22/11/21 — Java-script :-

Uses: → web apps, smart watch apps, IOT apps
→ mobile apps, desktop apps, API's, cloud services.

⇒ (script) ⇒ used to connect JS to HTML.

⇒ console.log("Hello World").

⇒ Declaration of variables :-

int a; (C language).

* Variable : A variable is a "named storage" for data.

declaring: var variable-name; (or) let variable-name; (or) const variable-name;

```
Var a= 10;
console.log ('Value of a is', a).
Var a=123;
Console.log("Value of a is ", a)
```

∵ Value of 'a' using 'var' declaration can be modified.

```
const b= 200;
console.log("Value of b is", b)
b=300;
console.log("Value of b is", b)
```

∵ Value of 'b' cannot be modified as using 'const' declaration variable cannot be modified

Output:-

Value of a is 10
Value of a is 123
Value of b is 200
Error ——— —

```
Const empName = "Ravi kumar";
Var salary= 50000;
Var age;
```

* Dynamically typed nature of JS :-

⇒ `type of x` → used to know data type of variable declared.

⇒ initially it is undefined until a value is initialized to variable.

⇒ x=123; || x='hello'; | x = true;

console.log("datatype of x is", typeof x)

Output:

datatype of x is number | string | boolean.

---

* ~~Uppercase~~ lower : sum of two numbers

* ~~lowercase~~ upper : SUM OF TWO NUMBERS.

* Camelcase : sumOfTwoNumbers.

---

var a=100; var b=200;

var sum = a+b;

console.log("result =", sum)

var div = a/b; (quotient).

var mod = a%b; (remainder).

var pow = a**b; (a^b)

(==) [Compares only data but not the datatype].

(===) [first compares datatype then data]

${first number} [prints value of first number].

Control statements :-

→ if

→ if else

→ if else if

→ for loop

→ while loop

→ do while loop

→ switch statement.

```
if (condition)
{
} //if block.
```

```
if else
if (condition)
  {
  }
else{
  }.
```

```
if else if.
if (condition-1) {
}
else if (condition-2){
  }
else if (condition-3){
  }.
```

```
var a=100;
var b=200;
if (a>b) {
  console.log (first, "is big")
}
else {
  console.log (sb, "is big")
```

```
for(var i=1; i<=3; i=i+1){
  console.log ("hello").
}.
```

Output:-

hello (7 times)

hello

+= compound addition
-= compound subtraction

*= compound multiplication
/= compound division

%= compound modulus.

Conditional operator :-(?:)

Condition ? expression-1 : expression-2;

2/11/21

* Functions :-

```
function fname (param1, param2, ....)
{
  statement 1;
  statement 2;
  statement 3;
  return output;
}
```

```
//function declaration.
function findSumOfNumbers(a,b){
var sum=a+b
console.log('sum is ${sum}')
return sum;
}
```

```
//call
findSumOfNumbers(10,20)
findSumOfNumbers(123,345).
```

Output :-

sum is 30

sum is 468.

findSumOfNumbers (10,20,30,40, 50,60,70).

* (It considers only two arguments and two values even if more than 2 values are assigned to it)

```
function findsum (a,b) {(// function declaration)
    return a+b;
}
var a = function (a,b) { (// function expression)
    return a+b
}
a(10,20) //(call)
```

```
var test = (a,b) => { (// arrow function)
    return a+b
}
```
↳ a+b (directly).

```
var test = (a,b) => a+b.
var result = test(10,20)
console.log(result)
```

* Global: Value declared as global is can be used or accessed by any no. of functions.

⇒ Any variable declared in function cannot be accessed globally.

* Any variable declared in (block level)

→ can be accessed any where in the function.

* "let" is used for accessing of variables at block level.

```
// global
var a = 10
function test1(){
  console.log('value of a in test1
      is ${a}')
}
function test2(){
  console.log('value of a
  in test2 is ${a}')
}
test1()
test2()
```

Output:-

value of a in test1 is 10.

Value of a in test2 is 10.

24/11/21

Arrays: Pack of data.

→ Arrays → Objects.

* both are used in storing a pack of data.

let a=10.
let b=20
let c=30.

arr = [10, 20, 30, 40, 50, 60, 20].

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewpoint" content="width=device-width,
        initial-scale=1.0">
    <title>Document</title>
    <link rel="stylesheet" href="1.css">
</head>
<body>
    <script src="1.js"></script>
</body>
</html>
```

```
let numberArray=[10, 20, 30]
Console.log(numberArray)
```

Output:-

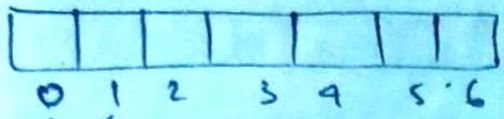
(3) [10, 20, 30].

```
Console.log(numberArray[0])
```

Output:

10.

Index: Relative distance from starting of array (elements)



distance → distance '1' as one element is present at index '0'.
is 0 as no element is present before this

Iteration of array :-

```
//for loop.                                  → numbersArray.length.
for(let index = 0; index < 3; index ++){
    console.log(numbersArray[index])
}

//while loop.
let index = 0;
while(index < numbersArray.length){
    console.log(numbersArray[index])
    index++
}.

//for-of loop (present only in Javascript).
[for(let 'v' of arr-name){
}] → syntax

for(let v of numbersArray){
    console.log(numbersArray
    console.log(v)
}.
```

Array Operations :-

→ insertion (First - Middle - Last)

→ deletion (" - " - ")

→ method (Similar to function)

function test() { } ==== > test()

* function can be called directly.

* method cannot be called directly like functions.

[obj.method()] → in this way

→ Inserting Elements into an array :-

At Begin: unshift (1,2) → 'n' number can be inserted → length of new array.

Initial

10	20	30	40

↳ Array.

1	2	10	20	30	40

```
d1 = NumberArray.unshift(1,2).
```

Output :- 6. [Gives size).

At End :- push(100,200).

10	20	30	40	100	200

Index based insertion :- splice (2, 0, 123) → [ ] (element adding)

index (Replacing or just adding)

10	20	123	30	40

splice (2, 1, 123).

10	20	123	40.	

splice (3, 0, 45, 55).

10	20	30	45	55	40.

Removing elements :-

10	20	30	40

begin: shift(). →

20	30	40

end: pop(). →

10	20	30

index based deletion: splice(2,1).

10	20	40

Update: splice(1,1,200) →

10	200	30	40

Error handling :-

* When an error is occured in JS then the program is terminated/stopped

Eg:
```
let a=10;
console.log("a is ",a)
console.log("b is ",b)
console.log("hello")
console.log("how are you")
```

Output:-
```
a is 10.
b is not defined.
```

* try{ }.
* catch(){ }.
  arg is passed

```
try{
  console.log("b is",b)
}
catch(e){ }  ← maybe anything.
  → console.log("error occured")
  consde.error("error occured",err.message)
```

Output:-
```
a is 10.
error occured
hello
how are you.
```

13/12/21

ES6 methods of array (Higher order functions):

→ Callback function.
→ Higher orda function: Fn either receives a fn as arg. or, return a function.
→ A function which can pass as an arg to another function.

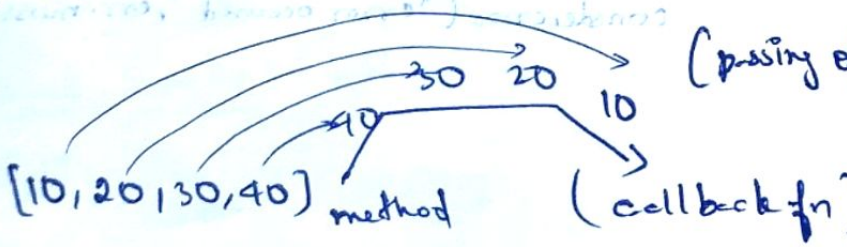
```
const test = (a) => {
    conole.log("Value of a is ", a)
}
//Call.
test (1,2]) or test(10).
```

general function call.

```
const test1 = () => {
    xhurn 'test1"
}
const test2 = (a) => {
    console.log("hello")
    console.log (a())
}
//call test 2
test 2 (test1).
```

Output:-
hello.
test1.

Array functions (ES6):-



(passing of elements one by one)

» Callback fn executes until (size of array) times.

① filter:- Used in filteration of data &. based up on condition.

```
let arr2 [2,-98,345,76,99]
let values = arr.filter (function (element)){
        xhurn element>70
})
console.log( values).
```

Output:-
[345,76]

values zarr.filter (element => element > 70)
console.log(values)

* Filter method can be used only to filter or select, but not for modification.

② Map : modification of data

let modifiedData = arr.map (element => element + 10)
console.log (modifiedData).

Output: [31, -88, 355, 86, 59]

(element >= 10) ↓↓ (boolean format) ↑ [True, false, true, true, true]

③ Foreach {- to iterate array.

arr.forEach (element, index) => {
  console.log (value at index ${index} is ${element})
})

④ reduce : Reduction of code.

[1,2,3].reduce ((accumulator, element) => {}) ⇒ syntax

let sum = arr.reduce((acc, element) => acc + element)
console.log("sum=", sum).

Output: sum = 100

⑤ Find : find an element.

⑥ FindIndex : finding index of an element.

let r = a.find (e => e == 76) || (find Index) ⇒ replace.
if (r == undefined){
  console.log("Unsuccessful")
} else {
  console.log("Element found")
}

14/12/21.

Object:- Representation of physical existing entity

Object creation ⇒ ① Object literal ⇒ { key: value; key: value; } objects of diff type.

② Constructor function

↓

Multiple objects of same type.

Creation of obj:-

```
const obj.person = {
    id :100;
    name: 'ravi';
    age : 21,
    city : 'hyderabad'
};
```

console.log(typeof(person)) ⇒ object

console.log(person) ⇒ {id:100, name:'ravi', age:21, city:'hyderabad'}

* obj is the unordered collection of data

Properties:- ① obj.key ⇒ value
obj['key'] ⇒ value } returns value of key

console.log(person.name) ⇒ (ravi).

② Adding new keys dynamically.

obj.new-key = value-new

③ Delete objects from it.

delete obj.key.

⇒ delete person.abcd (abcd 'key' is not present).

⇒ The error is not thrown in JS, if key is present it returns its value else it does not return anything even no error/warning message is displayed.

⇒ No compile time error.

④ Modifying of keys is also possible.

`obj.key = new-value`

person.name = 'ravi kumar' ⇒ {'ravi kumar"}

⑤ Iterating an object:

```
for-in loop.
    for (let v in obj)
    {
        ——— ——
    }. ——— —
```

key ↑

v - receives the keys in the syntax.

```
for (let v in person){
    console.log(v).
}.
```

key ↓

⇒ O/P: Id
Name
Age
city
designation

only keys are returned.

console.log(person[x]) ⇒ O/P: 100
ravi kumar
21
hyd.
Manager

values are printed.

```
for (let k in person){
    console.log(`${k} is ${person[k]}`)
}
```

both keys & values are returned

* Complex objects:

Primitives: Numbers, strings, boolean.

Non-primitives: Arrays, objects, functions.

```
skills: ['html', 'css', 'bootstrap'],
address: {
    street: 'kpHB'
    city: 'Hyd'
    pincode: 500085.
} }.
```

Accessing them
⇒ person.address.city.
O/P ⇒ hyd.

```
getsalary : function () {
    let hra = this.basic * 0.15 .
    let da = this.basic * 0.10 .
    let salary = this.basic + hra + da .
    return salary .
}
```

"this" points/refers to the current object else if its not given then it searches for basic outside the obj and performs the operations on it.

* Constructor function :- (Multiple objs of same type)
* First type is defined as all objs are of same type.

```
function functionname (params)
{
    basic logic
}
```
} Normal fn.

```
function Function Name (params)
{
    ~~business logic~~
    object initialization logic
}
```
} Constructor fn.

» function Person (id, name, age, skills, add, basic .

```
// an empty obj is created & xfered by this.
this.id = id;
this.name = name;
this.age = age;
this.skills = skills.
this.add = add.
this.basic = basic;
this.getsalary = function () {
    let hra = this.basic * 0.15;
    .
    .
    return salary;
}
```

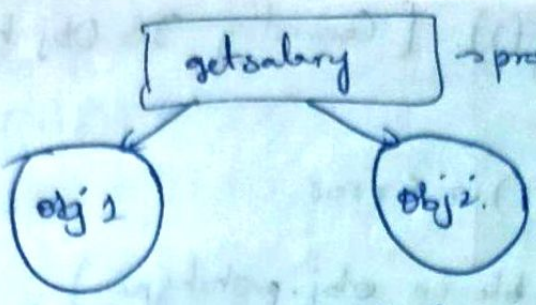
// create person1 obj.

```
let person1 = new person (100, 'ravi', 21, ['html'], ; console.log(
console.log(person1).
```

20,000

person1.getsalary()).

15/12/21



*getsalary function is common for all the people working. So writing its in obj makes the code lengthy unnecessarily.

*so it can be written outside the obj and accessed by every obj by prototype property.

Prototype property:-

⇒ Every object contains this property by default.

⇒ It is a space where all objects inherits features

object	Object.
↓	↓
Datatype	constructor (to create object datatype)

⇒ console.log(person.prototype) ⇒ {constructor: f}

⇒ constructor: f person(id, name ---)

[[Prototype]] : Object.

//add method to person prototype.

```
person.prototype.getsalary = function(){
    . . . . .
    . . . .
    . — — .
    return salary
};
```

console.log(person1).

O/P: id : —

name : —

Object.prototype.

↓ Inherits from prototype of

myobj1.prototype.

↓ " "

myobj2.prototype

* Prototype : Object.

⇒ getsalary : f().

constructor: f Person(id, ——

[Prototype]: Object.

console.log(person1.toString()). [Converting JS Obj to str].

• O/P = [object Object]

(, , ...abcd()) → Error.

(abcd is not even available in obj.prototype)

⇒ So it gives an error.

Classes :- (to create shape of object).

• New syntax for constructor function. (ES6)

* Constructor is present only in JS.

```
class ClassName {
        constructor(){}
        methods()
    }

class PersonType {
        constructor(){
          // obj initialization logic. → (id, name ——————)
        }
        //methods (function initialized inside obj is called method).
        getsalary(){
          // business logic → (getsalary() * same)
        }
    }
```

* Prototype is directly added. without using prototype keyword.

* Types of parameters :-

→ function test(a) {
        console.log("a is ", a)
    }

→ test() // Call. [O/P : a is undefined.

test(10) // call [O/P a is 10.

} It doesnot show error. If finding it is difficult.

① default value parameter:

* function test(a=1){
  }
  test() // O/P ⇒ a is 1.

② Rest parameters:

function tst (a) {
}
test(10,20,30). //O/P ⇒ a is 10.

function tst (a,b){
}
O/P, a is 10
b is 20.

Rest parameter : (...a)

function tst(...a)
[Capable of both args].
tst(10,20,30,40,50,60,70)

O/P: a is [10,20,30,40,50,60,70]

* Multiple values are taken by single rest arg.

function (b, ...a)
accepted

function (...a,b)
not accepted, technical voilation
∵ rest parameter itself accepts both values

* **Spread Syntax:-**

* to merge arrays/objects.
* to create copy of array/object.

let arr1 = [10,20]
let arr2 = [30,40]
let mergedArray = [...arr1,...arr2].
console.log(mergedArray)
O/P: [10,20,30,40]

Any no. of arrays can be merged.

```
let obj1 = {
    a:10,
    b:20
}
```

```
let obj2 = {
    x:100,
    y:200
}
```

```
let mergedObject = {...obj1, ...obj2}
console.log(mergedObject).
```

O/P - {a:10, b:20, x:100, y:200}

Creating copy of array

```
let skills = ['html', 'css']
let copyskills = [...skills]
skills.push("bootstrap")
console.log("skills array", skills)
console.log("copy of skills", copyskills).
```

O/P: skills array
['html', 'css', 'bootstrap']
copy of skills
['html', 'css']

Creating copy of obj:-

```
let person = {
    id: 100,
    name: 'ravi',
}
let copyPerson = {...person}
person.city = "hyd"
console.log("person")
console.log(copyperson)
```

O/P: {id:100, name:'ravi', city:'hyd'}
{id:100, name:'ravi'}

Destructuring :-

* Unpacking ⇒ array / obj

array ⇒ arr[index]

object ⇒ obj.key / obj['key'] } get index values

```
let arr = [10, 20, 30]
let [, , c] = arr.
console.log(c).
```

o/p ⇒ 30.

```
let obj = {
    x: 100,
    y: 200
}
let {y} = obj;
console.log(y);
```

o/p: 200.

20/12/21

Modules :- (refactoring)

⇒ many no. of modules can be created for a single one.

Eg: main.js ⇒ users
⇒ products.

users.js ⇒ module.

products.js ⇒ module.

* Communication among modules is established by export & import

export :- two types :- * default export
* named export.

Import : import var-name from 'path of module'.

* type = "module" is mentioned in the html file while linking both of them.

main.js

```
import users from './users.js';

export
console.log(users.obj);

import {products from
'./products.js';
```

users.js

```
let users = ['ravi', 'raju'
    'kiran']

export default users;
let userobj = {
    city: 'hyd',
    basic: 500006
};
```

- exp default {users, userobj}
  must be mentioned in {} braces.

products.js

```
export (named export)
let products = ['tv', 'music
player', 'washing machine']
```

> same name must be used while importing.

- Inner functions & closure :- | closure = inner fn + state (data) around it.

```
function test(){ (//outer function)
    function testinner(){ (Inner fn to write private logic)
    }
}
```

(:' its accessible only by test fn but not the other ones).

- Testinner(). // It shows not defined & must be called inside the fn itself or
  return testinner.

```
let result = test()
result().
```

```
function testinner() {
    let inside = 123;
    console.log("Outside is ", outside)
    console.log("Inside is ", inside)
}
```

O/P :-

Outside is 100
Inside is 123

* Inner function can access data declared outside of it, but outer inner cannot access inner fn's data.

```
function testInner() {
    let inside = 128;
    let sum = outside + inside;
    return sum;
}
```

O/P:
223.

```
function test() {
    let outside = 100;
    function testInner() {
    }
    return testInner;
}
let result = test();
console.log(result());
```

* According