his tutorials of Namaste JavaScript season 1 and season 2 and other playlists,Topics Covered in notes
- Js Execution Context
- Hoisting
- Function Declaration vs Function Expression
- undefined vs not defined
- Scope
- Errors: reference, syntax, type
- Closures
- Callbacks
- How async js code works?
- Js Engine (v8)
- High-Order functions
- Promises
- Promise API
- Async/Await
- Try/Catch
- this keyword
- "use strict"
- call apply bind
- Currying
- Prototype
- Inheritance
- Abstraction
- Encapsulation
- Static props and methods
- Event bubbling and capturing
- Event delegation
- async/defer

Javascript Execution Context

when JS engine scans code, it makes an env → EC.

Two Types → 1. Global → Created when JS code first starts to run and it represents the global scope in JS.

[this] → window obj in browser and global in Nodejs

2. Function → Created when function in code starts executing

Execution Context works in two phases

1) Memory Creation phase → Allocate location to variables and functions, stores variables with values as undefined and function reference

Complete definition

2) Execution phase → Starts through entire code line by line. assign values to variables in memory.

For functions, when invoked JS once again create a new function execution context (FEC)

when function return the value, FEC will be destroyed.

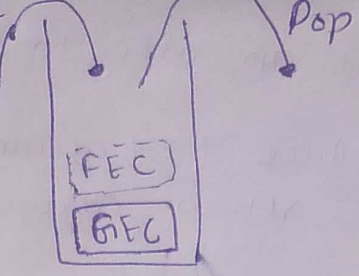
→ Once the entire code execution is done, GEC will be destroyed also.

Call Stack → To keep track of contexts, JS uses a Call stack. A Call stack → (Execution Context Stack, Runtime stack, Machine stack

↓

Last In first Out.

Representation →



* Call stack has fixed size depending on system or browser

exceeding ↓ Results → stack overflow err.

{Hoisting} ⇒ Behaviour in JS where all declaration of a function, variable, or class goes to the top of the scope they were defined in.

↓

functions becomes accessible even before the line it was declare

↓

The whole process is done during memory creation phase.

variables → undefined

function → Complete definition

```
printHello();

Function printHello() {
  Console.log("Hello")
}
```

→ works fine since before execution printHello() → already stored in memory

Variable Hoisting

Hoisting Doesnot occur in function expression, only on func declaration.

(Var) variables are hoisted but with default value undefined.

(let) and (Const) → hoisted but inaccessible before default initialization → gives error

↓

some happens with Classes.

* Hoisting happens based on the scope. It sets variables and function to the top of the scope.

Global Scope
Block scope, function scope → local scope

Temporal Dead Zone : Area where variable is hoisted but inaccessible until it is initialized with a value.

Applies → let and const (x var).

```
Console.log(name)
        ⋮
let name = "Tapesh"
```

→ TDZ

First Class function / citizen

Ability to use function like values:
- assign to a variable
- pass as a argument
- return function

Function Declaration v/s Expression

Function Declaration (aka Statiment)

Named fx where you declare name after fx keyword

```
function print() {
    console.log("Tapesh")
}
```

Function Expression → anonymous functions, where use function keyword without name, then we assign that to a variable.

```
const print = function() {
    console.log("Tapesh")
}
```

Parameters → Used in function (p1, p2)

Arguments → Used when calling let xs = sum(p1, p2)

Function Exp benefits or drawbacks.

- Fx Exp → Cannot access before initialization
- Fx Exp → need to be assigned to be used later.
- Anonymous fx are useful for anonymous operations.
  - Examples → IIFEs, Callback fx

var → undefined Hoisting.

Cannot access before initialization. (let, const)

Named function Expression

```
const print = function u() {
    console.log("Tapesh")
}
```

But → u x accessible.

Global Object - Window Object

created with creation of GEC.

object that always exists in the global scope.

When we run JS inside browsers it creates a js object named as "window" ← Global obj of JS

Global variables can accessed anywhere in project using 'this' keyword.

Undefined vs Not defined

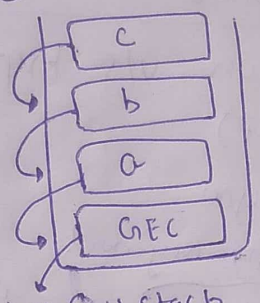
Predefined global variable that represent the value assigned to a variable during the ~~compiled~~ memory creation phase

Means variable has not been declared at all. Accessing such variable results in a Reference error.

→ A variable that has been declared but not assigned a value is a undefined and a variable that has not been declared at all is not defined.

Scope : area where an item (var, fu) is visible and accessible to other code.

```
let name = "Tapesh";

function a() {
    function b() {
        function c() {
            console.log(name)
        }
        c()
    }
    b()
}
a()
```



null Call stack

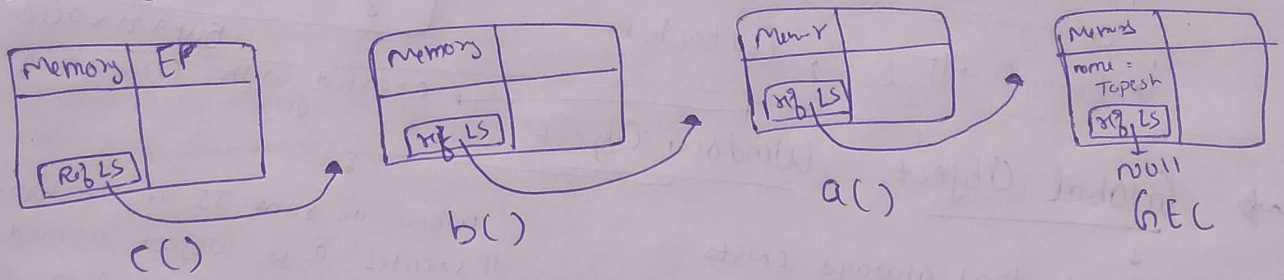
[c] → access of variable and function in his own scope + lexical scope of parent. (b, a, GEC)

[b] → access of own scope + lexical scope of parent (a, GEC)

[a] → access of own scope + lexical scope of parent (GEC)

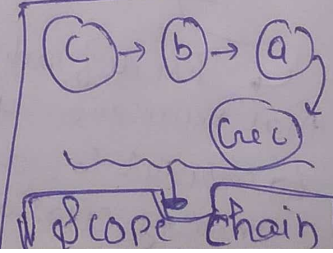
[GEC] → access of own scope + parent → since no parent → [null]

Lexical Scope (LS) : (static scope), a place where all the variables and functions of parent lies. This scope passed to the children scope (nested scope).



Scope Chain : Determines the hierarchy what JS code must go through to find the lexical scope (origin) of specific variable that got called

Means Code first find variable name in C's scope, if not found goto lexical scope (b's scope), (a's scope), (GEC's scope), returns not defined error.



Errors

- **Refrence Errors** : When you attempt to use a variable (or a refrence) that hasn't been declared.
  - Accessing undeclared variables
  - or Accessing a variable outside of its scope.
- **Syntax Errors** : When code did not conform to the structure or syntax rules of Js lang.
  - incorrent use of lang constructs
  - missed or extra punctuations.

  ↓ No line execute in this case.
- **Type Errors** : When operation is performed on a value of an unexpected type.
  - trying to call function that is not a func.
  - accessing props or methods of null & undefined

Block → Block is defined by curly braces { ... }

↓ (compound statment) → Used to combine multiple Js statments into one group, we can use it wherever Js excepts one singe statment.

```
if (true) {
  ...
}
```
→ Block

Block scoped → Scope in which we can access all variables and functions of that block. Lexical scoping also works here.

let and const are block soped - means let and const cannot be accessed outside the block.

But var is global scoped.

Shadowing → If we have same named variable outside the block. Block variable shadows the outside variable.

```
var a = 1 → Shadowed
{ var a = 2
  log(a) → 2
}
log(a) → 2
```

→ In case of let and const →

```
let a = 1 → Shadowed
{ let a = 2
  log(a) → 2
}
log(a) → 1
```

Because let and const variables stored in different memory location in block object for block variables, and script object for ~~local~~ global variables.

- → Global → reserved for var
- → Script → separate memory for let and const outside block scope
- → Block → sep memory for variables inside scope (let & const).

→ Illegal Shadowing → when outside there is let & const variable but inside var variable.

```
[ let a = 10;
  { var a = 20; } ]
```

↑ Example

Becoz if a variable is shadowing any outside variable, it should not cross the boundary of its scope.

(But) →

```
[ var a = 10;
  { let a = 20; } ]
```

← works fine. since let is creating in separate block scope.

✳ In case of function, var does not modify outside var with same name, since function creates it's own sep. memory.

→

```
var a = 20;
function (x) {
    var a = 30;
    log (a) → 30
}
x()
log (a) → 20
```

Closures

A closure is the combination of a function bundled together with references to its lexical environment.

In JS closure is function that remembers its outer variables & can access them even when called independently.

```
func x(){
    let a = 12
    return func y() {
        log (a)
    }
}
Const y = x()
y()
```

→ Here y has the access to all variables of x because of closure.

Advantages

1. Data hiding / Encapsulation
2. State Retintion
3. Currying
4. Memoization
5. Asynchronous Programming
6. Event handling.

```
function counter() {
    let count = 0
    return function () {
        count += 1
        return count
    }
}
let c1 = counter(), c2 = counter()
c1() → 1
c1() → 2
c2() → 1
```

Disadvantages

1. Variables declared inside a closure are not garbage Collected. [Garbage Collection → done when function works completed. All the variables → deleted]
2. Too many closures can slow down your application. Caused by duplication of code in memory.

Imp Example

```
function x() {
    for (var i=1; i<=5; i++) {
        setTimeout(() => {
            console.log(i)
        }, i*1000)
    }
}
x()
```

O/P: 6 6 6 6 6

Because in lexical scope, we have refrence to i not that current value. So when they all started printing, i values was 6.

Soln

1. Change var i to let i since let is a block scope variable, so new reference is created for every iteration

2. Solution using var

```
function x() {
    for (var i=1; i<=5; i++) {
        setTimeout(...)   // new copy each time
        function closure(i) {
            setTimeout(...)
        }
        closure(i)
    }
}
```

O/P: 1 2 3 4 5

Callback Function → Passing function as an argument to another function is valid in JS, that passed function is call back function.

Ex →

```
function print (callback) {
    callback()
}
print ( function () {
    log ("Tapesh")
})
```

callback function

Need? → Helps to develop Async JS code, cb makes sure that a function is not going to run before a task is completed but will run right after the task has completed

Async Call back Example → Can use Cb for event declarations

Example + Closure Ex

```
setTimeout ( () => {
    log (" After 3 sec ")
}, 2000)
```

```
function closure () {
    let count = 0
    btn.addEventLis (click, () => {
        log (count++)
    })
}
closure()
```

Increment counter on every button click using closure & call back

Tip: Event listners consume a lot of memory which can potintially slow down the website therefor it is a good practice to remove if not in used.

How Asynchronous JS Code work?

Blocking: When we do tasks that takes a lot of time in the main thread like (API call, image processing etc).

means

So, we have to wait until the current task has finished

↳ These task are blocking the call stack or ~~main~~ main thread

↓

Solution → **Asynchronous Callbacks**

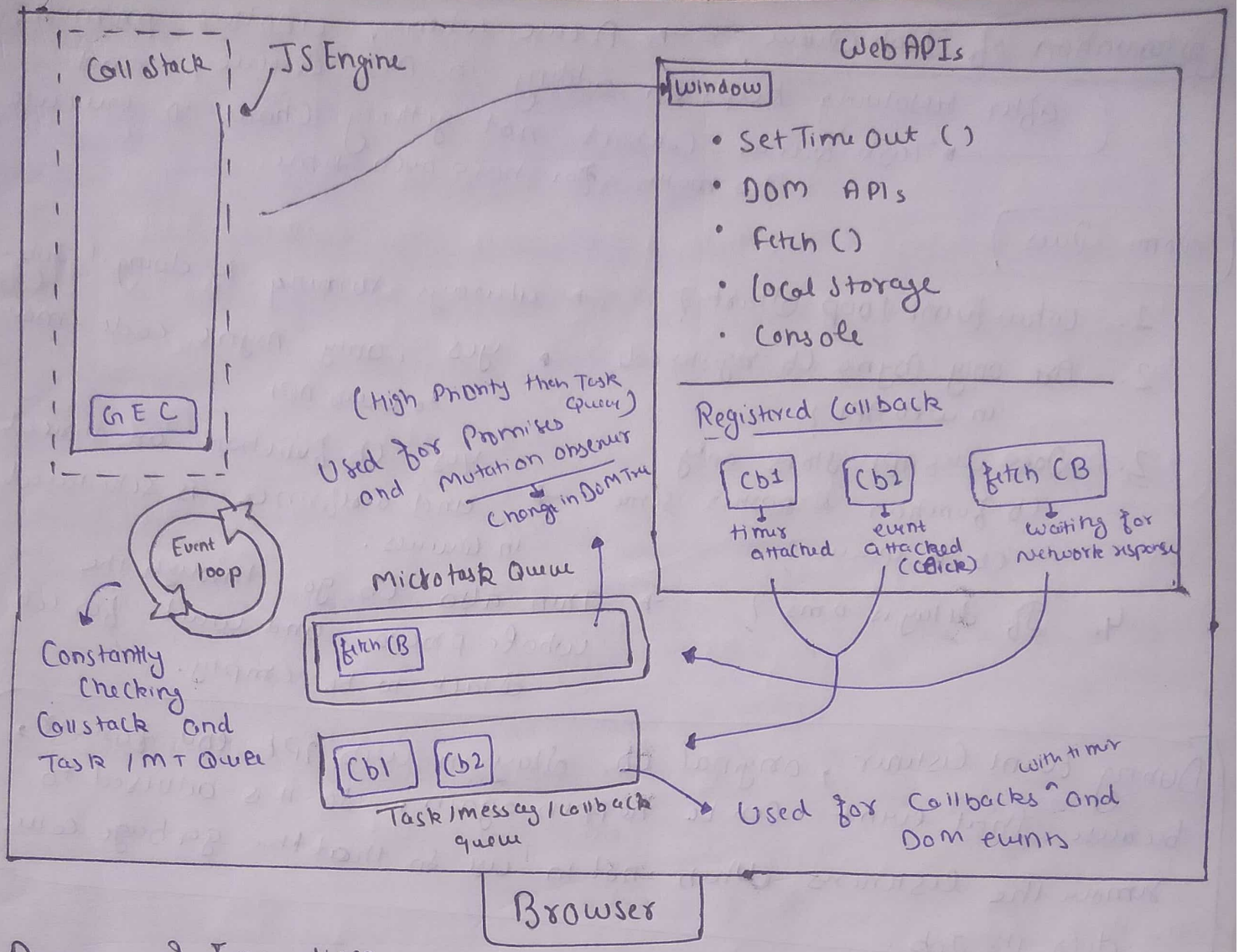
Web APIs → SetTimeOut, Console, fetch, DOM Apis

↑ for browser

C/C++ APIs → Event loop, task Queue, Micro task Queue

↑ for NodeJs

✗ part of Js engine, ✓ Browser / NodeJs env.



Process of Execution

- All the Callbacks get registered in Web API with their attached timer and event.
- Meanwhile other code keeps on running.
- When timer has expired or Callback is ready to do its work, Callback is pushed to the task queue. But not immediately executed.

* **Event Loop** → Job is to look into the call stack and determine if the call stack is empty or not. If callstack is empty, it looks into the Microtask Queue & task Que to see if there is any pending callback waiting to be executed.
If yes, then Event loop pushes the Callback to the top of the stack and then it execute.

Micro task Queue → ES6 introduced, Used for Promises and Mutation Observer (changes in Dom tree)

Higher Priority than Task Queue, means until MT Queue ~~becomes~~ not empty, Task Queue has to wait.

Starvation of Task Queue → A case when, another promise after resolving keeps on adding in MT Queue, and
→ Task Queue Callback not getting chance to execute. no matter for how much time.

Some Ques

1. When Event loop start? → always running & doing it Job.
2. Are only Async Cb registred in web API? → yes, only Async code moves to web API
3. Does web API store only Cb function & pushes same Cb to Queue? → yes Cb function or stored and a reference is scheduled in Queues.
4. If delay is 0ms? → Then also Cb go through the whole process and wait for Call Stack to be empty.

During Event listner, original Cb stay in web API env forever, because that event can be happen again, so it is advised to remove the listeners when not in use so that the garbage collector does its Job.

How JS Engine Works?

Browser

- **Rendering Engine**
  - Blink → RE responsible for the whole rendering of Dom trees, styles, events, V8 integration.
  - → Paints the content on your screen.
- **JS Engine** → Executes and Compiles JS into native machine code.
  - Chrome - V8
  - Firfox - Spider Monkey
  - safar - JavaScript Cor
  - Edge (IE) - Chakra

JS Engine Architecture → not a machine, but a program written in low level language (~~C++~~). It takes high level (JS code) and converts it into machine level code.

3 main steps to execute the code →
- Parsing
- Compilation
- Execution.

Parsing → Reading code line by line

2 major process:
- Break down code into tokens.
- Pass it to a **Syntax Parser**.

Takes the code and convert in into AST (Abstract Syntax Tree).

AST
↳ way of representing code as a tree like structure.

astexplorer.net

Compilation → Js uses JIT (Just in Time) compilation.

JIT involves an interpreter ~~alone~~ + compiler both.

Process: Ast goes to interpreter → Convert high level code to byte codes → then Execution

while doing this it takes help of compiler to optimize code on the runtime.

Execution → Execution x possible without
- Memory Heap
- Call Stack

Memory heap → place where all the variables & functions are assigned to memory.
→ also has garbage collector (GC) used to free up memory space whenever possible.
↓
Uses Mark and Sweep Algo to free up memory

Higher-Order Functions (HOF): Functions that takes functions as arguments or return a function as this result.

Why to use? → Code reusability
Abstraction of complex logic
Improved code readability
enhanced composability of functions

Built in HOF in Js → map, filter, reduce, forEach, addEventListener.

Callbacks Problems → 2 Problems

1. Callback hell
2. Inversion of Control

1. **Callback Hell** : Phenomenon happens when we nest multiple callbacks within a function is callback hell.
   - Shape of code resembles pyramid → "Pyramid of doom"
   - makes code difficult to maintain and understant.

Example →

Code increasing horizontally as well

```
api.createOrder(cart, function() {
    api.payment(function() {
        api.showOrder(function() {
            api.walletUpdate();
        })
    })
})
```

2. **Inversion of Control** : Means loosing the control over the code while using callbacks. Abse are giving control to the another nested callback function, which increses the dependencies to other one. ui don't know whats hap pening behind the scene

Solution → Promises

Promises : An assurance or guarantee that something will happen in future is promise. Can have two outcoms.

↓ In Js

fulfillment failure.

Promise is an object that will produce a single value some time in the future.
- if success → resolved value
- if failure → reason.

what value

It can have 3 possible states,
- → Pending : Default state
- → fulfilled : if successfull
- → rejected : if failed

Promise goes from 1

Pending → fulfilled

or

Pending → rejected

we can create promise using promise constructor

while using promises we have full control over the logic and callback, so we can also solve inversion of control problem using promises.

```
Const promise = new Promise ( (resolve, reject) => {

});
```

Call this function with the result if work done

else call this func with the error msg.

How to attach Callbacks? → using .then() method.

takes two callback function → first → if resolve, second → if rejected

```
Promise.then (
(value) => { Code after resolve },
(reason => { Code after reject }
)
```

→ we can omit first or second Callback if not required.

.then() → always return another promise so Chaining can be done easily.

→ using promise and .then() chaining we can solve Callback hell like problems

How to Handle errors in Callback? → using .catch() method

if anything goes wrong in promise → this method with the reason for that error.

Using catch → Error no longer remains uncaught.

```
fetch (url) {
  .then ( () => )
  .then ( () => )
  .then ( () => )
  .catch ((error) => log(error))
}
```

this will be called if we got error in any of the above .then method.

In chaining .then() method return its value in next .then's callback.

we can add .catch() in blw also, that will take care of all .then() above it.

.finally() → Method that runs always no matter promise resolved or rejected.

.then() → ~~runs only when promise resolved~~

.catch() → runs only when promise got error

→ handle result of promises whether rejected or solved.

Promise APIs → used to run multiple promises parallely

↳ Most common methods → 4

all() race() allSettled() any()

Promise.all()

- takes array (iterable) promises as argument.
- if all fulfilled → returns array of values we get after resolving.
- if any of rejected → stops running and return the error immediately.
  - Fail-fast → Behaviour

Promise.allSettled()

- Comes after ES 2020
- returns array of all settled promise whether fulfilled or failed.
- array value is object { status → fulfilled / rejected; value / reason }

Promise.race()

- returns single promise as an output
- returns promise who settled fast → resolved / reject.

if fastest resolved, it will stop there, we will not get any error if other promise failed or not, they will just not get executed.

Promise.any()

- returns a single promise who resolved first.

it will look for promises that resolved, once any promise resolved it will return the value.

→ if no one fulfilled → returns * aggregate error

we can get reason of rejection of all promises using error.errors → array.

Aggregate Error
↳ Several error wrapped in a single error.

Async/Await

- syntactical sugar over .then() & .catch() chains.
- make promise handling more prettier

async keyword to create asynchronous function,

```
async function x() {
}
```

always returns promise

if we don't return promise, JS automatically wraps the returned value in promise.

Await

Keyword basically makes JS wait until the Promise resolved or rejected

Example

```
async function x() {
    Const response = await fetch(URL);   // before promise
    Const jsonData = await response.json();
    conslog.log(jsonData);
}
```

Behind the scene

Wherever JS engine see await keyword, it suspends the complete function call and go and do the other work from call stack.

When Promise gets settled the function come back in Call Stack and continues executing.

Error Handling with Try/Catch

```
try {
    // Here we try to resolve promise
}
Catch (error) {
    // do work when got error in above block.
}
finally {
    // run whatever happens
}
```

all this inside function

Why Async/Await ?

- ✗ need of nested callbacks.
- ✓ simplify syntax
- ✗ Chaining

---

this Keyword → Always refers to an object.

Object it refers to will vary depending on how and where this is being called.

this → ✗ Variable
✓ Keyword

Value → ✗ Changed or reassigned.

↝ this in global scope → refers to global Object

Window → in browser
global → in Node.js

↝ this inside function

Value depends on strict / non strict mode

→ Strict mode → this refers to undefined
→ non strict mode → this refers to global object

(why?)

Also, matter how it is being called
(Not window ref) func() → undefined
window.func() → window obj

→ this substitution

if value of this is undefined or null this keyword will be replaced with global object in non strict mode.

↝ this inside an object method

```
Const obj = {
    a: 10
    x: function () {
        log (this)
    }
}
obj.x()
```

x got reference of obj

→ So here this refers to object obj.

this → { a: 10, x: function }

↝ this inside Call bind apply ← Sharing methods used to share properties of different obj.

```
Const obj2 = {
    b: 10
    ~~var function () {~~
        ~~log (this)~~
    ~~}~~
}
Obj.x.call(obj2)
```

passing this

→ Here obj2 → x & function, So it is taking x function of obj.

→ x function will print obj2 { b: 10 }

this inside arrow function

do not have their own this → so → they take this of their ~~lexical~~ enclosing lexical environment.

Examples

```
const obj = {
    a: 10
    x: () => { log(this) }
}
obj.x()
```

Lexical env of obj → Global Object

```
Cont obj = {
    a: 10
    u: function () {
        () => { log(this) }
    }
}
```

Lexical env of x which is obj

this inside DOM

refers to the element where called

```
<button onclick="log(this)"></button>
```

→ Print the button Element.

"Use Strict" → ES5, JS code execute in strict mode.

↳ declare beginning of script or a function.

why?

- easier to write secure JS
- bad syntax → to → errors

Not Allowed → using variable/object without declaring it → `u = 2`

↑ x let, x var x const

- → X Deleting variable and func ~~names~~.
- → X duplicate param names of func
- → X Octal numeric literals
- → X write to read only props.
- → X delete undeletable props.
- → eval, arguments, static etc, private, public → X use as variable name
- → eval x declare variables
- → this behaves diff inside functions

Call, Apply, Bind

→ Call() → Change the context of the invoking function

means → helps you replace value of this inside a function with whatever value you want.

```
func.call ( thisObj , args1 , args2 , ...)
```

(if x provided → Consider Global Obj) ← value that needs to be replaced

→ other required argument for func.

```
Const  student = {
         name : "Tapesh"
         print: function () {
                  log (this.name)
                }
       }
Const  student2= {
         name : " other name"
       }
student.print.call ( student2 )
```

this obj

changed this of student.

→ apply() → Same as call, only difference is arguments are passed as array.

```
func.apply ( thisObj , [arg1, arg2 . . . .])
```

→ bind() → Creates a copy of a function with new value of this, which we can invoke later.

```
Const newfunc = func.bind ( thisObj , arg1 , arg2 )
newfunc()
```

↓ functionality similar to Call

Currying : Transforms a function with multiple arguments into a nested series of functions, each taking a single argument.

```
f(a,b,c)
        f(a)(b)(c)
```

Why?
- helps avoid passing same variable again and again
- helps to create HOF
- less errors and side effects. (Checking Method - Checks if you have all the things before you proceed).

```
Const addCurry = (a) => {
    return (b) => {
        return (c) => {
            return a+b+c
        }
    }
}
addCurry(a,b,c)   => O/P 6
         1 2 3
```

Infinite Currying

↓

until user gives input.

```
inf(curry (a)(b)(c) ..... ())
```

empty input ← breaking condn

Ex

```
Const sum = (a) => {
    return function (b) {
        if ((b))   // if not passed then we go in else condn
            return sum(a+b)   // a+b → new arg
        else
            return a
    }
}
```

Currying vs Partial Application

In currying nested functions == arguments

↓

means each func must have single arg

```
f(a,b,c) => f(a)(b)(c)
```

Partial application transforms a function into another function with smaller arity (lesser args).

↓

```
f(a,b,c) => f(a)(b,c)
```

Write a function Curry that convert f(a,b,c) into Curried function f(a)(b)(c).

```
function curry ( func ) {
    return function curried func ( ... args ) {
        if (args.length >= func.length ) {
            return func( ... args)
        }
        else {
            return function (... next) {
                return curried func ( ... args, ... next);
            }
        }
    }
}
```

Initially got a then a,b then a,b,c

When get all arg call the original func

request b then c

Other request for next arg in other func call

first pass a,b then a,b,c

```
Const func = (a,b,c) => a+b+c
Const curried = curry (func)
log ( curried (a)(b)(c) )
```

Currying Using Bind

for Currying we Can also use bind function and separate the arg in separate function

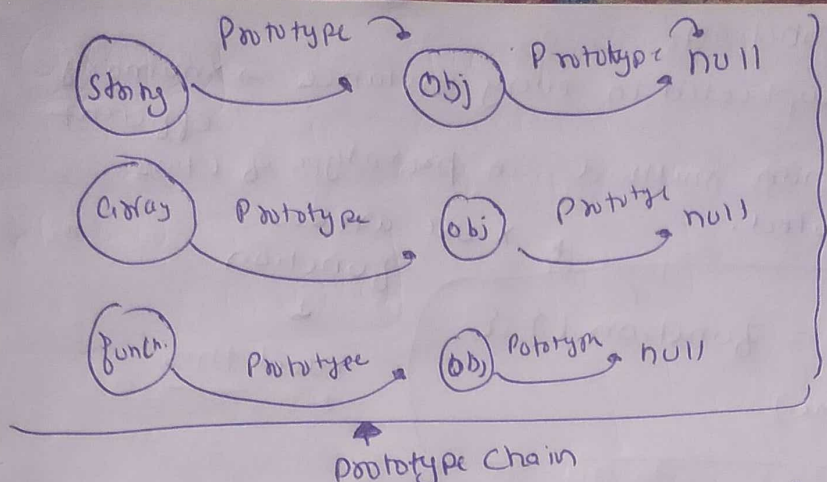
```
const add = (a, b) => {
    log (a+b)
}
[curry = add.bind (this, a)
curry (b)]
```

or

```
[add.bind (this, a) (b)]
```

Prototype In JS, object can inherit properties of another object, and the object from where these properties are inherited is called prototype.

Strings, array, objects → methods come built in within each type of data structure.



If we are accessing any prop from string / arr / functions, the prop not only searched in itself, in the prototype and prototype of prototype and so on until a prototype chain is reached.

Every Obj in Js has internal private prop [[Prototype]], x access directly in code.

→ But to find [[prototype]] of an object, we can use ↓

① log (Object.getPrototypeOf (arr))

② log (arr.__proto__)

→ prints all inbuilt func & prop arr have.

Prototypal Inheritance → Basically the ability of Js objects to inherit properties from other object. like array have access to all the properties of obj.

OOP in Js

Achieve this using constructor function.

Js is prototype based procedural language which means it supports both functional and object oriented programming.

→ Starts capital → convention

```
function User (name) {
    this.name = name
    this.printName = function () {
        log (this.name)
    }
}
const person1 = new User ("Tapesh");
const p2 = new User ("xyz");
```

Only with new keyword to create new instance

ES6 → class keyword
Classes are just syntactic sugar over constructor functions.

```
class User () {
    name;
    ~~printName ()~~
    constructor (name) {
        this.name = name;
    }
    printName () {
        log (this.name)
    }
}
const p1 = new User ("Tapesh")
```

Problem with Constructor function
↳ printName method duplicated in every instance → ✗memory efficient

So we need to put this common method in prototype of class
So it ~~the~~ will not be duplicated.

```
User.prototype.printName = function () {
    log (this.printName)
}
```

✗ use arrow function
↓
need this

↳ Initially User.prototype → empty obj {}

In classes, methods automatically put inside prototype.

Inheritance — we can inherit properties and functions of parent class in a child class or constructor function.

To inherit
↓
In constructor function
① Call the parent function (get props)
② Link the prototype (get methods)

→ in classes
① extends Keyword used
② super Keyword used to call constructor of parent

```
function Child (name) {
① Parent.call (this, name)
}
```

this of Child class gets all the props to Child this

name → all the args required in parent

② Link prototype

```
Child.prototype = Object.create (Parent.prototype)
```

make sure to write above Child prototypes.

```
class Child extends Parent {
    constructor (name) {
        super (name)
    }
}
```

super (name) → pass all args required

call the parent constructor

Abstraction — we can make properties & method private so that no one outside the class can access these properties.

↓
use '#' before property name or method name.

```
#name
```

```
Const ob = new User (name)
ob.#name = "xyz"
```
↓
throw error ✗ access to private prop

Encapsulation → Process of hiding and securing properties of objects.

we need to provide other mechanism to access these private properties

getter setters → since we can access pvt prop in class.

Syntatic Suger

```
getName () {
    return this.#name;
}
setName (newName) {
    this.#name = newName;
}
```

Obj.getName() → // name

Obj.setName(abc) → // set newname = abc

```
get name () {
    return this.#name;
}
set name (newName) {
    this.#name = newName;
}
```

Obj.name = newName → Calling setter

console.log(obj.name) → Calling getter

Static properties and methods

↳ shared by all the instances of a class

```
static count = 0
static getCount () {
    return className.count
}
```

→ can access using classname or this.constructor.

→ x access of class intances & x avail in objects

→ Static props & methods are inherited.

```
const obj = new className ()
obj.count → Error
obj.getCount → Error
```

→ Static props are initialized once

Similar to this (but outside class)

```
className.count = 0
className.getCount = function
}
```

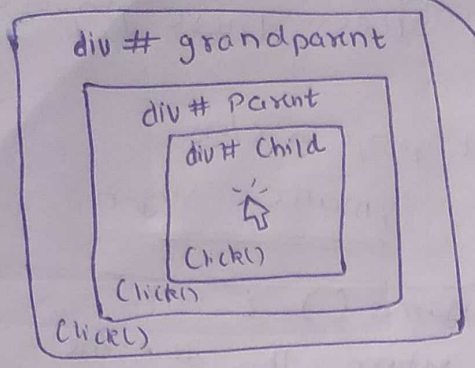
className.getCount () ✓

this.constructor.getCount() ✓

static block → we can create static block which will run when first time static method is used.

```
static {
    ....
}
```

Event Bubbling / Event Capturing



→ Two Phases of propagation.

How events travel through the (DOM)

Bubbling → Travels from target to root → highest level parent of target.

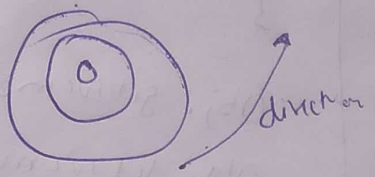
target → Clicked element

if target is child

first Child clicked
then Parent Clicked
then grand parent Clicked

root where event bubble stops

(Event Bubble up)

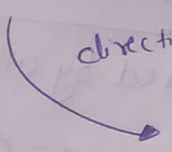


aka (trickling) ← **Capturing** → Travels from root to target

if target is child

first grandparent Clicked
then Parent Clicked
then Child Clicked

(Event trickles down)



We can trigger bubbling / capturing and control over propagation

→ element.addEventListner("event", callback, useCapture)

Optional boolean value
default → false → Bubbling
True → Capturing.

These propagations takes time and increase workload

Solution? → e.StopPropagation() prevents further propagation

```
grandp.addEventLis(click, (e) => {
    log(grandParent)
})
Parent.addEventLis(click, (e) => {
    e.stopPropagation
    log(Parent)
})
Child.addEventLis(click, (e) => {
    log(Child)
})
```

if child is clicked

O/P → Child clicked
Parent clicked

Propagation Stopped →

e. stopPropagation() → Stops all the parent event listiners but not other handlers on the target

means if any target has multiple event, other will also be called

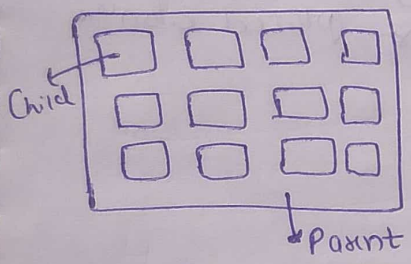
So O/P will be 1 2

```
Child.addEventLis (click, callback)
Child.addEventLis (click, callback2)

callback (e) => { e.stopPropagation()
                  console.log (1) }

callback2 (e) => { console.log (2) }
```

e. stopeImmediate Propagation() → Stops all the parent event listeners + other event listeners to target

So, only 1 will be printed.

Event Delegation → A technique in Js where we delegate the responsibility of handling an event to a parent element. By doing so, we avoid attaching multiple event listners to individual elements.

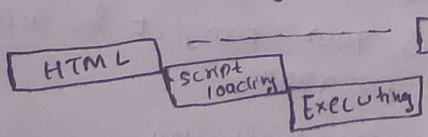


- ✓ performance improvement
- ✓ Dynamic element (adding new element will auto have event listner)
- ✓ Code simplification.

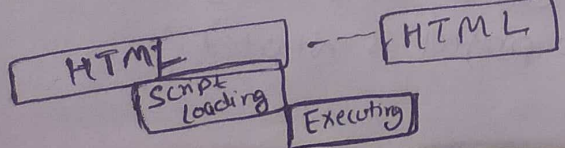
```
Parent.addEventLis ("click", (e) => {
    Console.log (e.target)   → Child that clicked.
    do work on e.target
})
```

Script Loading → <script> <script async> <script defer> → 3 ways

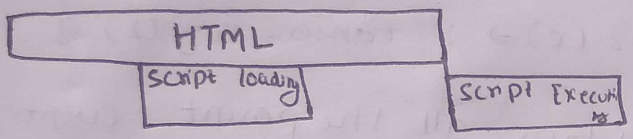
(1) <script> → HTML file will be parsed until the script file is hit, at that point parsing will stop and a request will be made to fetch the file (if external). Then script be executed and then HTML parsing resumed.



(2) <script async> → Downloads the file during HTML parsing will pause the HTML parser to execute it when it has finished downloading.



③ <script defer> → defer downloads the file during HTML parsing and will execute it after the parser has completed. defer scripts are also guranteed to execute in order that they that they appear in document.



When should I use what?

→ if script is modular & does not rely on other scripts then use async

→ if script relies then use defer.

→ if script is small & is relied upon by an async script then use an inline script with no attr placed above the async scripts.