→ (Execution context in JS)

8. The 'this' argument values:-

* There are two default arguments to every function call: 'arguments' and "this."

(i) function foo() {
    console.log("Hello"); console.log(this) → window reference
}
foo(); // method #1

→ In this case value for this parameter is the global object — (window)

(ii) var obj = { };
obj.foo = function() {
    console.log("Hello"); // Hello
    console.log(this); // object { obj & foo: obj.foo() }
};
obj.foo(); // method

method 2:→ Calling functions as property of an object referee

this referee :- The object referee .
↳ in above example — obj

(iii) new foo();

method 3:- Calling standalone functions using 'new' keyword

'this' referee :- The newly created object

function foo() {
    // var this = { };
    console.log("Hello"); // Hello
    console.log(this); // object { } - empty object
    // return this;
}

new foo();

9. working on objects with 'this' reference :-

Ex.2

function Bicycle(cadence, speed, gear, tirepressure) {
    this.cadence = cadence;
    this.speed = speed;
    this.gear = gear;
    this.tirepressure = tirepressure;
    this.inflateTires = function() {
        this.tirepressure += 3;
    }
}

var bicycle1 = new Bicycle(50, 20, 4, 25); // tirep = 25
bicycle1.inflateTires(); → Tirepressure = 28

var bicycle2 = new Bicycle(50, 20, 4, 30); → tp = 30
bicycle2.inflateTires(); tp = 33

bicycle1.tirepressure = 28;
bicycle2.tirepress = 33

'this' referee for above two function are different even though one of them inside another.

~~this~~ Every function could have different this reference.

1. using the call function:-

```
function Bicycle(cadence, speed, gear, tirepressure) {
    this.cadence = cadence;
    this.speed = speed;
    this.gear = gear;
    this.tirepressure = tirepressure;
    this.inflateTires = function() {
        this.inflateTires + 3 = 3;
    }
}
var bicycle1 = new Bicycle(50, 20, 4, 25);

function Mechanic(name) {
    this.name = name;
}

var mike = new Mechanic("mike");
mike.inflateTires = bicycle1.inflateTires;
mike.inflateTires(); // NaN (undefined + 3);
```

this → mike instance

Object None will be cbr method name

console.log(this) → Mechanic { name: "mike", inflateTires: f, tirepressure: NaN }

⇒ Method-4

```
function Foo() { }
Foo.call();
```

A function in JS is an object and that object can have properties turns out every function is object ie JS has the a property which happens to another function.

That property is call.

Foo.call(); Calls the function like Foo();

Foo. this 'call()' property takes an argument that could be an object. whatever object we will pass in call property it takes that object and binds this ~~object with the call object~~ to the 'this' reference ~~of~~ on function Foo.

Ex:

```
function Foo() { this.abc = def; }
Foo.call(bicycle1);
```

bicycle 1

bicycle1.inflateTires(); ⇒ Tirepressure = 29.

```
var bicycle2 = new Bicycle(50, 20, 4, 30);
bicycle2.inflateTires(); TP → 33
```

```
function Mechanic(name) {
    this.name = name;
}
var mike = new Mechanic("mike");
mike.inflateTires = bicycle1.inflateTires;
mike.inflateTires(); // tirepressure = NaN;
```

✓ mike.inflateTires.call(bicycle1);
bicycle1.Tirepressure; // ⇒ 31

✓ mike.inflatTires.call(bicycle2);
bicycle2.Tirepressure; // 36.

11. When constructors are not good enough :-

* Javascript objects don't "own" methods. They just have properties and any property can be a function.

```
function Bicycle(cadance, speed)
{
    this.cadance = cadance;
    this.speed = speed;
    this.inflatespeed = function() {
        this.speed += 10;
    }
}

var bicycle1 = new Bicycle(50, 40);
var bicycle2 = new Bicycle(50, 80);
```

If we create objects ~~of~~ from Bicycle method than every bicycle object will have inflatespeed.

→ Suppose we are creating a employee management system and we need to create 1000 employee that if we do like above method then all the employee object will have method.

actually method ~~should have~~ should be in every object it should be at a common place so every object can access it.

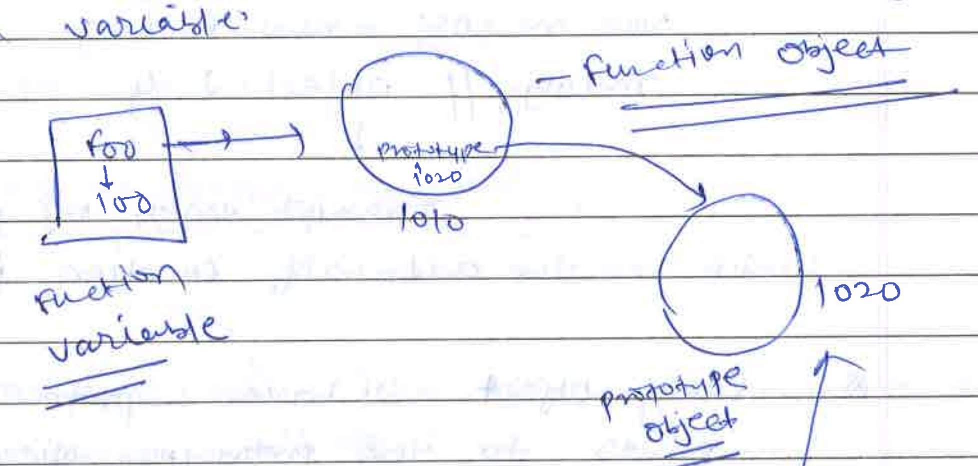
* for this reason concept of prototypes comes into picture.

* There is a new 'class' keyword in the newer version of javascript (ES6) that simulates class-like behaviour. But Javascript does not have the class concept.

12. Introducing the prototypes:-

```
function foo() { }
```

When Javascript processes function it creates an object whose address hold by function variable.

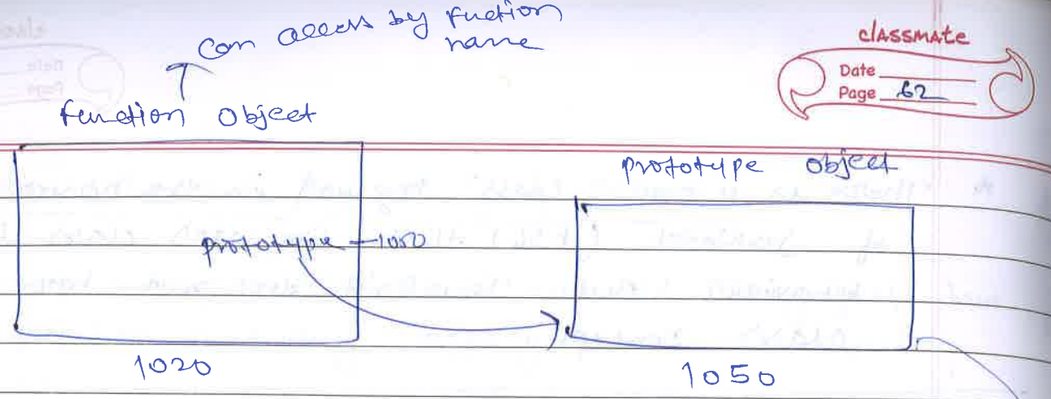


console.log(foo); // function foo()

→ actually it creates two object
(i) function object
(ii) prototype object

* function object have one property prototype which holds the address of prototype object.

console.log(foo.prototype); // Object { , + more ... }



* When JS process any fuction it allway creates these above two object.

```
function foo() { }

var myObj = new foo();
myobj; // object 2 {}
```

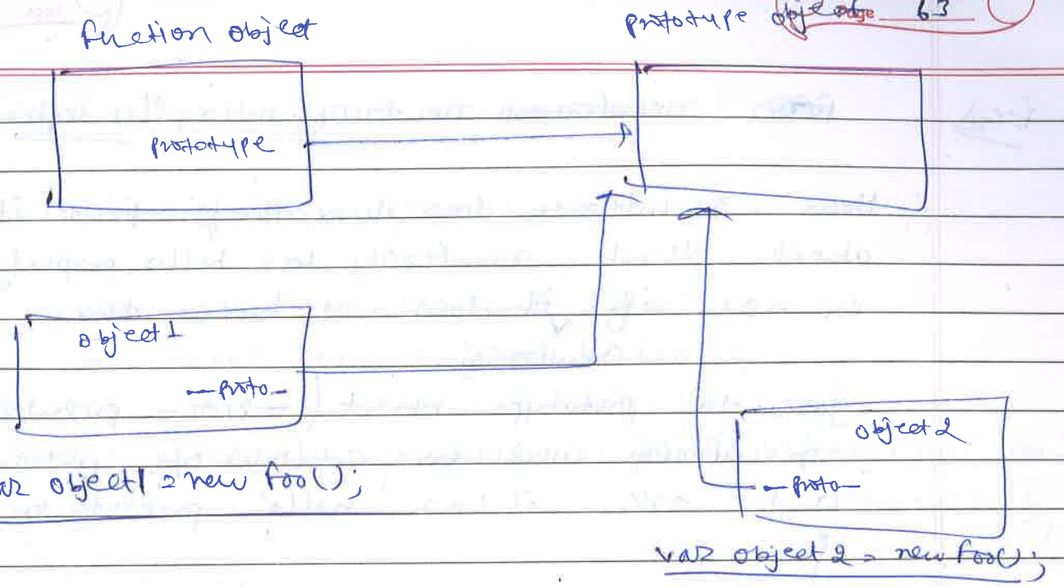
Javascript assign one property __proto__ which has the address of fuction prototype object.

* Every object will have __proto__ property which points to the prototype object of respective scope.

myobj; // object 2 {}
↓
This will have __proto__ which hold the prototype object of foo fuction

for every function two objects will be created (i) function object (ii) prototype object

every object will have under proto: '__proto__' wether it is created by a function or normal creation



13. Property lookup with prototypes:

Ex:

```
function foo() { }

var newfooObj = new foo();
```

foo.prototype.test = "This is the prototype object of fo";
foo.proptype.test // "___"
newfooObj.__proto__.test // "___"

Ex3 ~~foo,~~ ~~newfooobj~~ newFooObj.hello; // undefined

Here JS interpreter does two things first it will check that newFooObj has hello property or not if it does not have than, it newFooObj goes to prototype object (—proto— property of newFooObj which has address of prototype object) and ask it has 'hello' property or not.

from

```
{ newFooObj.__proto__.hello = "prototype object";
  newFooObj.hello; // "prototype object" }
```

```
{ foo.prototype.abc = "abc";
  newFooObj.abc // abc }
```

```
{ foo.prototype.hello; // "prototype object" }
```

newfooobj.test; // "This is the prototype object"
↓
its inside —proto—

Now

```
newfooobj.test = 10;
newFooObj.test; // 10
```

delete newFooObj.test // true

newFooObj.test; // "This is the prototype object"

14. Object behaviours using prototypes:—

```
function Employee(name) { this.name = name; }

var empl = new Employee("Jim");
var emp2 = new Employee("Pam");

{ Employee.prototype.playPranks = function(){ console.log("Prank Played!!"); }; }
```

assigned a fuction to the prototype object of Employee of function.

→ Now how may objects we will create with this Employee fuction with new keyword. that will have this fuction.

```
→ Employee.prototype.playPranks(); // Prank Played!
→ empl.playPranks(); // Prank Played!
→ emp2.playPranks(); // Prank Played!
```

```
var emp3 = new Employee("Dwight");
emp.playPranks(); // Play Played!
```

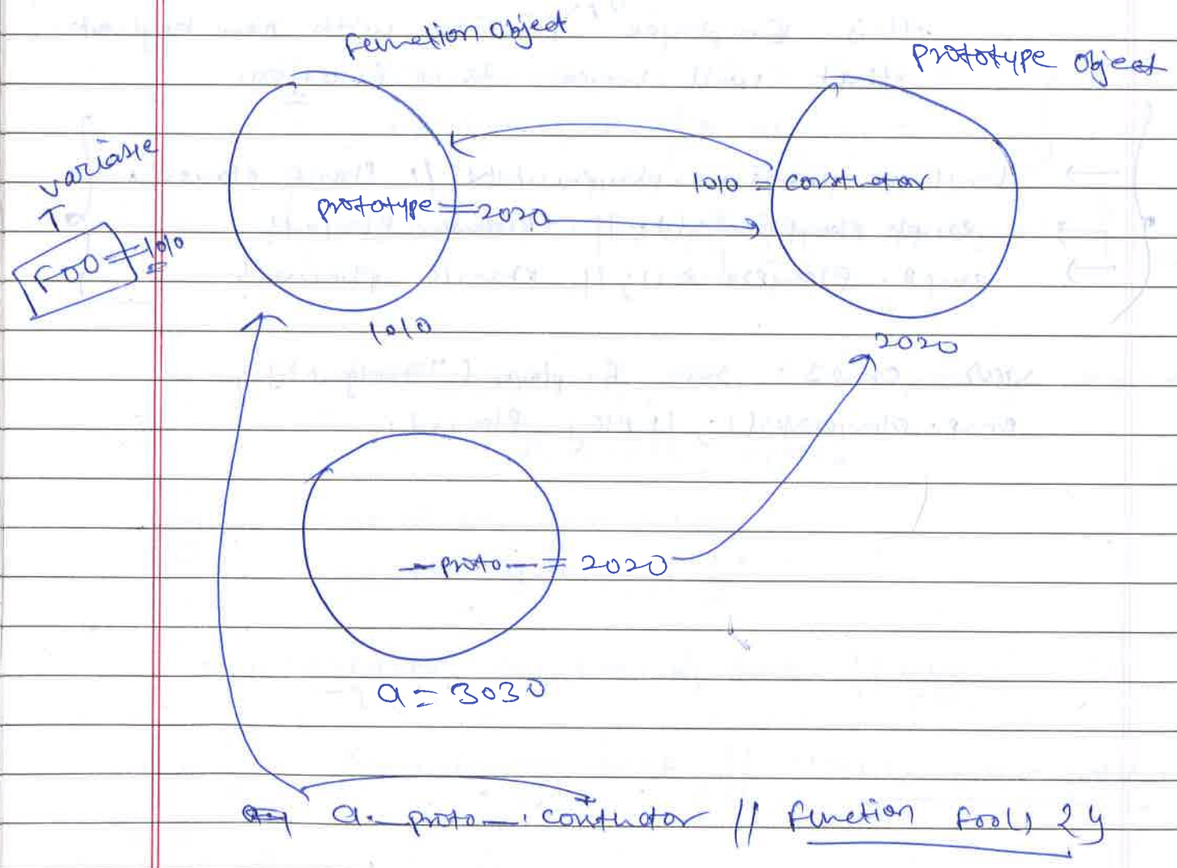
15

Objects links with prototypes:-

function foo() { }

```
var a = new foo();
a || object { }
    → __proto__ → dunder proto
a.__proto__
```

The double-underscore are reffered to as "dunder" ~~double underscore suffix~~ So, this property is called dunderproto.




```
var b = new a.__proto__.constructor();
b || object { }
```

=> we should not use '__proto__' dunder proto.
=> Instead of this we should use prototype property of function object to set some methods or property on prototype object.

```
foo.prototype.greet() = function () {
    console.log("Hello");
};
```

```
var myObj = new foo();
```

```
myObj.__proto__.testjy = function () {
    console.log("Testjy");
};
```

=> we should not use '__proto__' dunder proto.
not recommended

Ex:

function foo() { };

var obj1 = new foo();

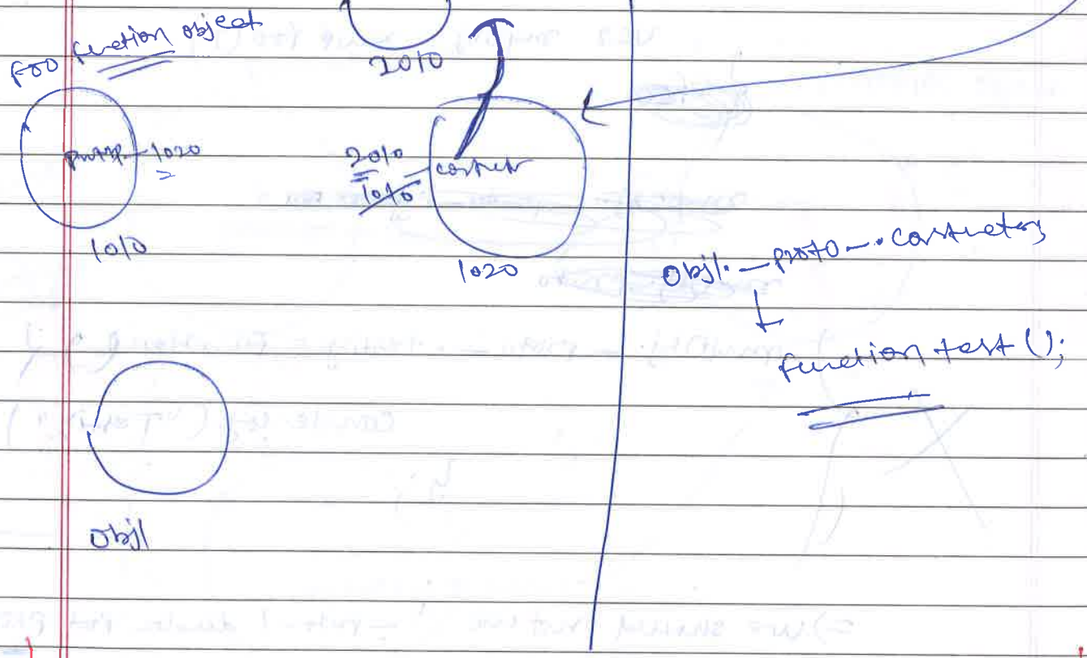
obj1.__proto__.constructor; // function foo() { }

obj1.__proto__.constructor = function test() { };

var obj2 = new obj1.__proto__.constructor();

obj2 // Object { }

obj2.__proto__.constructor; // function test() { }



Conclusion:- __proto__, prototype, constructor these all are actually reference and we can modify them.

→ what we have done above is just to show that that are references not recommended to use.

16. The Global function

Object function:-

In case of browser (window)

→ In JS runtime environment just like we have global object like that we also have global functions. and one of those global functions is called 'Object'

name of that function is 'Object' but type is 'function'.

→ this 'Object' function is object also beoz in javascript every function is a object as well.

Console.log(Object); // function Object() { }

Object(); // Object { }

Creating empty object:-

(i) var simple = { };

(ii) var obj = new Object();

These two are same

(i) is shortcut actualy internaly is calls new Object(); to create fuction

simple === obj ✗

simple; // Object { }

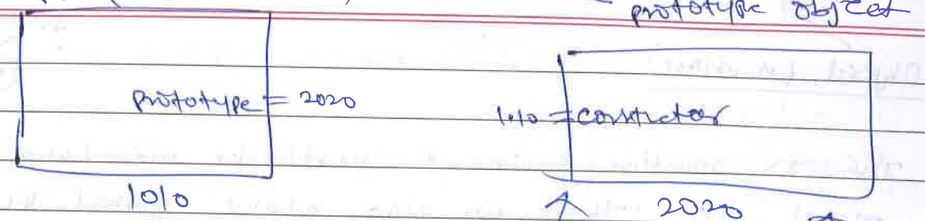
obj; // Object { }

Prove that (i) and (ii) are same

obj.__proto__ === simple.__proto__

simple.__proto__ === Object.prototype

function object for (object function) → global function

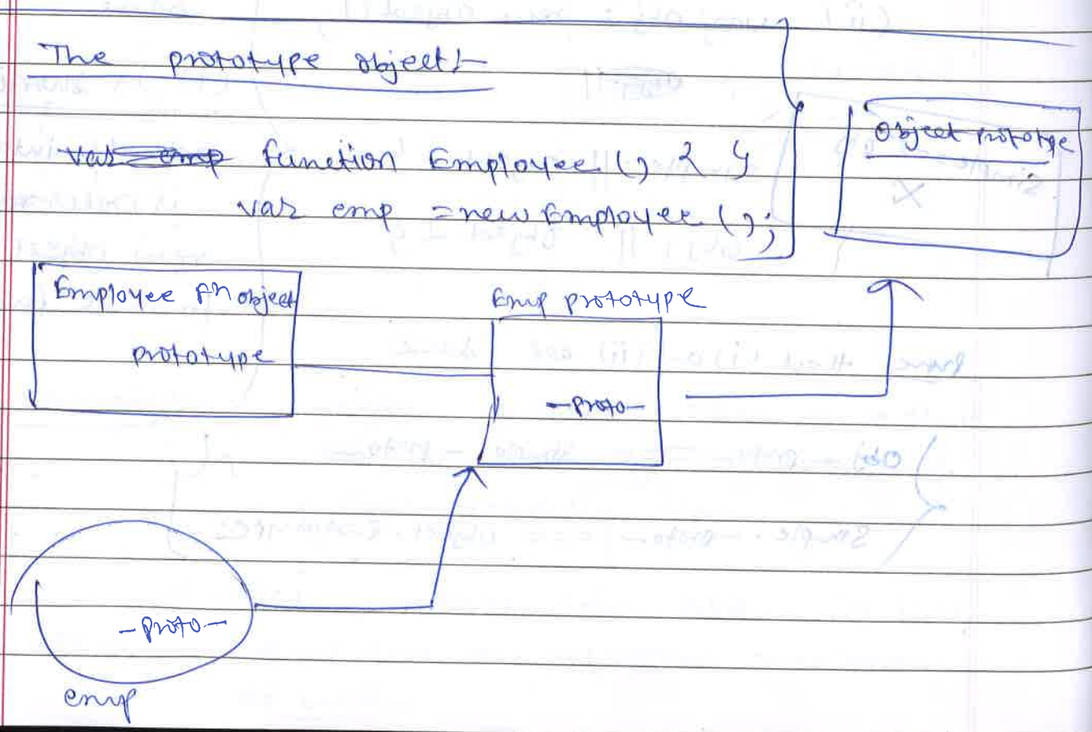


(i) var simple = 2;

(ii) var obj = new object();

(i) and (ii) are same, we are creating object through 'new object();' in same way.

17. The prototype object:-



* The automatically created prototype object is actually created using 'new object()'

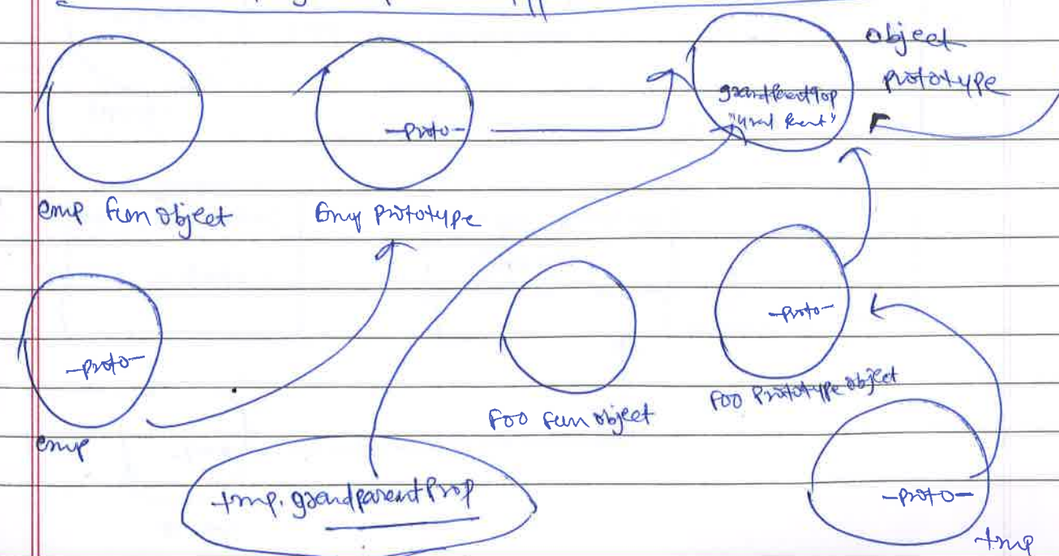
```
{ emp.prop = "Employee";
  emp.prop || Employee }
```

```
{ emp.__proto__.parentProp = "parent of Employee";
  emp.parentProp || "parent of Employee" }
```

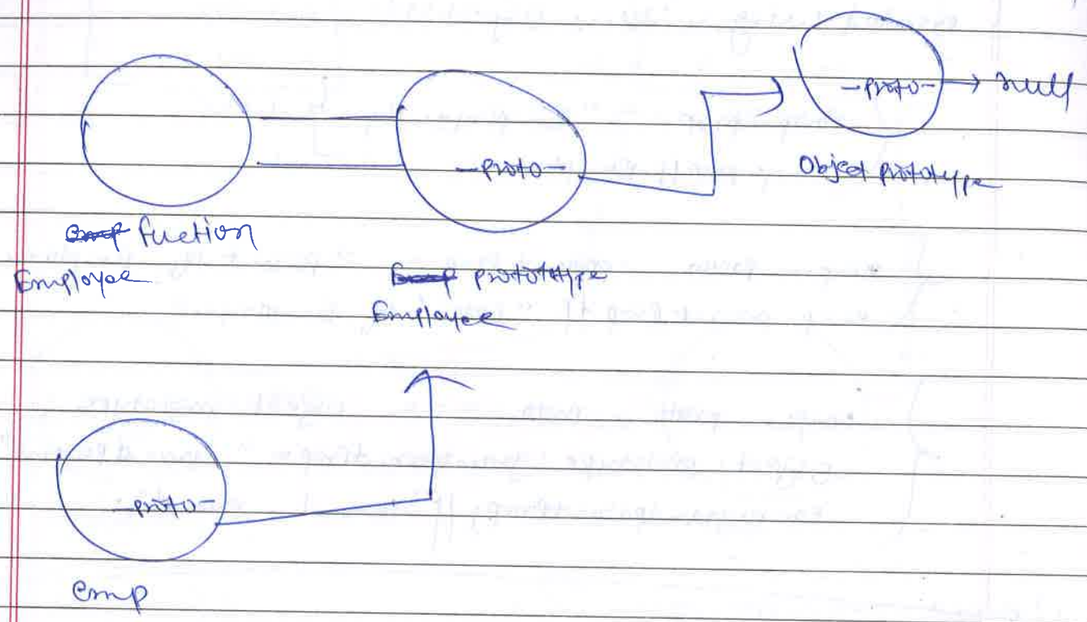
```
{ emp.__proto__.__proto__ === Object.prototype;
  Object.prototype.grantparentProp = "Grandparent";
  emp.grandparentProp; || "Grand parent". }
```

* If we add any property in Object prototype (Object.prototype) we essentially adding that property to every object in our system.

Ex: function Foo() {}
var tmp = new Foo();
tmp.grantparentProp || "Grand Parent".



∞ Object prototype also has __proto__ that points to null



18'

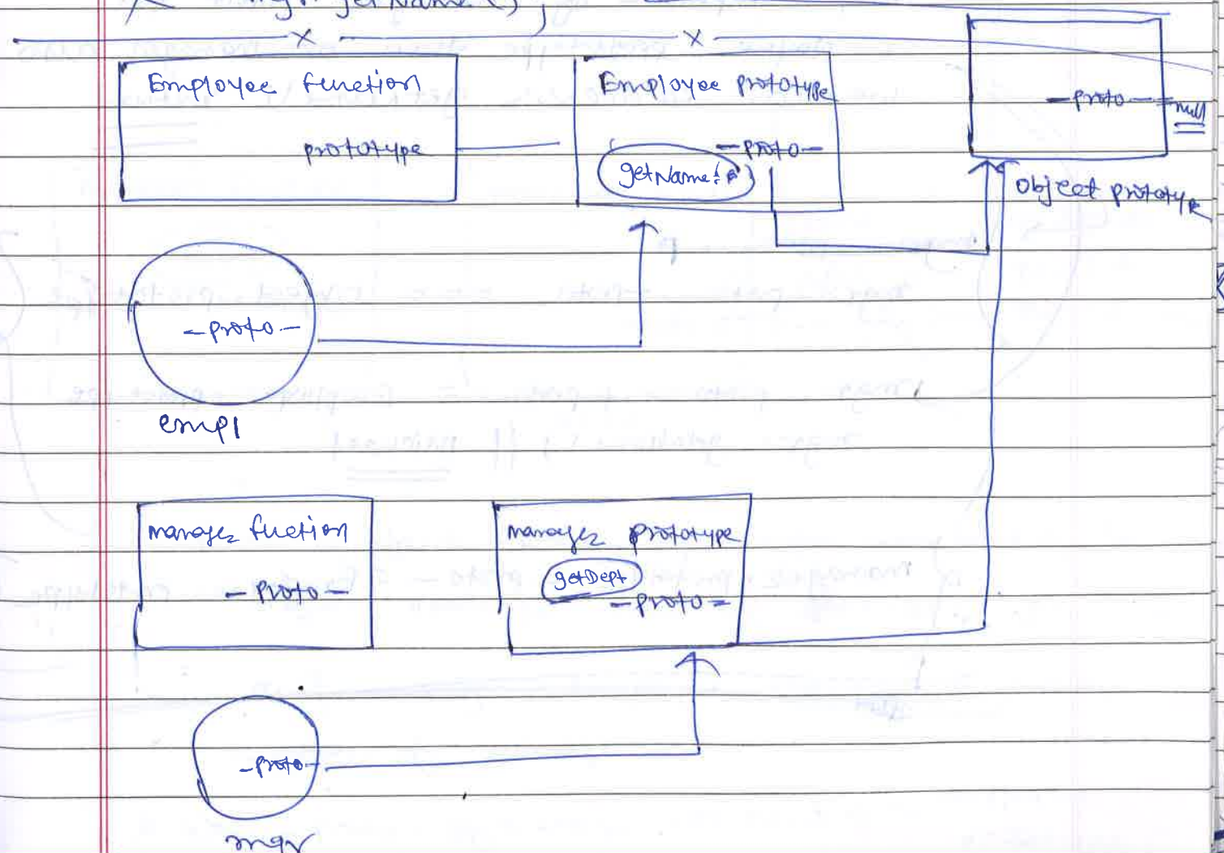
Inheritence in Javascript:-

```
function Employee(name) { this.name = name; }
Employee.prototype.getName = function() { return this.name; }
var empl = new Employee("Jim");
empl.getName(); // "Jim"

function Manager(name, dept) {
    this.name = name;
    this.dept = dept;
}

Manager.prototype.getDept = function() { return this.dept; }
var mgr = new Manager("Michael", "Sales");
mgr.getDept(); // "Sales"
```

✗ mgr.getName(); → is available on Employee.prototype



above example Manager is also a employee. for employee getName() method is available and this is available in Employee prototype.

This getName() should be accessible by ~~manager~~ Manager because manager is also a employee.

Method-1 → keep getName() method in Object prototype but this is bad because than getName() will be available to every object.

Method-2:

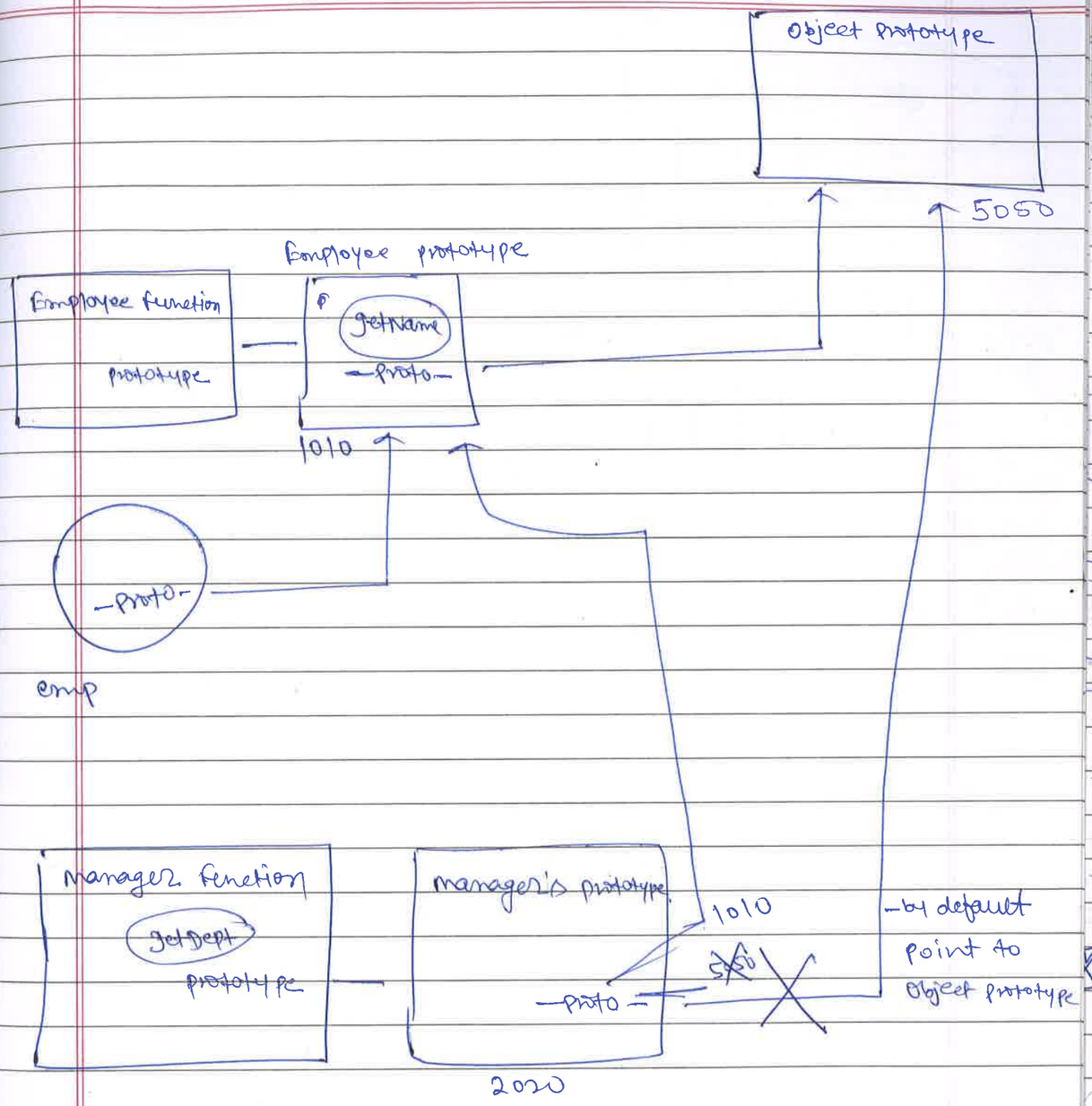
Currently under proto (—proto—) property of manager is pointing to Object prototype. if we change this default behaviour and —proto— of manager point to employee prototype than on manager also we can access getName() method

mgr. —proto— . —proto— === Object. prototype

mgr. —proto— . —proto— = Employee. prototype

mgr. getName() || Michael

manager. prototype. —proto— = Employee. prototype



Here by default —proto— of manager was pointing to object prototype now we change the Pointer and assign Employee prototype Object to —proto— of manage.

* Now —proto— of manage is pointing to employee prototype.