ES-6 — Ecma Script.

ES-6 - Features:

Var, let, const, Template literates, string Interpolation, function.

Function Expression, hoisting default parameter spread (or) Rest parameter, operator.

Deep and shallow copy → arrow function.

Template literals — 2015

" " + → string Interpolation. (multiple - use not)

↓ Back_tricks.

---

hoisting

Call the function name() before function.

Ex: add() → hoisting

function add() {
logic
}

→ declare
Argument passing.
return.

Ex:

error

test() read ↑A

skip Let msg = "hello"

function test() {
console.log(msg)
}

↙ Next read

msg-not defined

function → before calling

declaration Let msg = "hello"

test()

function test() {
console.log(msg
}

Output: hello

Function Expression → not before calling

A function Expression has to be stored a Varibable and can be accessed using Variable Name.

Ex:
```
let even = function (num) {
    return num % 2 == 0
};
console.log (even(10));
```

```
let even = (num) => {
    return num % 2 == 0
}
console.log (even(10))
```

Default parameter:

Ex:
```
function greet (name) {
    console.log ("hi", name, "welcome") }
greet ("ram")
```
output: hi ram welcome

```
greet ()
```
output: hi undefined welcome.

Ex:
```
function greet (name = 'user')
gkeet ()
```
output: hi user welcome.

Arrow function:

```
(parameter) => {
    logic }
```

Ex:
```
let even = num => num % 2 == 0
console.log (even(10))
```
→ 10 (output)

Spread / Rest

Act like function - ல் இருக்கும் local variable

Multiple Arguments stored in Array.

Syntax: ...

Spread parameter:

Ex:
```
let check = function (...arg) {
    conside.log (arg)
let a = 1
for (value of arg) {
    a *= value }
return a }
console.log (check(1,2,3,4,5))
```

→ [1,2,3,4,5] → local variable array store

→ arg = [1,2,3,4,5]

→ for of loop

Output: 120

Multiplication Answer

Parameter pass [illegible] use arguments

```
Let  Check = function() {
        console.log (arguments)   → arguments = [1,2,3,4,5]
           Let  sum = 0
   for (i=0; i< arguments.length ; i++ ){
           sum + = arguments[i] }
              return  sum }
Console.log (Check (1,2,3,4,5))
```

<u>Output</u>: 15

Shallow and Deep Copy

Shallow - Reference (non-primitive) → copy Address.

Deep - primitive → copy the value.

Ex:

Primitive - Deep

```
var a=10
var b=a;
   b = 20;
console.log (a) → 10
console.log (b) → 20
```

No affect → <u>Reason</u>: copy only value.

Reference - Shallow

```
var arr1 = [1,2,3,4]
var arr2 = arr1
   arr2[0]=10
console.log(arr1)=[10,2,3,4]
console.log(arr2)=[10,2,3,4]
```

affect → <u>Reason</u>: coppy the whole Address

Change shallow to Deep:

<u>For array</u>

```
var arr2 = [... arr1]   arr2[0]=10
```

(... → spread operator; use array simple)

```
Console.log (arr1) = [1,2,3,4].
Console.log (arr2) = [10,2,3,4].
```

<u>For object:</u>

```
var obj1 = {name : "anna",
            age : 24 , }
var obj2 = {... obj1}   → use object
obj2.name = "Raju"
```

```
console.log (obj1) = {name : "anna",
                      age : 24 }

console.log (obj2) = {name : "Raj",
                      age : 24 }
```

---

object → string ⇒ JSON.stringify.

string → object ⇒ JSON.parse

Next - object → Changing Deep Copy.

```
var obj1 = { name : "John",
             age : 24,
             mark : { T : 25,
                      E : 40, }
           }

Obj2 = JSON.parse (JSON.stringify (Obj1))
```

Email and password Array of object:

```
<body>
    <form id = "form" >
        <input type="email" id = "email" value = "email
        <input type = "password" id = "pass" value = "pass
    </form>

<script>
    Let resform = document.getElementById ("form")
resform.addEventListner ("submit", function (e) {
        e.prevent Default ();
    const remail = document.getElementById ("emai
    const rpass = document.getElementById ("pass")
```

```
const formdata = {
        email : remail,
        pass : rpass
            }
        storedata (formdata)
    })
function storedata (formdata) {
    const storeformdata = JSON.passe (localStorage.
                getItem ("formdata")) || [ ]
    storeformdata.push (formdata)
Local storage.setItem ('formdata', JSON.stringify
                (storeformdata))
        }
</script>
</body>
```

invoke promise — promise (object) → call back function.
use new keyword

Promise, promise.all, promise.chaining.

promise → success (resolve) using then → success parameter pass
failure (reject) using catch → failure (error

promise → 2 argument இருக்கும். 2-ம் function தான்.
→ New use

Syntax: new promise ((resolve, reject)) ⇒
2 argument are call back fch

Ex:
```
<script>
    const result = new promise ((resolve, reject) ⇒
        {
        let pass = true
        if (pass){
            resolve ()
        }
        else { reject ()} })
```

result.then(success()).catch(failure()) → calling

```
function success(){
    console.log("you are pass")
}
function failure(){
    console.log("you are fail")
}
```

Output: you give boolean variable pass = true.

Output : you are pass.

Pass = failure.

output : you are fail

---

loading progress Bar:

html:

```
<div id="progress">
    <div id="bar"></div>
</div>
<p id="loading"></p>
```

JS:

```
const bar = document.getElementById("bar")
const msg = document.getElementById("loding")
let barwidth = 0;
const download = () => {
    barwidth++;
    bar.style.width = barwidth + "%"
loading msg.innerHTML = (barwidth == 100) ?
    `${barwidth}%` : `${barwidth}% progress`
}
let interval = setInterval(() => {
```

```
If (barwidth ==100) {
        clearInterval (interval Id)
        }
else {
    download()
    }
}, 1000)
```

---

setTimeOut:

1) function menu (fut1, fut2) {
output.innerHTML = 'I had ${fut1} and ${fut2}'
}

setTimeout (menu, "apple", "Orange")
// function → time → name

output:

I had orange and undefined.

---

2) function menu (f1, f2) {
output.innerHTML = "I had ${f1} and ${f2}'
}

setTimeout (menu, 1000, 'apple', 'orange')

output:

I had apple and orange.

Every:

It act like AND It return true. if the function return true for all element. It return false if the function return false for the statement It give boolean value.

Ex:

```
Let num = [25, 35, 45, 65]
Let res = num.every ((a) => {
    return a > 20;
})
log.res
```

output: true.

```
Let num = [20, 35, 45, 65]
Let res = num.every ((a) =>
    a > 20)
log (res)
```

output: false.

use if:

```
if (res==true){
    log("pass")
}
else { log("fail") }
```

SOME: It act like OR. It return boolean value. It return true if the function return for one element.

Ex:

```
let num=[25,35,45,65].
Let res = num.some((a)=>{
    return a>35
})
log(res)
```

output:

True.

Call back function:

It is a function passed as an argument to another function.

Ex:

```
function greet(name, msg, call){
    log(name, msg)
    call()
}
greet('anna', 'welcome', 'msg')
→ function msg(){
    log("welcome")
}
```

Output:

anna welcome
welcome

Promise

Ex: 2

```
const result = new promise((resolve,reject)=>{
    Let pass = false
    if (pass){
        resolve(250)
}
```

```
else {
    reject (250)
    })

result. then ((total) => log ('you are pass mark:
                                ${total}'))

.catch ((total) => log ('you are fail mark:
                                ${total}'));
```

EX:3

```
function random (){
return new promise ((resolve , reject) => {
    Let rand = math. floor (math .random()*8)
    if (rand ==0){
        resolve()
        }
    else {
        reject()
        }
    })
}
random (). then (() => log ("success")). catch (() => log
                                ("unsuccess")).
```

Change callback hell / to / promise:

```
function download (url, call ){
    set timeOut (() => {
        log ('downlodding ${url}')
        call (url)
        }, 1000)

Const url1= "https. //www. google1"
Const url2= "https. //www. google2"
Const url3= "https. //www. google3"

download (url, function (url){
    log ('process ${url}')
    download (url2 , function (url){
        log ('process ${url}')
        download (url3, function (url){
            log ('process ${url}') });
        }); });
```

output:

download
process
download
process
download
process .

Callback function:

Ex: 2:

```
function even(number){
    return number%2==0
}
let numbers = [1,2,3,4,5,6,7]
function filter(numbers, fn){
    let result = [];
    for (number of numbers){
        if (fn(number)){
            result.push(number)
        }
    return result
}}
log(filter(number, even)
```

output: 2,4,6

```
Let num= [1,2,3,4,5,6]
let filter = (num,fn)=>{
    Let result=[]
for (num2 of num)){
    if (fn(num2)){
    result.
    return result
    }
    }
log(filter(num, function without name
    function(num2){
    return num%2==0
    })
```

Async: → Hotel (asynchronous)

function work simultaneously.

Ex:

```
const url = "http//www.google"
function download(url){
  setTimeout(()=>{
   log('Download ${url}')
  },1000)
function process(url){
    log('process ${url}')
}
download(url)
process(url)
```
} → calling

output:

process

download.

Sync: → bank (synchronous)

function work step by step

Ex:

```
const url = "http://www.google"
function download(url, call){
  setTimeout(()=>{
   log('Download ${url}')
    call(url)
  },1000)
function process(url){
    log('process ${url}')
}
download(url, process)
```

For Access More resolve (or) reject [Change callback hell To promise]

```
let url1 = new Promise ((resolve, reject) => {
    const geturl = true
    const url = "http// www.google1"
if (geturl){
    setTimeout (resolve, 1000, 'Downloading ${url}')
    }
  else {
    reject ()
    }
})
Let url2 = new Promise ((resolve, reject) => {
    const geturl=true
    const url = "http// www.google2"
if (geturl) {
    setTimeout (resolve, 2000, 'Downloading ${url}'
    }
  else {
    reject ()
    }
})
```

array

```
url1.then ((msg) => console.log (msg)).catch (() => log ("nothing"))
url2 .then ((msg) => log (msg)).catch (() => log ("nothing"))
```

---

promise.all(): If one condition is false. It not print any value. It return value in an array.

```
promise.all ([url1, url2]).then ((msg) => log (msg)).catch
    ((mum) => log (mum, "failure"))
```

Ex: use promise:

1st process complete
2nd process complete.

Input [10, 20]

output 30.

```
Let value1 = new promise ((resolve, reject) => {
    resolve(10)
})
```

```
let value = new promise ((resolve, reject) => {
        resolve(30)
    }) → console.log(a) → return promise
                          ↗ array
promise.all ([a,b]).then((msg) => {
    Let s = msg.reduce (a,b) {
        return a+b
    }
    console.log (s)
})
```

Output: 30

---

promise.any() If one condition is true. It print value in an array form promise.any([url1, url2]).then((msg) => log(msg)).catch(num) => log(num, "failure"))

---

promise Chaining:

Ex:
```
const promise = new promise ((resolve, reject) => {
        resolve()
})
promise . then (() => log ("first")) . then (() => {
        return new promise ((resolve, reject) => {
            setTimeOut (() => { log("second")
                resolve ()
            }, 2000 )
        }) . then (() => {
            setTimeout (() => { log ("Third") }, 1000 ) }).
```

API: * Application programming interface. (connect)
* It return promise.

promise:
* pending
* fulfill
* result

API
1) get
2) post
3) update
4) delete

Step-1: fetch → get url to fetch
JSON → convert url to value.

Objects:

```
fetch ('url').then ((res) => res.JSON ()).
then ((msg) => {
        for (mess in msg) {
            log (mess) → Type movie
            log (mess, msg [mess]) →
            log (msg.type) → love
        }
})
```

{ id : 1,
type : "love",
movie : romi
}

type : love
move : Remo

Array of object:

```
const row = document.quary selector ("#row");
fetch ('https://api.tvmaze.com/shows/1/episodes')
.then ((res) => res.JSON ())
.then ((msg) => {
        for (message of msg) {
            Create (message) →
            log (message)
        }
})
```

} = calling function

```
const Create = (data) => {
    row.innerHTML += '
        <div>
        <img src = "${image}">
        </div>
        <p> ${rating} </p>
        <p> ${Summary} </p>'
```

Sep-25: Destructuring:

Array → declare name for each value in an array.

Array Destructuring:

```
Let name = ["John", "Doe"]
Let [firstname, lastname] = ["John", "Doe"] (or) name
log (firstname)
```

for Change Value:

```
firstname = "Anna"
log (firstname)
```

for add:

```
let name = ["John", "Doe"]
let [first, last, color = "Red"] = name
```

Output: log (color) = Red

2) object destructuring:

```
Let person = {
        name = "Anna",
        age = 25
        }

Let { name, age } = person.
log (age)
```

for Change name:

```
let {name : name1 , age : age1} = Person.
log (age1)
```

name change பண்ணலாம்

for add keys:

```
let {name : name1, age : age1, color = "Red"} = Person
log (color) = red.
```

promise: pending, fulfilled → resolve, reject, result.

Async and Await:

use await for async.

1) true → resolve.

2) false → print reject value as error (uncaugh (in promise)

so use try and catch shop.

Async → try (success)
catch (error)

2x-1:

```
function status () {

}
log (status())
```

output: undefined

Because it has value and it doesn't return any values.

for EX:
```
function status () {
    log ("Hi")
}
log (status ()) → Hi
```

EX-2:
```
async function status (){
    log ("Hello")
}
log (status ())
```

output:
Hello
▶ Promise {<fulfilled>: undefined}.

EX: 3: promise + Async + Await

```
Let url = new promise ((resolve, reject ) => {
    const geturl = true, false
    const url = "https://www.google.com"
if( geturl ) {
    setTimeOut (resolve, 3000, 'Downloading ${url}')
else {
    reject ("you are rejected")
    })
```

```
async function status(){
    try {
        let res = await url;
        log(res)
    }
    catch(err){
        log(err)
    }
}
log(status())
```

output: true
▲ Promise {<pending>}
Downloading https://www.google.

output: false
▲ Promise {<pending>}
you are rejected.

This:

```
function check(){
    log(this)
}
check()
log(this)
```

→ Inside function u give. output ▲window

→ outside

This inside object:

```
let person = {
    name: 'John Doe',
    getname: function(){
        log(this.name)
    }
}
person.getname()
```

→ John Doe

Access this keyword value in another function:

```
let person = {
    name: 'John',
    getname: function(){
        log(this.name)
    }
}
```

→ undefined
use person.name
(Person.name) → John

```
setTimeOut(person.getname, 1000)
```

→ *It gives no values in output
* Because function inside object can't be access in another function

(or)

```
let f = person.name
setTimeout(f, 1000)
```

→ output: John

* We can access function inside object in another function using this method.

this and bind:

function inside object can't be access in another object but we can access with bind method

```
let person = {
    name: 'John',
    getname: function(){
        log(this.name)
    }
}
let per = {
    name: 'Resh'
    age: 25
}
let f = person.getname.bind(per)
setTimeout(f, 1000) → Resh
```

This and call and apply:

```
const car = {
    name = 'care'
    start() {
        log('start the ${this.name)
    }
}
const aircraft = {
    name: 'aircraft'
    fly() {
        log("I am fly")
    }
}
```

car.start.call(aircraft) → start the aircraft

car.start.apply(aircraft) → start the aircraft

ex-2:

```
let person = {
    name: 'Resh',
    age: 25,
    getname(url) {
        log('Hi I am ${this.name} ${url}
    }
}
let per = {
    name = 'preeti'
    age: 23,
}
```

person.get.call(per, "welcome") → Hi I am preeti welcome

person.get.apply(per, ['welcome']) → [] use → Hi I am preeti welcome

Bind: Let server = {

```
    name : 'Apache',
    json : prompt ("Enter the number")
    turnon() {
        log ('The ${this.name} is on')
    }
}
Let computer = {
    name : 'Dele'
}
server.turon.bind (computer)
server.turon.bind (computer)() → Dele is on
```

→ f() { console.log (This ${this.name} is on') }

Class - oops • object oriented programming language

Ex: Reference Day Data Type.

Syntax: class Person { } → object

Person → Always start with new name capital because it is constructor

Classname

Syntax — condition:

1) Put Class
2) class name Always start with capital letter.
3) variable → Direct ಆಗಿ use ಮಾಡಲಾಗುವುದಿಲ್ಲ.
4) Methods → use in Direct

For Ex: Methods - use in direct calling.

```
class Person {
    constructor (name, age, profession) {
        this.name = name
        this.age = age
        this.pro = profession }

    details () {
        log ('My name is ${this.name}')
    }
}
```

Must use this, You Don't put this, it is not worked.

Calling place: class name, constructor name Call ಮಾಡಲು ಬಳಸಿ Same - ಆ ರೀತಿಯಲ್ಲಿ.

Ex: Let per = new person ("Dae", 20, "Teacher")   argument pass பண்ணி அதை மேல Call பண்ணு.

per. Details()

output: My name is Dae.

Another One: Static

If you put any variable name in static, you can access the variable only using Classname. variable name.

For Ex:

```
class Student {

    static number = 0

    constructor (name, age, profession){
        this.N = name
        this.A = age
        this.P = profession
        person. number ++ }
    }

Let Stu1 = new Student ("Apta", 15, "Teacher")
Let Stu2 = new Student (" kalai", 20, " Developer")
log(person.number)
```

output: 2.

Methods – using static:

For Ex:

```
Class Person {

    constructor (name, age){
        this.N = name
        this.A = age

    static Detail() { log('${ name this.N} is powerful'}
    }
```

Let -ல store பண்ணி call பண்ண முடியாது. use only Classname. methods.

Person. Detail()

output: person is powerful → this.N automatically assign person (Classname).

Value கொடுத்தாலும் Classname தான் assign பண்ணும்.

Inheretum Inheretence

Parent class name — constructor used in Child class name. but not access the child class name constructor in Parent class name.

Syntax:

```
class parentclassname {
    constructor () {
        Details
    }
}
class Childclassname extends parentclass name { }
```

For Ex:

```
class Parentperson {
    constructor (name, age){
        this.N = name
        this.A = age
    }
    Details () {
        log ("my name is ${this.N} and age is
                ${this.A} ')
    }
}

Let per1 = new Parentperson ("John", 25)

class Childperson { extends Parentperson {
    newmsg () {
        log ("hello") ; log ("my name is ${this.N}")
    }
}
```

Calling:

```
Let per2 = new Childperson ("Veni", 35)
per2.Details()
```

Output:

my name is Veni and age is 35.

but message not access. because Parent

Child -ல parent constructor Access பண்ண - use super

for Ex:

```
class Childperson extends Parentperson {
    constructor (name, age)
      super (name, age) ---> 
  msg () {
      log ('this.N)
      log (this.A)
      }
  }
```

Child உள்ள access பண்ண super. அதாவது, Child function parent key இருந்தா அது access பண்ண constructor and super use பண்ணலாம்.

super -ல (parent உள்ள argument தரும் வேலை செய்யும் இருக்கும்.

calling:

```
Let per1 = new Childperson ("Kalai", 20)
per1. msg()
```

output:

```
kalai
20
```

---

for Ex:

```
Class   Parent vahicle {
   constructor (make, model, year) {
        this.m = make.
        this.mo = model
        this.y = year.
          }
  detail () {
       log (this.n, thismo, this.y)
          }
    }
Class  Childvehicle  extends  Parent vehicle {
   constructor (make, model, year, Type) {
   super (make, model, year) ---> type வேண்டாம்.
    this.type = type }
  Childdetail () {
        log (this.m, this.type)
          }
Let per = new  Childvehicle ("bike", "lenova", 2023, "2wheels")
Per.child detail ()
```

அது Parent -ல இல்ல.

output: bike
2 wheels.

export - கொடுக்கிறது / import - வாங்குதல்

```
<script src="main.js"></script>
<script src="main.js" type="module"></script>
```

→ export + import Methods

→ app.js = file — Export

```
class Calculation {
    Solve() {
        console.log("welcome")
    }
}

Let a=10, b=20, result

function sum() {
    result = a*b; return result
}

function mul() {
    result = a*b
    return result
}
```

---

way of export:

```
export default calcalation
export { a, b, result, sum, mul }
```

→ ஒரு file-ல் ஒரே ஒரு default தான் இருக்கும்.

→ multiple purpose use

another one:

```
export class calculation {
    Solve() { log("welcome") } }
export Let a=10, b=20, result
export function sum() { return a+b }
export function mul() { return a*b }
```

---

import → main.js :

```
import calculation, {a, b, result, sum, mul} from "./app.js"
```

```
Let calc = new calculation()
calc.solve()

sum()
log(result)  --> 30

mul()
log(result)  --> 800
```

React -JS → 2013 - open browser. but Since 2011.

* JS Library React
* Declarative
* created by facebook, was created in 2011 by Jordan walke.

* JS framework angular
* component based

Based on components:

header.js, about.js → create individual page for each section

---

Declarative:

JS, HTML, CSS → JSX → JS XML notation.

with declarative you just tell what to do and in imperative you also tell how to do (too mach of intrauctions or in other words (coding steps))

SPA:

* SPA - Single page Application.
* Application operates in Single web page.
* instead of loading separate html pages for different interaction, SPA dynamically update the content on the same page.
* It initially loaded the entire content and update parts of the page dynamically.