- props and methods

```
var a = {
    name : "Kiran",
    class : "BTech",
    number : 123
    tasks : function(){ }
}
```

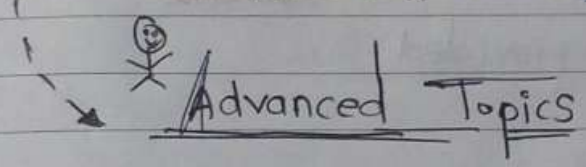
props

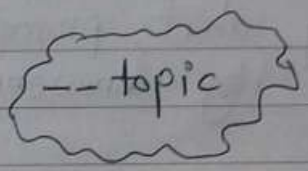
method

@iranshinde

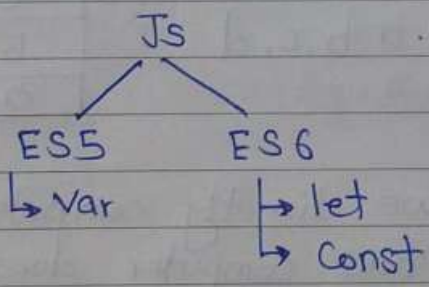
when a value of a property is a function, then it is method.

a.name = "Meow"; → updating..





-- difference betn var, let, const



∴ ES5 + ES6 = var, let, const

1) var old js me hai
2) var is function scoped

i) let, const new js me hai
ii) let, const are braces scoped.

e.g

```
function abcd(){
    for(var i=0; i<10; i++)
    {
        console.log(i);
    }
    console.log(i); → accessible
}
```

```
for(let i=0; i<10; i++)
{
    console.log(i);
}
console.log(i); → not accessible
```

3

var is accessed all over the fun^n.

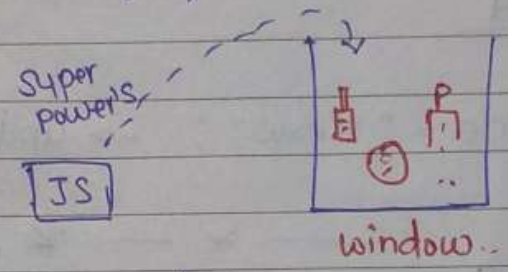
3) var adds itself to window object.

iii) let, const don't add themselves to window object.

Window object

IMP

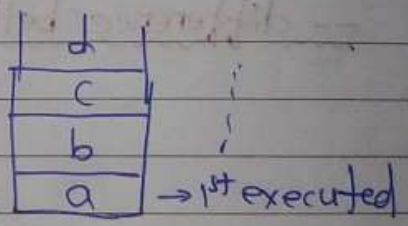
Js language me kai sare aaise features hai, jo hum use karte hai, par vo js me nahi hai...
Js vo sare features window object se leti hai..
→ Window is a box of features



e.g. alert, prompt are provided by browser..

-- browser context API

↳ window
↳ stack → a, b, c, d
↳ heap



1+2+3+4 = 10; we directly compute this, but how computer does it..

1+2 = 3 → store 3
3+3 = 6 → store 6
6+4 = 10 → print

} where, 3, 6 are stored during the execution ?

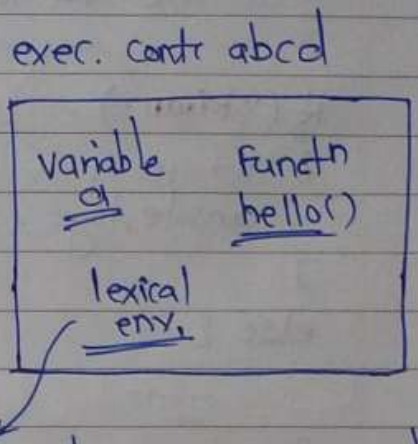
↓

in heap memory..

-- execution context

whenever a function is called, its execution context is created which store's the **variables**, **functions** and **lexical environment.**

```
Function abcd()
{
  var a = 10;
  Function hello(){
     var b;
  }
}
```



it tells who can access what?

e.g. Funn abcd has access to var a & hello()
but, it can't access var b; → function scope is the reason..

-- copy reference values

e.g. var a = [1, 2, 3, 4]
var b = a; ✗ wrong way

* using spread operator...

∴ var b = [...a] → this is how it is done..

e.g. var obj = { name: "meow" }

∴ var objCopy = {...obj}

-- truthy vs falsy

How to identify falsy ?

↓

0, false, undefined, NaN, null, document.all

↓

all the values other than this are truthy

```
if ("Kiran")
{
   console.log(..);
}
else {
   ——
   ——
}
```

```
if ( NaN )
{
   ——
}
else {
   ——
   ——
}
```

executed..

-- For Each

→ only applied on array's

e.g. var a = [1, 2, 3];

val

```
a.forEach ( function (val) {
      console.log( val * 2);
})
```

This never updates the original array..

This function run's for every element of array..

```
Function (val) {
   ——
   ——
}
```

↓↓

Anonymous function i.e. without name..

-- For in

→ only applied on objects

```
var a = {
   name = "Kiran",
   age = 22.
}
```

key's

```
For ( var key in a)
{
   console.log (a); -> accessing keys
}
```

if → a[key] → value

-- callback function's → Async Part

→ this is a function which run's after sometime, after finishing of some ~~tasks~~

e.g.
```
setTimeout ( function () {
    console.log ("callback");
}, 2000);
```
→ this function run's after 2 seconds..
1000 ms = 1 sec

-- First class Functions

↳ you can treat the functions as a variable..

i.e.
```
var a = function () {
    console.log ("Meow");
}

a();
```
o/p: Meow

(or) you can pass funn to fnc

```
function abcd (a) {
    a();
}

abcd ( function () { console.log ("Hello") });
```

-- how array's created behind the scene..

everything is object in js

```
var a = [1, 12, 15];
```
⇓
```
var a = {
    0: 1,
    1: 12,
    2: 15
}
```

```
a[-3] = 1;
a[-2] = 12 ..
```

```
typeof a;
```
o/p: 'object'

i.e

```
[1, 12, 15]
 ↓  ↓  ↓
-3 -2 -1
```
negative indices

```
: -3: 1,
  -2: 12,
```

```
delete a.age
```
→ name of property..

-- higher order function

↳ It is the function which accepts another function as an argument (OR) return's a function.

eg. abcd(function(){ console.log(...) });

```
function abcd(){
   return function(){
     };
  }
```

-- Constructor function

→ It is the type of normal function in which **this** keyword is used & while calling that function we use **new** in

```
function makeBiscuit(price)
 {
    this.width = 10;          this.price = price;
    this.height = 15;
    this.color = "Brown";
    this.taste = "Sugary";
  }

var Bis1 = new makeBiscuit();
    Bis2 =
     :
```

you can have many instances..

you can also pass parameters

-- new keyword

→ It always creates a blank object

-- iife immediately invoked function expression :-

It is the way to run a function immediately and create private variable inside it,

```
(function(){
  var a=12;
})()
```

↳ when you write this, it becomes iife

console.log(a) → not accessible..
only accessible within the func,
∴ you can return it.. try to return obj. & store in vari-

```
∴ var ans = (function(){
              var a=12;
              return { };})();
```

now, you can define methods in return

1. accessing

```
getter: function(){
   console.log(a);
}
```

2. setting value

```
setter: function(val){
   a = val;
}
```

```
∴ var ans = (function(){
      var a = 10;

      return {
        getter: function(){
           console.log(a);
        },
        setter: function(val){
           a = val;
        }
      };
})();
```

ans.getter();

-- Prototype

```
var obj = {
    name : "Kiran";
}
```

goto browser, console
type obj.
there, will be 'name' in suggestions
but also {[[prototype]]}

→ we don't created it then ? it comes with every object & provided by javascript.

Examples :- arr.length
property by js.

-- prototypical inheritance

→ accessing properties of another class../ function / object

e.g. there are humans who can do several tasks but also there are sheriyan's students who can do some of their own tasks plus all the tasks of human.

```
var human = {
    canTalk : true,
    canWalk : true,
    canFly : false
}
```

@kiranshinde

```
var sheriyanStudent = {
    createAwesomeWebs : true,
    createAwardedWebs : true
}
```

∴ sheriyanStudent.__Proto__ = human;
↓
contains all props of human,

∴ sheriyanStudent.canTalk;

-- this keyword

① in global scope

```
console.log(this);
```
↳ window

② in a function

```
function abcd()
{
    console.log(this);
}
```
↳ window

③ inside a method

```
var obj = {
    name : "Meow",
    getter : function(){
        console.log(this);
    }
}
```
↳ object..

④ eventlistener

```
var btn = document.
        qyerySel.("id
btn.addEventListener(
"      ", function(){
    console.log(this);
}
```
↓ element

-- call, apply, bind

if you have a function and a object. & you want to run that function but also you want that the by default value of this to be changed to object then we apply these methods.

call

```
function abcd(){
    console.log(this);
}
```
this.name ✓
↳ by default behavior is now changed..

```
var obj = { name : "meow" }

abcd.call(obj);
```

#apply when you want to send multiple values with obj.

```
function(val1, val2, val3) {
    console.log(this, val1, val2,);
}
var obj = {name : "Kiran"};
abcd.call(obj, 1,2,3); → ✗ wrong
```

↓

```
abcd.apply(obj, [1,2,3]); → Correct ✓
```

#bind

```
var bindfun = abcd.bind(obj);
bindfun();
```

↓

when you want to run it in future..

e.g. event listeners

-- pure functions

1. It should always return same output for same input
2. It will never change the value of global variable.

```
function abcd(a,b) {
    return a*b;
}

var ans1 = abcd(1,2); ✓
var ans2 = abcd(1,2); ✓
```

→ impure function's are having opposite behavior.