Null vs Undefined

Undefined → It is a type, undefined means variable declared but no value has been assigned to the variable.

```
eg: var a;
    console.log(a);
    // output : undefined
```

→ It is global object

→ value doesn't exist in the compiler

Null → absence of value. It is an object

When we define a variable to null then we are trying to convey that variable is empty.

is primitive data i.e., not an object & has no methods or properties

eg: string, no, null, symbol

```
eg: var a = null;
    console.log(a);
    // output = null
```

```
null == undefined   // output: true
null === undefined  // output: false
```

→ operator compares values as well as data types of operands

→ operator does type conversion of operands before comparison.

converting data of one type to another. eg) string to no.

typeof (null)

→ typeof is an operator & not a function, so, parantheses are not necessary, we can write like

```
typeof null;
//output: "object"
```

→ Unfortunately, in js, data type of null is object. we can consider it a bug in js that typeof null is an object

Void(0) in js

⇒ Js void 0 means returning undefined as a primitive value.

→ "Javascript: void(0)" in HTML document.
↳ It is used to prevent any side effects caused while inserting an expression in a webpage.

→ Void — Keyword in js — used as unary operator
↳ This operator specifies an expression to be evaluated without returning a value.

InnerText

- ignores spaces.
- used to print plain text information b/w tags
- can't insert HTML tags.
- It returns text without an inner element tag.

Inner.HTML

- considers spaces
- used to print content in HTML formats.
- can insert HTML tags.
- It returns a tag with an inner element tag.

eg)

```
<div id="test">
The following element contains
some <codes>code</codes>
and <italic>some text</italic>
</div>

<button onclick="innerTextFn()">
InnerText
</button>

<script>
function innerTextFn() {
var x = document.getElementById('test');
alert(x.innerText);
}
```

→ The following element contains some code and some text.

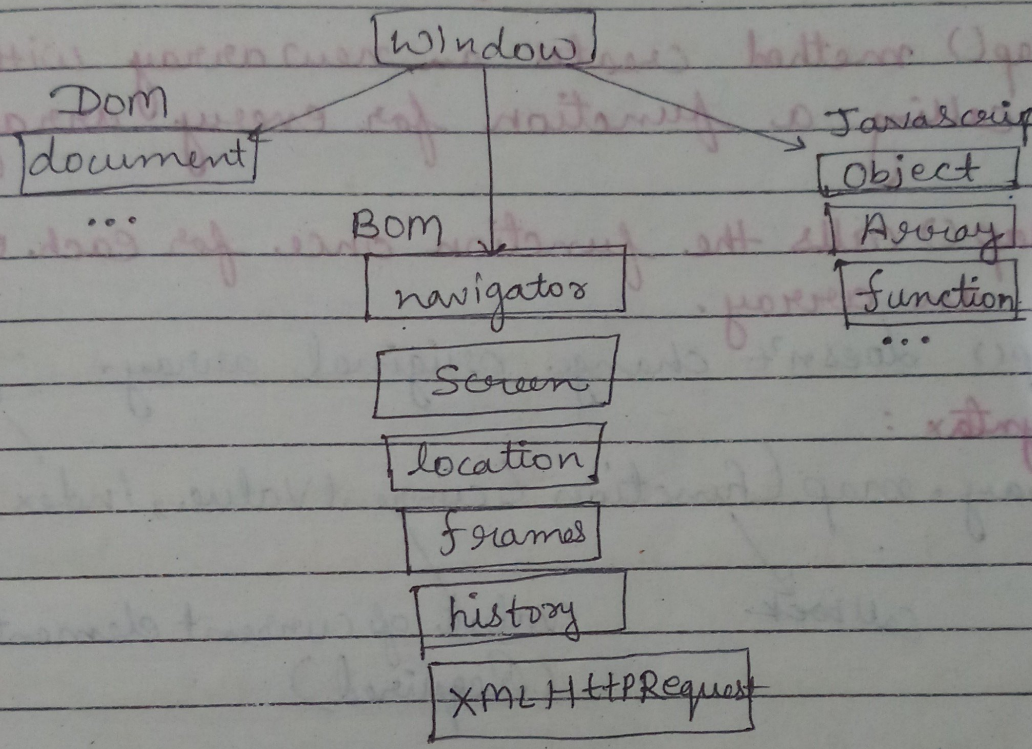
```
<button onclick="innerHTMLFn()">
innerHTML </button>

→ innerHTML() {
alert(x.innerHTML);
}
```

→ The following element contains some <codes>code</codes> and <italic>some text</italic>

BOM in JS.

Browser Object Model

→ allows js to talk to the browser about matters other than content of Page.



eg: Window object
↳ provides info of browser. Is window like inner, outer (Height & width).

location object
document.location === window.location
// true

Screen object
window.screen.

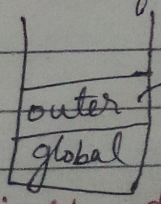
CLOSURES

→ Combination of function + lexical envt. in which function was created.

eg:
```
var i = 10;
function outer () {
    var j = 20;
    console.log (i, j);
    var inner = function () {          // function Expression
        var k = 30;
        console.log (j, k);
    }
    return inner;
}
var inner = outer ();      // calling outer funct.
inner ();                  // calling inner function.
```

→ It is a function having access to the parent scope (outer()), even after parent function has closed.

this inner() will return value of j = 20 bcz of closure despite of outer function has already been exit from the stack which shouldn't be happen.



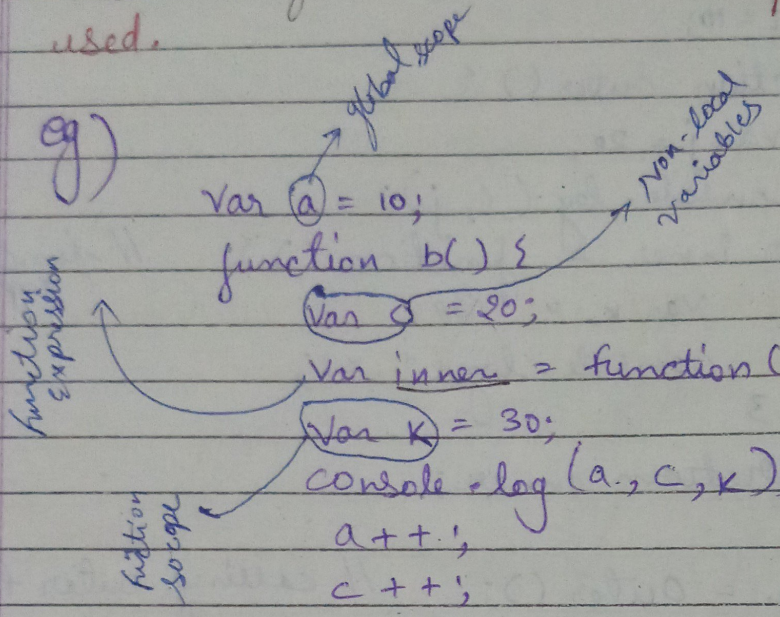
Here, whatever the variable defined bef inside the function Outer() they will not destroy or even after the completion/exit of Parent function.

memory will be reserved

uses of closure :-

① whenever a function is created within the function but it is accessible & called outside the main function then concept of closure is used.

eg)




```
Var a = 10;
function b() {
    Var c = 20;
    Var inner = function () {
        Var K = 30;
        console.log(a, c, K);
        a++;
        c++;
        K++;
    }
    return inner;
}
Var inner = ~~outer~~b();
inner();
inner();
```

Output: 10 20 30
11 21 30

K value will be destroyed after inner function exit.

inner ≠ outer(); [by writing these line here it will call outer function again & the value of I & K will be destroyed

Output : 10 20 30
11 20 30

Q. What kind of scoping does js use ?

=> Lexical

↳ ability for a function scope to access variables from the parent scope.

↳ happens in closures

but variable defined inside a function will not be accessible outside that function.

↓

but variable defined outside a function can be accessible inside another function defined after variable declaration.

CLOSURE vs LET → It is a block scope so every let variable has diff. block.

↳ In closure whatever the variable we declare it has either global / function scope it means all have same block or pointing to same block.

Binding in Arrow function

Arrow function declaration :

```
var func = (a, b, c) => (1);
```

Non-local variable refers to variables that are neither in the local nor the global scope..

Arguments is object for getting the arguments of the function called. Therefore arguments[0] is argument n.

eg:

Closure

→ used for currying (an advanced technique of working with functions.)
it translates function from callable as f(a,b,c) into callable as f(a)(b)(c).

→ function bundled along with its lexical scope is closure.

→ closures are capable of not only reading but also manipulating the variables of their outer functions.

Constructor & Prototypes

This keyword

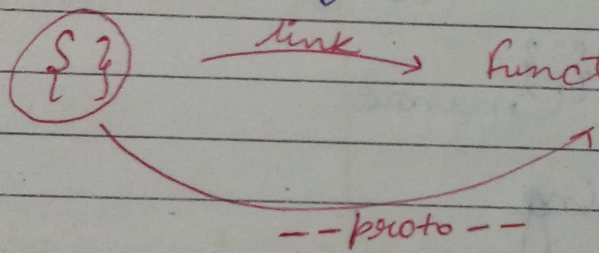
Rules :-

① New keyword :

→ When we are calling a function with "new" keyword.

(I) Create a new object {}.

(II) new keyword will link new object {} with prototype of that function i.e.,



(III) New keyword will call the function & ~~set~~ the value to the object that was created in ~~1st~~ step.

How new keyword will call the function?

⇒ functionname.call({})

Called internally

It will pass the new object

(IV) If the function is not returning anything return {} then it will return new object

eg:

```
function vehicle(name) {
    this.name = name;
    console.log(this);
}
new vehicle('Car');
```

// Output: vehicle {name: "Car"}
vehicle {name: "car"}

How it happens.?

i) {} → new object created
ii) {} → link vehicle.prototype
iii) vehicle → this → {} → name
iv) return {} → name

(II) Explicit binding

⇒ User will tell the Javascript that the value of function at the time of calling should be this or that.

Other way of calling function :—

① functionName.call(Argument); → must be an object

Using '.' dot method we are calling the function in JS.

② functionName.apply(Argument)

→ In explicit binding, we can call a function with an object when the function is outside of the execution context of the object.

Date
Page

eg:

```
const john = {
    name = "John";
};
function ask() {
    console.log(this, this.name);
}
//output: when,
ask.()  → window object
ask.call(John) → {name: "John"}, "John"
ask.apply(John) ↗
```

(or)

(iii) Hard Binding

Q. How call.(), apply() & bind() is different from each other?

call()	apply()	bind()
• 1st argument must be the object context	• Same as call() but → it	• Similar to call() but unlike call() method call funct directly bind returns a new funct. & invoke that new function.
• Other parameters can be values.	• Allows to pass arguments like • apply(object, arr)	eg: let newFn = functName.bind (object, arr[0], arr[1])
• eg: functName.call(object name, array)	• when we have multiple value arguments to pass use apply().	newFn();
• Arguments must be passed one by one.		
• eg: • call(object, array[0], array[1], array[2]);		

(iii) bind()

Eg: var Raj = {
    name: 'Raj',
    greet: function() {
        console.log('Hello', this);
    }
};

raj.greet()
// output: Hello { name: "Raj", greet: f }

bind()

Passing reference of greet to localAsk

function

var localAsk = raj.greet;
localAsk();
// output: Hello, window {---} object

Hard binding

User can tell the raj.greet funct. that when you are calling localAsk() then value of this should be some object. inside localAsk()

↓

Q. How we do local binding?

→ Using bind() method.

keyword

thisArg

var localGreet = raj.greet.bind(1stArg, ----);

raj

this bind() will return new funct.

Q. Why bind() method is useful?

=> Bcz →

Case I:

Same eg: of bind() method

write →

```
SetTimeout(raj.greet, 1000);
```

→ output: Hellow window object { }

the value of name will be lost bcz

SetTimeout call greet function after 1sec and it calls like this.

```
SetTimeout(callback, ms) {
    callback()
}
```

greet() ←

Case II:

```
SetTimeOut(raj.greet.bind(raj), 1000);
```

it will not lost the value of name.

Output:

(iii) Implicit Binding

→ It is applied when user invoke a function in an object using 'dot' notation. 'this' keyword will point to the object using which the funct. was invoked.

eg:

```
Var a = {
    name: 'Nisha',
    greet: function() {
        console.log('Hello', this);
    }
};
a.greet();
```

Implicit Binding (here user is calling greet() which is inside a object)

value of this keyword is a object itself.

Output: Hello {name: "Nisha", greet: f}

```
Var localAskFunc = a.greet;
localAskFunc;
```

output: `f() { console.log('Hello', this); }`

```
localAskFunc();
```

Output: Hello window object { }

(IV) Default Binding:

→ It helps in determining the value of this keyword inside the function.

Apply DB when all 3 rule is not ~~working after~~ applied ~~applying~~

Eg:

```
function ask() {
    console.log(this, this.name);
}
ask();
```

→ calling ask() in global context

output: window object { global }

then value of this will be window object inside global context

Note: In recent JS version, window object has a property called name with value equal to empty string ("").

while determining the value of this keyword we have to check the callsite function not function declaration.

(V) Putting all 4 Rule in One code.

```
var person = {
    name: 'John',
    ask: function () {
        console.log(this);
    }
};
new (person.ask.bind(person))();
```

Implicit binding rule

ask() → hardbounded this → person obj.

Explicit Binding

keyword

① rule

Output: ask {}
ask {}

② rule refers to ask();

③ rule, Passing Person object which is Hard binding

Person.ask.bind → give new funct.()

inside thisfunct()

the value of this is Person object

Person{}

Using new keyword, ① st rule will apply & value of this inside the new funct() will be instance of empty object {}

Why Prototype

eg:

```
function vehicle (numWheels, Price)
  {
    this.numWheels = numWheels;
    this.price = price;
    this.getprice = function()
    {
      return this.price;
    }
  }
Var vehicle1 = new vehicle (2, $0000);
Var vehicle2 = new vehicle (4, $00000);
console.log(vehicle1);
console.log(vehicle2);
```

//output: vehicle {numWheels: 2, price: 50000, getprice: f}

//output: vehicle {numWheels: 4, price: $00000, getprice: f}

Here, the getprice() function has copies in each & every vehicle objects which should not be there that's why we will use prototype.

↓

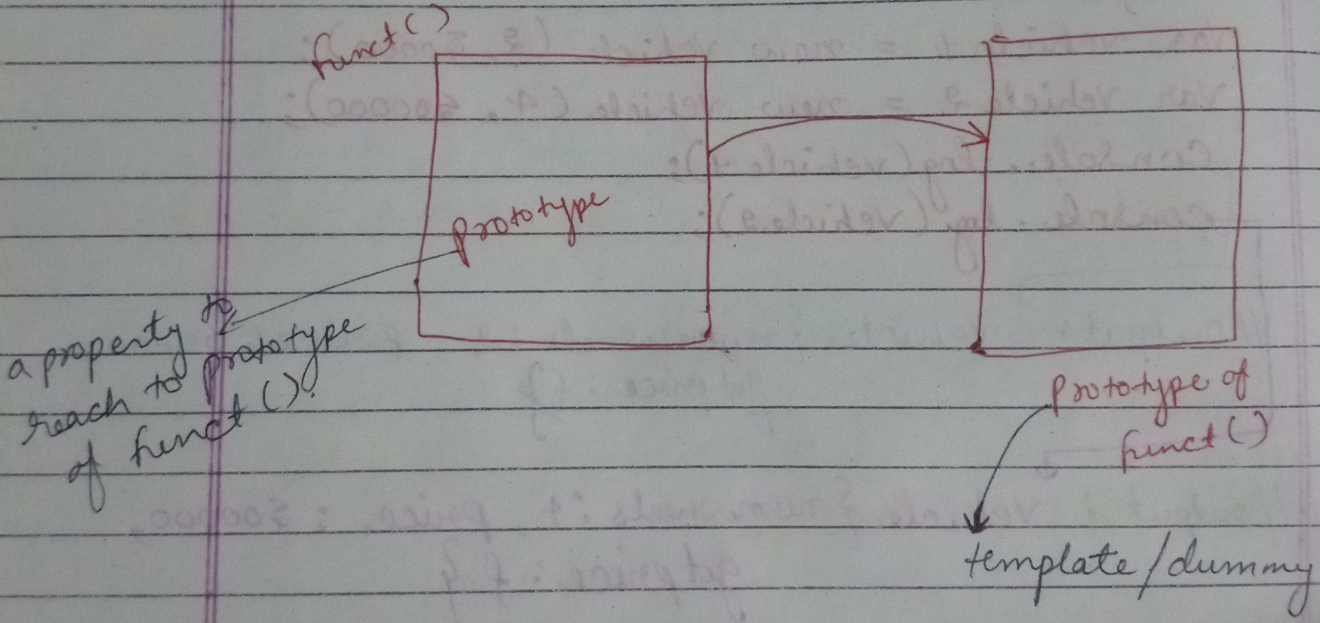
memory wastage

=> to access prototype :
FunctName.prototype
output: {constructor : f}

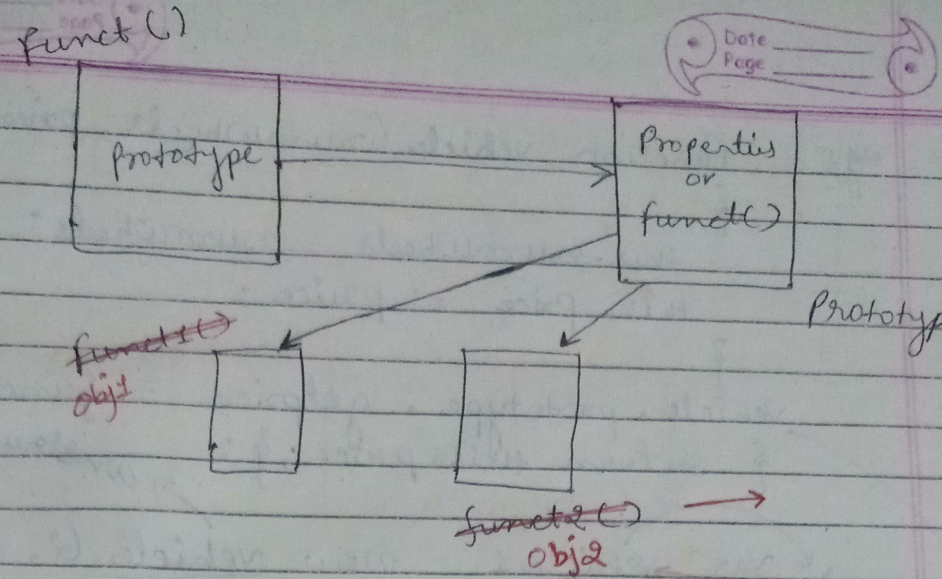
Note: All non-primitive is object in js as well as function is also an object in js.

Understanding the concept of Prototype

- for every function creation 2 objects are created one for the function & another for its prototype.



- if user is creating ~~func~~ objects in funct() by calling it in constructor mode then whatever objects that are created is the prototype of the previous prototype of funct().



- Whatever the properties / funct() we are specifying in prototype funct1() & funct2() can access it.

[obj1] [obj2]

↳ they have internal reference to [Prototypes]

- If user want to go back from prototype to that funct whose prototype it is then

~~Vehu~~ functName.prototype.constructor

↓

It will take user to the function

Benefits of Prototype :-

→ to avoid memory wastage we can write inner function in Prototype mode like

mainfunctName.prototype.key = innerfunction()

{ }

Now, all the objects can access this all the object are sharing the copies of & not wastage of memory.

eg: function vehicle (numWheels, price)
{
this.numWheels = numWheels;
this.price = price;
}
vehicle.prototype.getprice = function()
{ return this.price; };

constructor funct

object var vehicle1 = new vehicle (2, 500000);
var vehicle2 = new vehicle (4, 600000);

create

If we want to add more properties then we can write like this :-

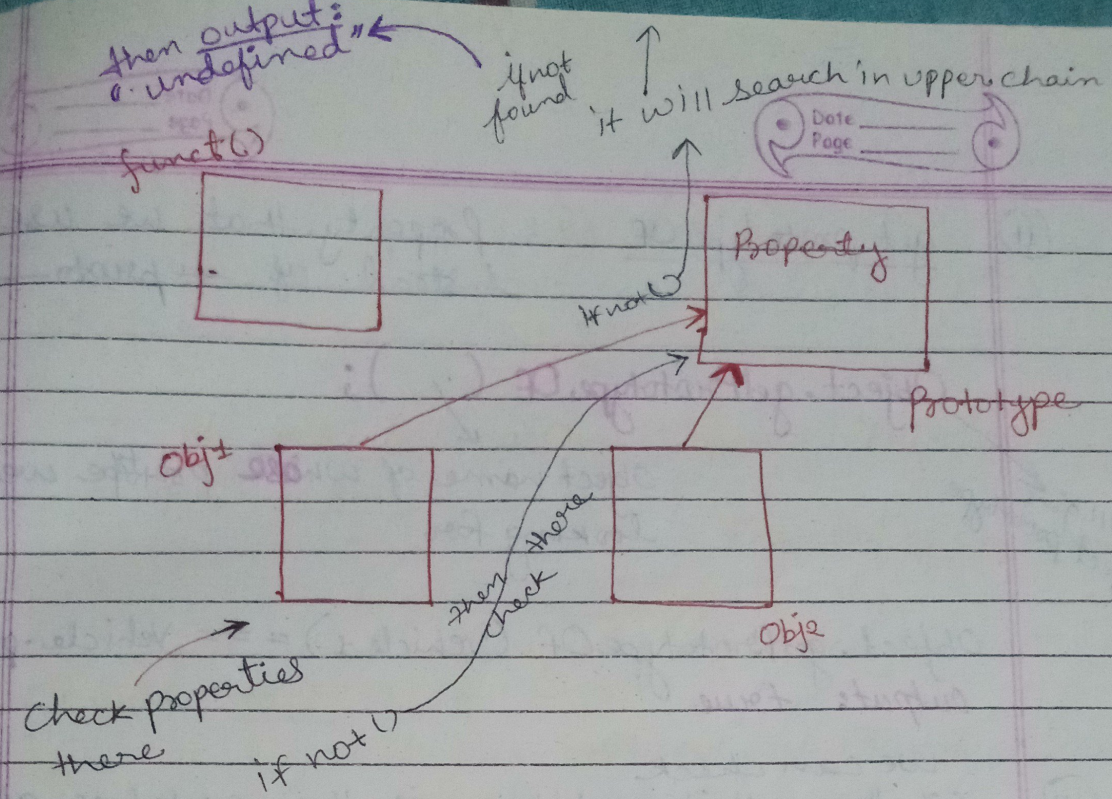
vehicle.prototype.color = function()
{ return this.color; };

→ we can add properties at runtime in js. using prototype.

→ prototype allows to share data among objects.
→ It allows us to add data & behaviour at runtime among objects.

Q. What happens when we try to search property?

⇒ First, it will look into the object if properties is not there, then it will go into prototype & check there if it is also not there then it will go upper & search till the end.



★ Prototypes are created for every function

~~eg: function~~

Properties of Prototype :-

① `__proto__` (call it as dunder proto) This property is available in all the objects that user creates.

~~this property is deprecated in ES6 Js~~ This property will take the user to the Prototype of that object.

eg: Vehicle1.__proto__

output: {getprice: f, color = "black", constructor: f}

Vehicle1.__proto__ === Vehicle.prototype

output: true

(II) getPrototypeOf : Property that we use instead of --proto--

Object.getPrototypeOf ( ); → Object name of whose prototype we are looking for

it will give Object prototype

Object.getPrototypeOf (vehicle1) === Vehicle.prototype
output: true

(III) we can check whether this prototype is the prototype of a particular object or not.

eg: vehicle.prototype.isPrototypeOf(vehicle1);
output: true

Pass the object whose Prototype you want to check.

(IV) come from object prototype
.hasOwnProperty ( ); → it will tell us about the property which is inside the object or what property the object are inheriting from the prototype.

eg: Objectname.hasOwnProperty ("Nameofproperty");

if property is inside the object then it will return true.

Every function has a property called 'name' value of default 'name' is set to the function's name.

If the property is not inside the object but it is the part of prototype chain then it will return false.

eg: vehicle1.hasOwnProperty('Price');
Output: true

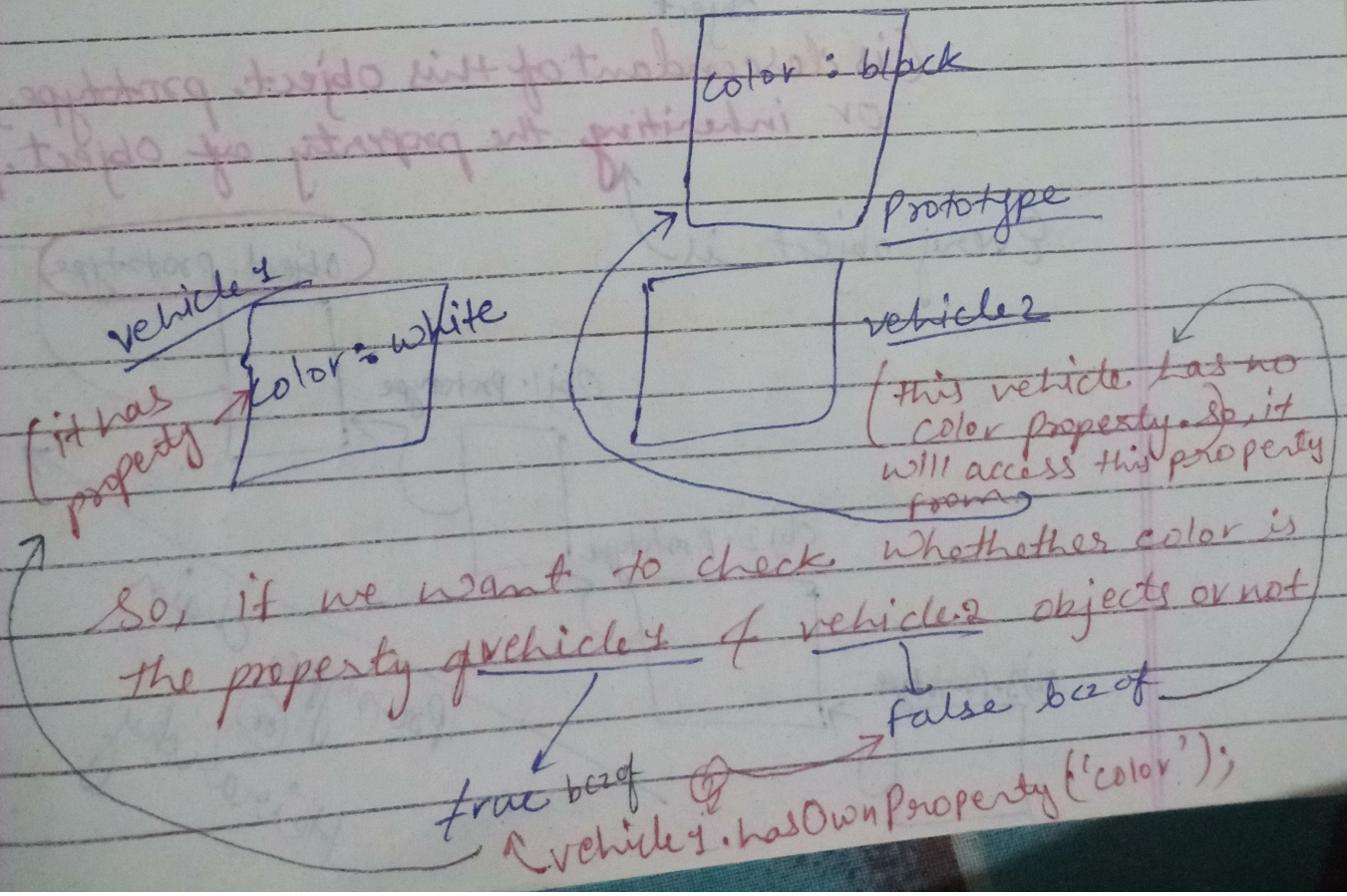
Property of obj vehicle1

→ vehicle1.hasOwnProperty('getPrice');
Output: false

Property of Prototype not the property of vehicle object.

eg:
Suppose we are changing the color to white of vehicle 1 then how we will do

Vehicle.Prototype.color = "red";
vehicle1.color = "white"



Objects

To create an empty Object:

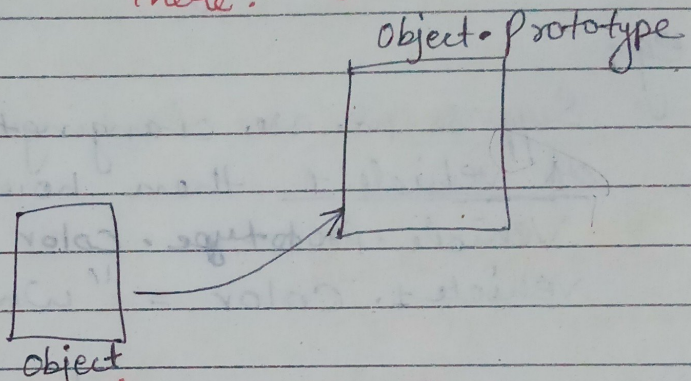
① var obj = { };

② var obj = new Object ();

output: undefined

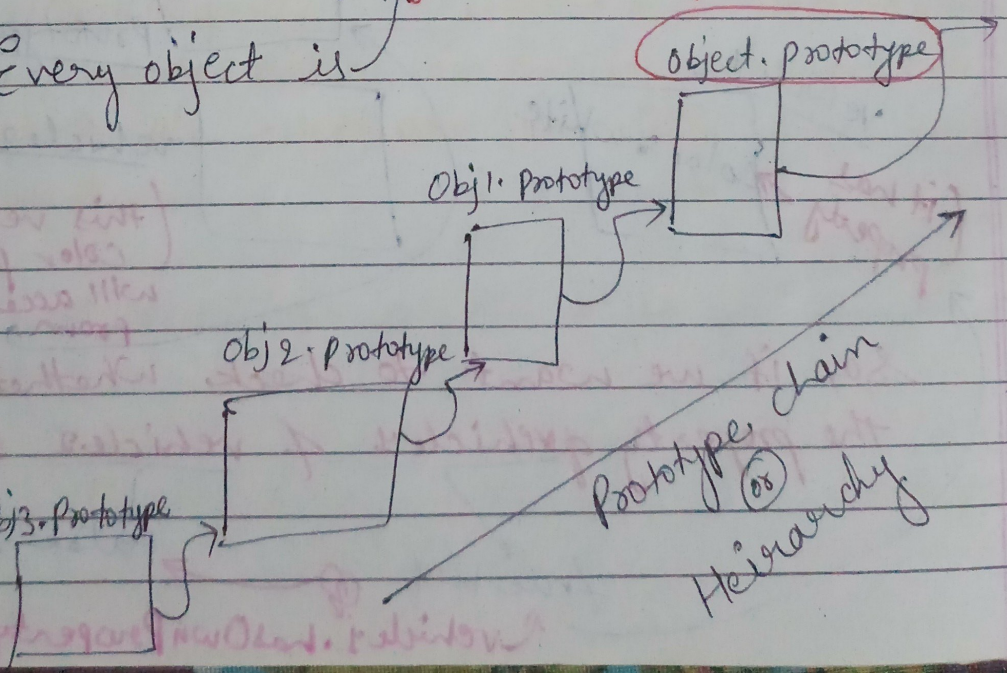
Object as a function exist in js, we don't need to create it.. its prototype must be means there.

~~var obj~~



(is descendant of this object.prototype) or inheriting the property of Object.prototype

Every object is



Object vs Object()

Object() → Constructor function

Object ↔ non-primitive object in js inherited from Object(), object has key-value pair

they are inheriting property from Object()

typeof(object) → "object"

typeof(Object) → "function"

eg:

```
Const object1 = new Object();
object1.property1 = 42;
console.log(object());
```

output: Object {}

CLASS (keyword)

- introduce in ES6.
- It is a prototype based inheritance → function

```
Class vehicle {
    constructor(numwheels, price) {
        this.numwheels = numwheels;
        this.price = price;
    }
    getprice() { return this.price; }
}
var vehicle1 = new Vehicle(2, 500000);
```

constructor ← constructor function

Prototype ← getprice()

- In ES6, If user want to create function inside object then,

```
var obj = {
    functionname() { }
}
```

older type

```
var obj = {
    functionName : function() { }
}
```

★ If we donot add a constructor to a class then by default empty constructor will be added automatically.

```
Class person {
  constructor (name)
   { this.name = name; }
}
Console.log (typeof (Person)); //output: Function
Console.log (Person === Person.prototype.
                        constructor);
```

Output: True → class person { constructor(name) }

```
Console.log (Person.prototype);
// Object
```

CLASS Expressions and Hoisting

Class

- We will not be able to call a class without new keyword. if we try to call it will throw TypeError: Can't call a class as a function.

eg:

```
Var obj1 = className( );
```

✗

- Hoisting is not possible If we try to fetch class before defining the class it will throw error: className is not defined.

eg:

```
Var obj1 = new className( );
```

class { }

Not possible

class declaration are not hoisted.

Normal function

- We can call funct without new keyword.

eg:

```
var obj1 = functname( );
```

✓

- Hoisting is Possible it means function declaration are hoisted.

eg:

```
var obj1 = new functname( );

function ( ) { }
```

Possible

- Function Expression — can create.

eg:

```
var functName = function ( ) { }
```

✓ Possible

- Class expression can also be created.

eg: unnamed class Expression
var classname = class { }

→ we can create named or unnamed class expression like

var classname = class classname { }

It will not be accessible outside the class.

Inheritance using classes

→ Inheriting properties of a class from other class, use extends

eg: class classname2 extends class1name

→ class car extends vehicle { }

It means car (class) will inherit properties from vehicle class

→ super.prototypename(); is used to call the prototype of parent class prototype

→ inside child class prototype function().

get Keyword is used to create a getter method in class.

ASYNCHRONOUS JAVASCRIPT

Promises → Asynchronous (Success, Rejection)

promise constructor

var promise = new Promise (function(resolve, reject) { // do sth }); ← has 2 arguments ← callback funct

Promises have 3 states

Result → undefined ← value

① Pending → initial (waiting) stage, neither fulfilled nor rejected

② Fulfilled → completed stage (operation was completed successfully)
Result → a value

③ Rejected → Operation failed
Result → an error object

Promise is settled only when it is in fulfilled or rejected stage otherwise promise will remain in Pending state.

eg:
```
Const promise = new Promise ((resolve, reject)
=> { const a = 10;
     const b = 15;
     if (a === b) { resolve(); };
     if else { reject(); }
});

promise.then(() => { console.log("Success"); })
.catch(() => { console.log("Failure"); });
```

=> Where promises is used?

→ Used to perform the task where user have to wait for sometimes.

Eg: API call, when user get data from server ↓ have to wait

→ To access the message which is inside resolve stage write,

Promise.then(() => { });

CALLBACK

→ Function are passed as an argument to ~~Object~~ other function & also we can return
→ ~~Fast~~ function from another function.

Callback is a function which can be passed as an argument to another function which is basically invoked sometimes later in the futture

Callback is used to complete routine & sequence of action.

Why Callback

Since js is a single-thread language bcz memory
- js has only one call stack & one heap
& js read the code line by line & execute one at a time.

no deadlock
↑
a single-thread language

Js is basically Synchronous in nature but how js perform or able to execute Asynchronous code.

↳ It reads & execute code line by line.

→ If we want to delay function execution then we will use callback.

→ Callback helps us to achieve asynchronous nature in javascript.

Q. Why do we use promises instead of callback

→ Because it is more readable.
→ To escape call back hell
→ Bcz it makes the code more maintainable.
→ bcz it can handle multiple callbacks at the same time thus avoiding undesired callback situation.

Promise.then & .catch

① .then() method

has 2 function as parameters :-

(a) resolved
(b) rejected but it is optional & to handle error user should us .catch() method instead of .then()

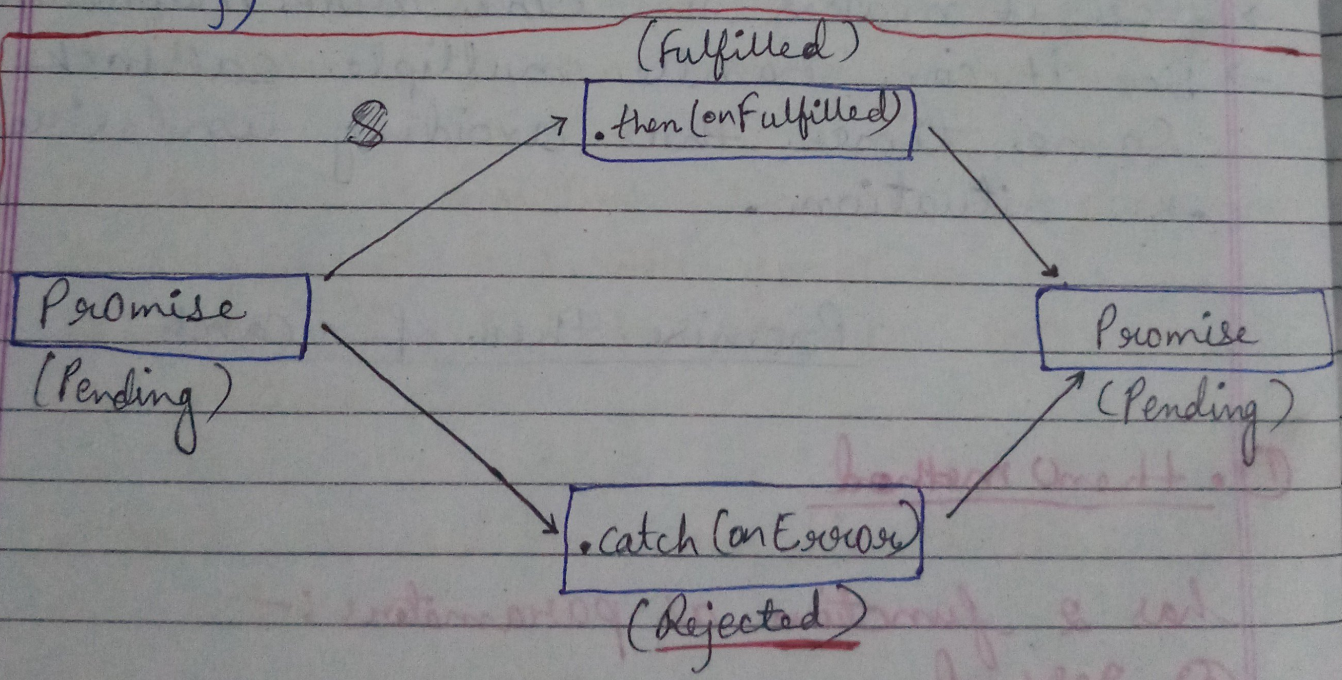
Syntax:

```
Promise
  .then(function(result) {
      // handle success
  }, function(error) {          // optional
      // handle error           // in .catch as well
  })
```

(11) : Catch() method is invoked when a promise is either rejected or some error has occured in execution.

→ used as a handler wherever there is a chance of getting an error.

```
.catch(function(error){
    // handle error
})
```



SETTIMEOUT

- SetTimeout is not a js function.
- SetTimeOut() is a part of webAPI provided by browser
- SetTimeout is given by the browsers. There are browsers API which has some functionalities like setTimeout, SetInterval, clearInterval given to us by the browser. DOM API is also a part of browser API.

  getElementById()
  QuerySelector etc.

All javascript Engine has these APIs. They are using the Browser that's why we are able to use these SetTimeout or other ~~API~~ by API.

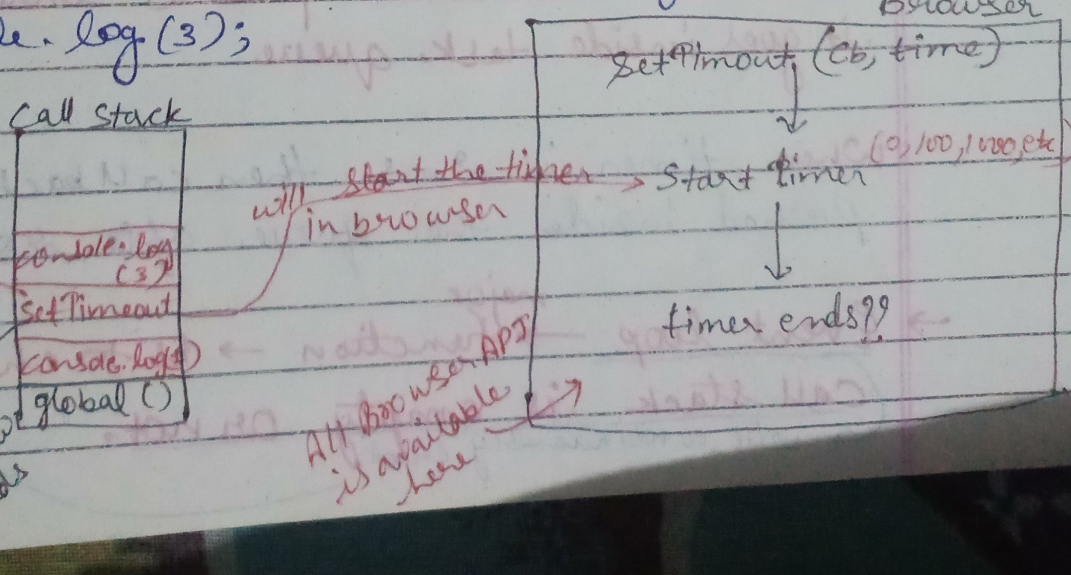
- SetTimeout has no execution context in stack where as every function has their execution context at the time of calling

How setTimeOut works in the backend:

```
console.log(1);
setTimeout(function(){ console.log(2); }, 0);
console.log(3);
```

(global context)

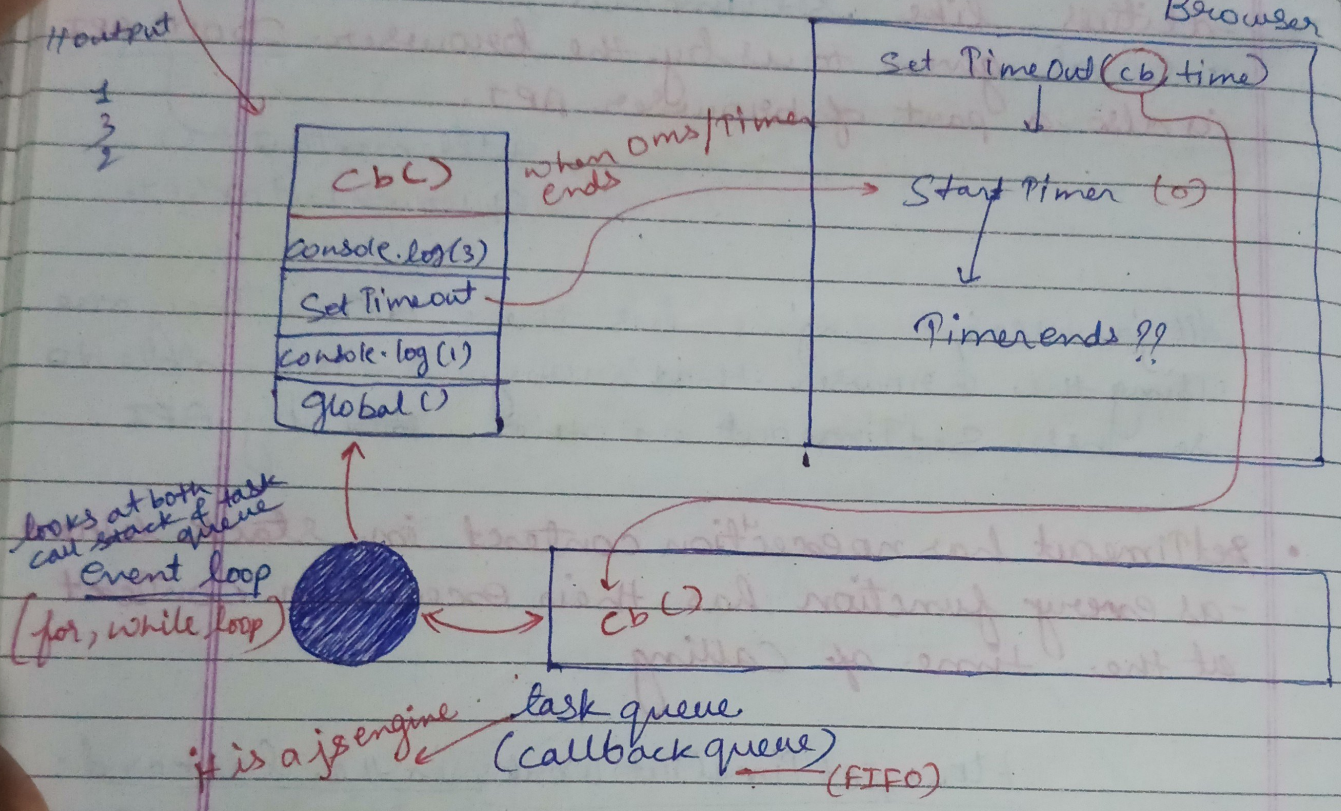
0 millisecs



SetTimeout is called at the end. Why?

↳ has no effect on call stack after timer ends.

★ Event Loop → interview question



→ When timer ends immediately, then whatever callback user have passed inside SetTimeout it goes inside task queue.

When timer is 2ms then callback will be pushed inside the queue after 2s.

→ Event loop → major function → is to check whether call stack is empty or not.

↳ will continuously check

Case 1: if callstack is empty, then, event loop will perform dequeue operation means
① it will get the 1st element from task queue & push it to the call stack.

That's why output is 1
3
2 → at last

⇒ we have to wait till the call stack is empty bcz only then setTimeout or call back will be executed.

PROMISES and Event Loop

How it works in js engine?

setTimeout(function() { console.log(1); }, 0);

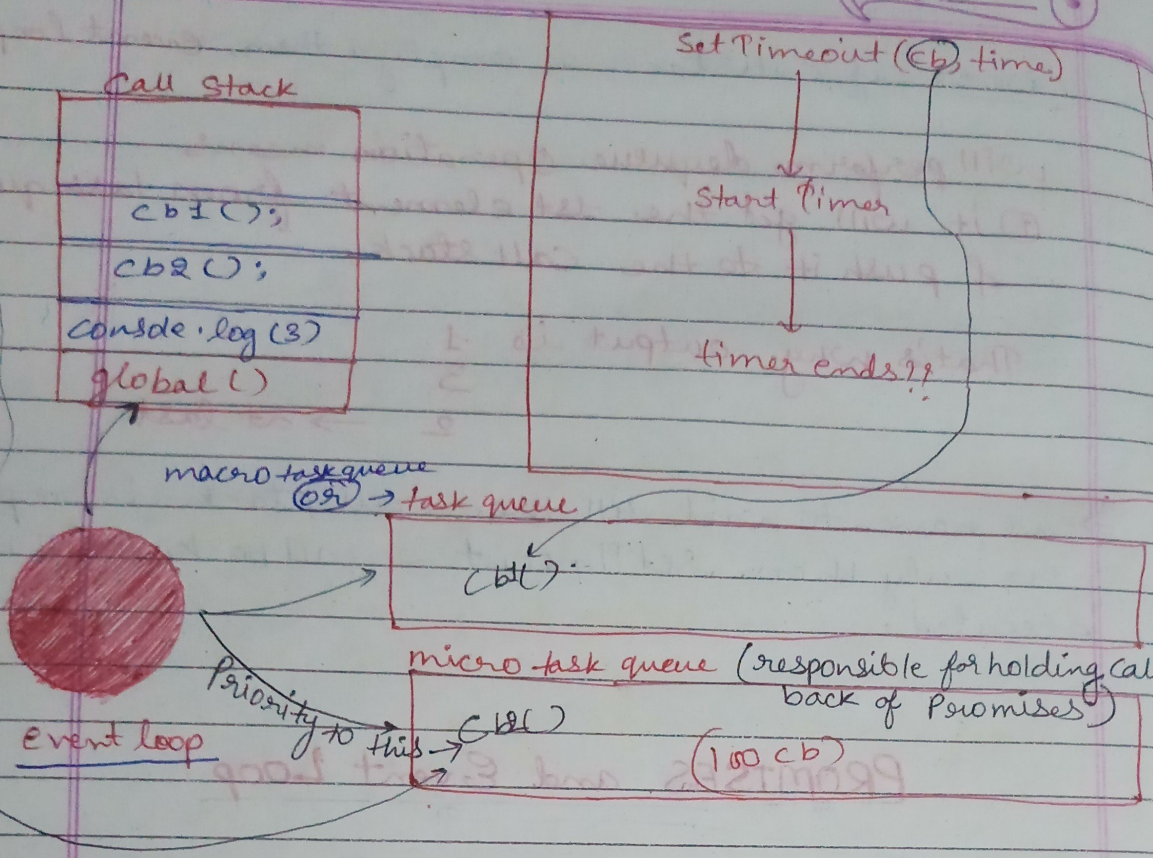
var Promise = new Promise(function(resolve, reject) { resolve(2); });
Promise.then(function(data) {
  console.log(data) });

console.log(3);

// Output: 3
2
1

1) SetTimeout
2) Promise
3) console.log()

cb → call back



→ Promise also have callback function which goes inside micro task queue.

Event loop will always give priority to the ~~from~~ micro task queue.

(from)

→ will deque cb2() inside to call stack. that's why 2 is printed in middle.

If suppose there will be 100 call back function then until & unless all those 100 will ~~not~~ be deque then only task queue call back will be deque & sent it to call stack.

★ Microtask are pushed.

★ task queue. task queue

Async

• "async"

eg:

Await

• "await

1) Sum
await

★ Microtask queue are the callbacks of promises are pushed into call stack by the event loop.

★ task queue or microtask queues ← callbacks of setTimeout → Pushed into → task queue by the event loop.

Async & Await

- Keyword
- Emerged in ES7 JS version
- helps us to deal with promises efficiently.

Async

- "async" Keyword is written at first with any function

eg: async function asyncTask() {
    return "Resolved";
}

async → Keyword
return "Resolved" → will return Promise

Await

- "await" Keyword → as soon as js engine encounter await, it will

1) Suspend the function quickly & thrown the await function in microtask queue

2) In the background, user have to wait till the function completion

3) Js will move to Synchronous code.

1) When the call stack is empty then await function will go to the call stack by the event loop.

await is basically used when we are getting data from the server. & then below code will be executed.

using async & ~~api~~ await we will ~~fetch~~ data from API.