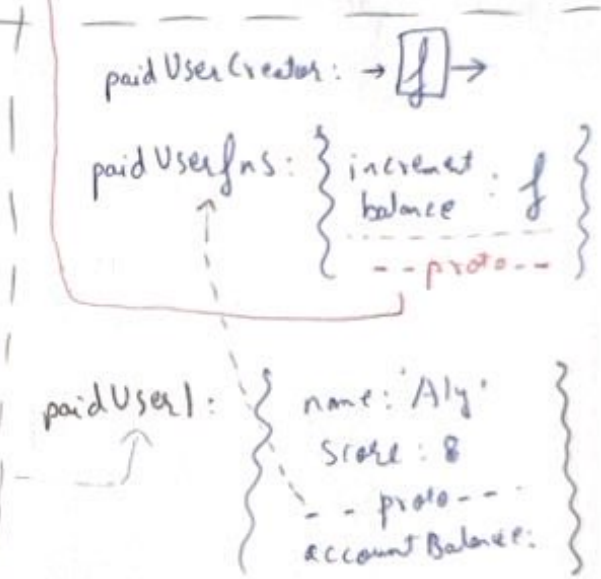
user1 = user(creator('Phil', 5))



→ user1.sayName();

paidUser1 = paidUserCreator('Aly', 8, 25)





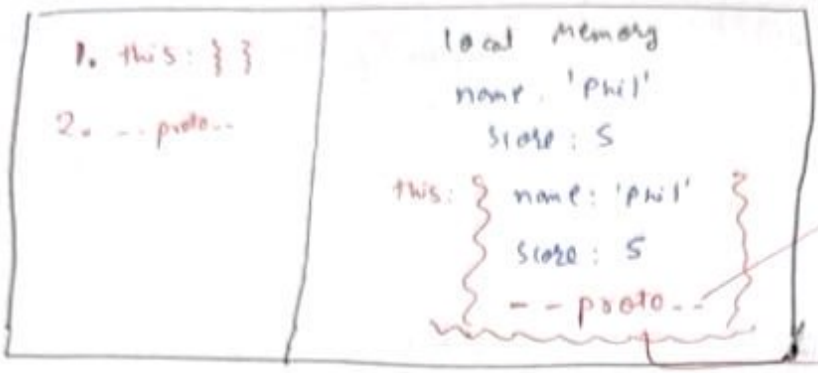
paidUser.increaseBalance();

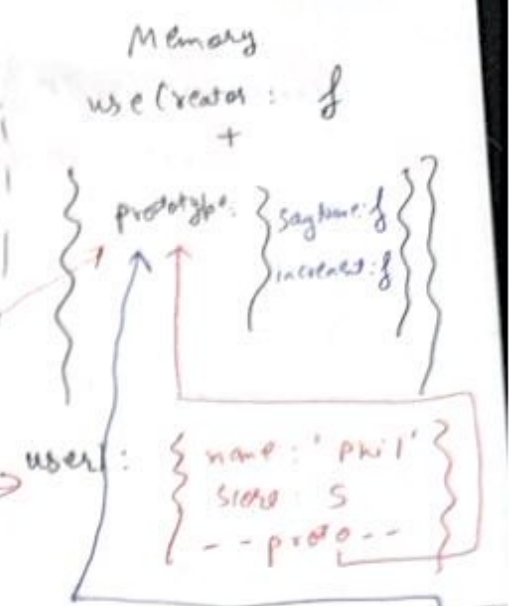
paidUser1.sayName();

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

user(reator.prototype.sayName = function() {
    Console.log("I'm" + this.name);
}

user(reator.prototype.increment = function() {
    this.score++;
}

const user1 = new user(reator("Phil", 5);
const user2 = new user(reator("Tim", 4);

user1.sayName();
```

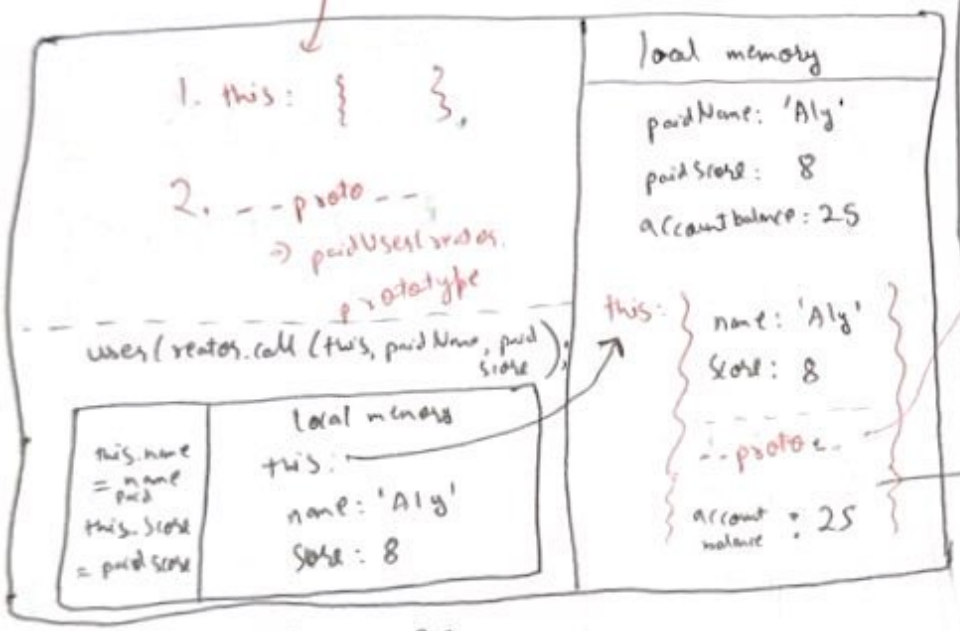
```
function paidUser(reator(paidName, paidScore, ab) {
    ~~user(reator.call(paidName, paidScore);~~
    ~~this~~

    user(reator.call(this, paidName, paidScore);
    this.accountBalance = ab;
}

paidUser(reator.prototype = Object.create
        (user(reator.prototype);

paidUser(reator.prototype.
    increaseBalance = function () {
    this.accountBalance++;
}

const paidUser1 = new paidUser
        creator("Aly", 8, 25);

paidUser1.increaseBalance();

paidUser1.sayName();
```

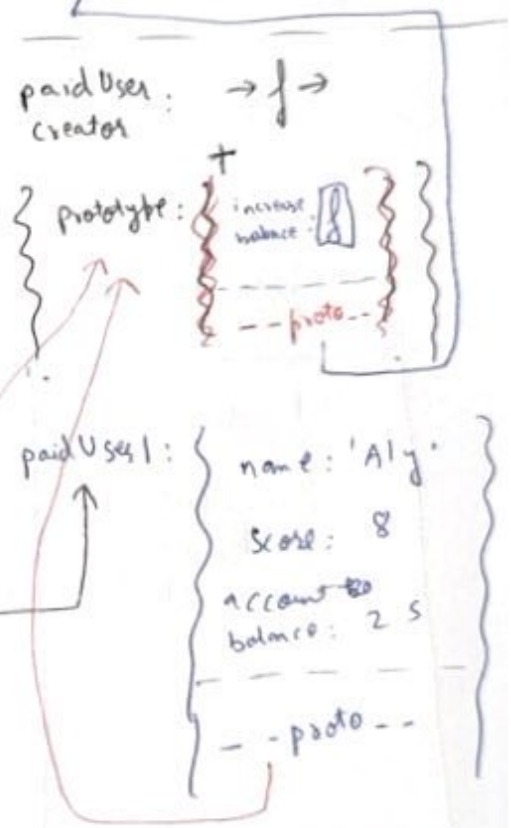
user 1 = new user(creator('Phil', 5));





paidUser1 = new paidUser(creator('Aly', 8, 25));



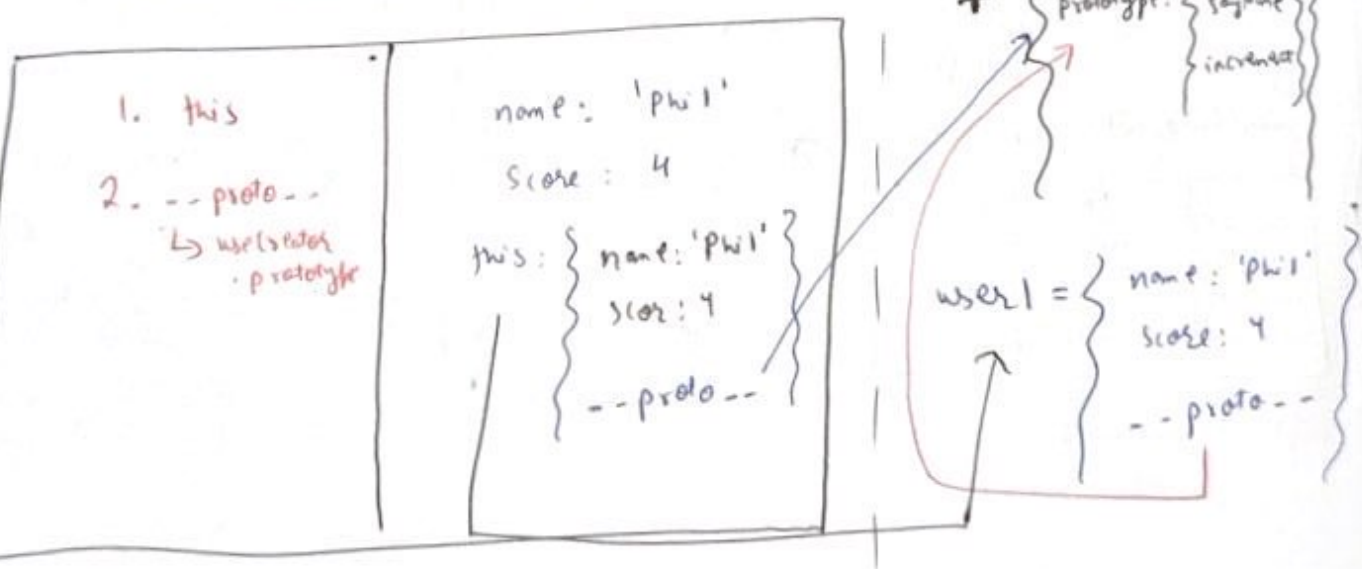


* this.accountBalance = 25

```
class user(reator {
    constructor (name, score)}
        this.name = name;
        this.score = score;
    }
  sayName()}
    console.log("I am" + this.name);
  }
  increment()}
    this.score++;
  }
}
```

```
class paidUser(reator extends user(reator)
{
  constructor (paidName, paid Score, accountBalance)
  { super (paidName, paid Score);
    this.accountBalance = account Balance;
  }
  increaseBalance()}
    this.accountBalance++;
  }
}
```

```
user1 = new user(reator('phil',4);
```



paidUser1 = new paidUser(creator('Aly', 8, 25));

* super('Aly', 8) return of super

~~reflect.construct~~

reflect.construct(user creator, ['Aly', 8], paid user creator)

this

this = new user(creator('Aly', 8));

↳ --proto-- is changed from user creator to paidUser creator

1. this: {}	local memory
2. --proto--	name: 'Aly'
3. return	score: 8
	this: { name: 'Aly' score: 8 --proto-- }

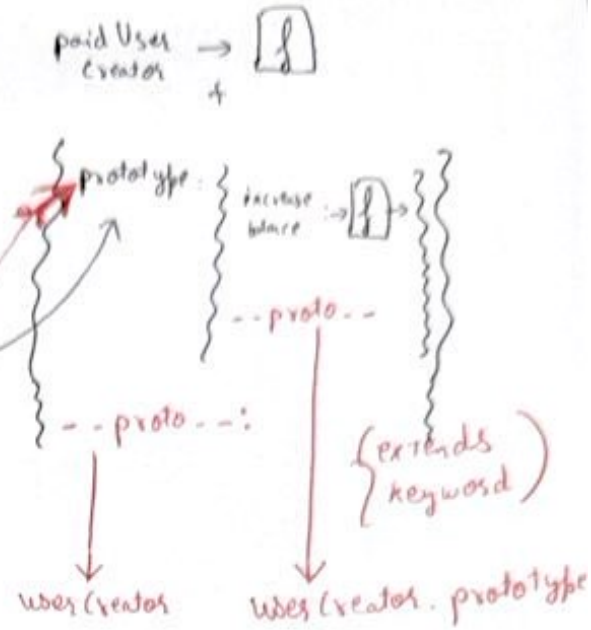
local Memory

paidName: 'Aly'

paidScor: 8

account balance: 25

this: { name: 'Aly' score: 8 --proto-- account balance: 25 }



* 'this' is uninitialized & we must have super call, 'this' is there, it's uninitialized but we cannot refer to it.

* Super must be called first because the new memory will be created in user creator. constructor function.

extends job

① paid User creator. prototype --proto--
⇒ user creator. prototype

② paid User. --proto--
⇒ user creator.

② nd step ensures super keyword has access to constructor of user creator

paid User 1 = { name: 'Aly' score: 8 account balance: 25 --proto-- }

↓

paid User creator. prototype