JUNCTION UP TRAINEE

MEENAKSHI SURYAVANSHI

SODHI RAJPUT

JAVASCRIPT NOTES

JAVASCRIPT

* History of JavaScript :→

→ Netscape Navigator (1995) → web browser
↳ Brendan Eich (Developed JS. in 10 days).
↳ Mocha
↳ Livescript
↓
JavaScript

→ Internet Explorer
↳ JScript (Language)

1996

JavaScript — JScript (ECMA Script)

* ECMA → International Organization to standardize programing languages

ES1 : 1997
↳ ECMA Script
ES2
ES3
ES4
ES5 : 2009 (lots of new features)
ES6 : 2015 (Biggest update in the history of js.
↳ Modern JavaScript

* babel is a JS compiler

TC 39
(Technical Community 39)

ES6 : ES 2015
ES7 : ES 2016
ES8 : ES 2017
→ official names

Note
⇒ Javascript is backward compatible
⇒ Javascript is not forward compatible.

Note → Shortcut to open console on browser = ctrl + shift + J

* To Link JS file in HTML file :→

In <head> tag →

```
<head>
<script src="file.js" defer></script>
</head>
```

In <body> tag →

```
<body>
<script src="file.js></script>
</body>
```

Program → To print string (Hello world) :→

e.g.

```
Console.log("Hello World"); // double quotes
console.log('Hello world'); // single quotes
Console.log(`Hello world`); // back tick
```

→ //Comments :- Ctrl + forward Slash / →

// Console.log can print something on Console.

* Variables :→ Variable are the name of location where we can store the data or information. we can change use that information later.

⇒ To declare variable :→

var firstName = "Meenakshi";
↑ Keyword ↑ variable name → assign I/p or value in variable
o/p →

⇒ To use a variable :→

console.log (firstName); o/p → Meenakshi

Note → Variable names are case sensitive.

⇒ To change value :→

firstName = "Suryavanshi";
console.log (firstName); o/p → Suryavanshi

Note ⇒ "use strict"; → to highlight errors.

* Rules for naming variables :→

1. We can't start with number

e.g. 1Value = 2; (Invalid)
Value1 = 2; (Valid)
console.log (value1 * 3); o/p → 8

2. We can use only underscore (_) or dollcor ($) symbols→

e.g. first_Name // snake case writing
_firstName
first$Name
$firstName

* Note → we can't use space
e.g. first Name (Invalid)

Note → firstName // camel Case writing.

⇒ Convention →

→ Start with small letter and use CamelCase.

---

* let Keyword →

We declare variable with let keyword.

→ block scope vs function scope.

⇒ var firstName = "Meenakshi"] valid
var firstName = "Suryavanshi"]

⇒ let firstName = "Meenakshi"] Invalid.
let firstName = "Suryavanshi"]

Note → we can't redeclare variable by using "let" keyword. But we can redeclare variable by using "var" keyword.

* declare constant →

we declare constant by using "const" keyword.

e.g. Const pi = 3.14;
console.log(pi * 2 * 2);

* String Indexing :→

let firstName = "Yogita";

character	=	Y	o	g	i	t	a
Index	=	0	1	2	3	4	5

⇒ to print character by indexing —

```
console.log (firstName[0]);
```

o/p ⇒ Y

⇒ Length of string :→ (property)

```
console.log (firstName.length);
```

⇒ last Index :— length - 1.

```
console.log (firstName (firstName.length - 1));
```

Note → spaces are also included to calculate length of a string.

* Methods of string →

→ trim()
→ toUpperCase()
→ toLowerCase()
→ slice()

examples-

⇒

```
let firstName = " Yogita " ;
console.log (firstName.length); // o/p = 8
firstName = firstName.trim(); // o/p = "Yogita"
console.log (firstName.length); // o/p = 6

firstName = firstName.toUpperCase();
console.log (firstName); // o/p → YOGITA
firstName = firstName.toLowerCase();
console.log (firstName); // o/p → yogita;
```

```
let newstring = firstName.slice(0,3);
console.log(newstring);
o/p → Yog ita
```

(or)

```
let newstring = firstName.slice(0,4);
console.log(newstring);
o/p → Yog
```

* typeof operator :-

⇒ Datatypes :- primitive datatypes -

1. → string
2. → number
3. → boolean
4. → undefined
5. → null
6. → BigInt
7. → Symbol.

Note :- typeof operator tell us the type of data

e.g.

```
let age = 20;
let firstName = "Mishti";
console.log(typeof age);        // number
console.log(typeof firstName);  // string
```

⇒ Convert number to string :→
we add empty string to convert number into string.

e.g: → console.log (age + " ");
→ console.log (typeof (age + " "));

o/p = string

e.g. age = 70;
age = age + " ";
console.log (typeof (age)); o/p = string

⇒ Convert string to number :→
→ we use (+) to convert string to number.

let mystr = + "30";
console.log (typeof mystr);

Note→ * let age = "10";
age = Number (age);
console.log (typeof age);

o/p = string

* let age = 12;
~~age = Number~~
age = string (12);
console.log (typeof age);

o/p ⇒ string.

* String concatenation →

```
let string1 = "Meenakshi";
let string2 = "Rajput";

let fullName = string1 + " " + string2;
console.log(fullName);
```

o/p ⇒ Meenakshi Rajput

e.g. ⇒

```
let string1 = "17"
let string2 = "18"

let newstring = +string1 + +string2;
console.log(newstring);
```

to convert string into number

o/p = 35

* template string :→

```
let age = 10;
let firstName = "Mishti";

let about = `My cousin's name is ${firstName}
            and she is ${age} year old.`
console.log(about);
```

placeholder

o/p → My cousin's name is Mishti and she is 10 years old.

→ **Undefined** - when we don't assign any value in variable after declaration. But we can't use const for this.

→ **null**

→ eg → let firstName ;
console.log (typeof firstName); o/p = Undefined

Note → We can print two or more values using single console.log

let firstName = "Meenakshi" ;

e.g. console.log (type of firstName, firstName);

o/p → ~~str~~ string Meenakshi

eg let myVariable = null ;
console.log (myVariable); // o/p = null

myVariable = "Meenakshi";
console.log (myVariable); // Meenakshi
console.log (typeof myVariable); // string.
console.log (typeof null);

↳ o/p = object
(bug) = error.

---

* BigInt → (2020)

let myNumber = 314;
Console.log (Number. MAX_SAFE_INTEGER);

e.g. let myNumber = BigInt (12); // 12n ;
console.log (myNumber);

Note → We can create BigInt by using keyword BigInt or using 'n' at last of the digits.

e.g. let Num1 = BigInt (156);
let Num2 = 156n;

* Operations of BigInt :→

console.log (Num1 + Num2);

o/p → 2312

Note → We can't use BigInt with another datatypes to perform operations.

---

* booleans & comparison operators →
true & false

e.g. let number1 = 5;
let number2 = 6;
console.log (number1 >= number2);

o/p = false

Note → comparison operators return true and false.

* == Vs ===

console.log (number1 == number2);

o/p = false

e.g. let num1 = 14 ;
let num2 = 14 ;
console.log (num1 == num2);
o/p = true
console.log (num1 === num2);
o/p = true

⇒ let num1 = "7";
let num2 = 7 ;
console.log (num1 == num2);
o/p = true.

Note → == only checks values. It doesn't check type of data.
=== checks both the values and datatypes of the data in variables.

⇒ Not equal to (!=) → let num1 = 7
let num2 = 7
console.log (num1 != num2) ;
o/p = false.

→ let num1 = 7
let num2 = "7";
console.log (num1 != num2) ; by checking // value
o/p = false
console.log (num1 !== num2);
o/p = true // by checking type & value both.

* Truthy and falsy values :→

↳ "abc"
1, -1
positive or -ve numbers

↳ false
""
null
undefined
0

It prints the else statements.

* if else condition :→

```
let age = 17;

if (age > 10) {
    console.log ("User can play ddlc");
} else {
    console.log ("User can play mario");
}
```

O/p = User can play mario.

e.g.

```
let num = 14;
if (num % 2 === 0) {
    console.log ("even")
} else {
    console.log ("odd");
}
```

o/p = even

e.g.

```
let firstName = " ";
if (firstName) {
    console.log (firstName);
} else {
    console.log ("firstName is kinds
                  empty");
}
```

o/p → firstName is kinde empty.

* Ternary Operators :→

```
let age = 15;
let drink;
if (age >= 5) {
    drink = "coffee";
} else {
    drink = "milk";
}
console.log (drink);
```

we can prefer ternary operator for this

(Conditional)

* Ternary operator Syntax → condition ? Expression1 : Expression2

e.g.

```
let age = 8;
let drink = age >= 5 ? "coffee" : "milk";
console.log(drink);
```

---

* (&) and - (||) Or operators :→

e.g.

```
let firstNam = "Meenakshi";
let favNum = 14;
if (firstName[0] === "M" && favNum >= 10){
    console.log("Name starts with M
                and favNum is greater
                than 10");
} else {
    console.log("Inside else");
}
```

Note → and (&) operator gives true value when both the inputs are true.

⇒ e.g.

```
if (firstName[0] === "m" || favNum<10){
console.log("Name and favourite Number");
} else {
    console.log("Inside else");
}
```

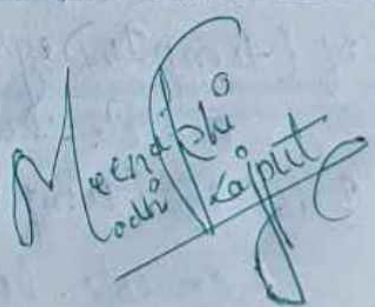
Note → OR (||) operator gives false value when both inputs are false.

* nested if else :→

```
let winning_number = 19;
let user_guess = +prompt ("Guess a number");
⇒ console.log (typeof user_guess, user_guess);
```

Note→ prompt takes input as string.
We use + to change string into number.

```
if (user_guess === winning_number) {
    console.log ("your guess is right");
} else {
    if (user_guess < winning_number ){
        console.log ("too low!!!");
    } else {
        console.log ("too high !!!");
    }
}
```



* if
else if
else if
else if
else

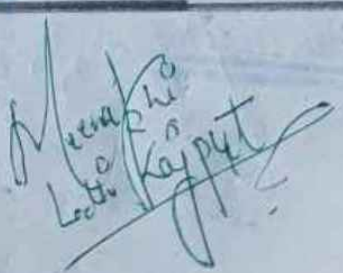
```
let tempInDegree = 35;

if (tempInDegree < 0){
    console.log("extremely cold outside");
} else if (tempInDegree <16) {
    console.log("It is cold outside");
} else if (tempInDegree <25){
    console.log("Lets go for swim");
} else if (tempInDegree <45){
    console.log("turn on AC");
} else {
    console.log("too hot!!");
}
```

→ We also can check conditions in reverse →

```
let tempInDegree = 37;
if (tempInDegree > 40){
    console.log("too hot !!!");
} else if (tempInDegree > 30) {
    console.log("lets go for swim");
} else if (tempInDegree > 20) {
    console.log("weather is cool");
} else if (tempInDegree >10) {
    console.log("extremely cold outside");
} else {
    console.log("extremely cold");
}
```

* Switch Statements →

```
let day = 2;
switch (day) {
    case 0:
        console.log ("Sunday");
        break;
    case 1:
        console.log ("Monday");
        break;
    case 2:
        console.log ("Tuesday");
        break;
    case 3:
        console.log ("Wednesday");
        break;
    case 4:
        console.log ("Thursday");
        break;
    case 5:
        console.log ("Friday");
        break;
    case 6:
        console.log ("Saturday");
        break;
    default:
        console.log ("Invalid day");
}
```



Loops

- for
- while
- do while
- for...in
- for...of

* while loop →

→ To print Natural numbers)

```
let i = 0;
while (i <= 10) {
    console.log(i);
    i++;
}
```

→ to print sum of natural numbers →

```
let ~~num~~ i = 0;
let total = 0;
while (i <= 100) {
    total = total + i;
    i++;
}
console.log(total);
```

(or) By using - $(n * (n+1))/2$

```
let total = ~~(num(nv~~ (n * (n+1))/2;
console.log(total);
```

* for loop →

```
for (let i = 0; i <= 9; i++) {
    console.log(i);
}
```

→ To print sum of 10 natural number by using for loop →

```
let total = 0;
let num = 10;
for (let i=1; i<=num; i++) {
    total = total + i;
}
console.log (total);
```

* break keyword

It ~~exe~~ stops the execution

```
for (let i= 1; i <=10; i++){
    if (i===4){
        break;
    }
    console.log(i);
}
```

o/p → 1
2
3

continue keyword

It skip the statement.

```
for (let i=1; i<=10; i++){
    if (i===4) {
        continue;
    console.log(i);
}
```

o/p → 1
2
3
5
6
7
8
9
10

* do while loop →

```
let i = 0;
do {
    console.log(i);
    i++;
} while (i <= 9);
```

Meenakshi Lodhi Rajput

ARRAY

= object

type = object

↳ Reference type

(ordered collection of items)

→ how to create array →

```
let fruits = ["apple", "mango", "grapes"];
let numbers = [1, 2, 3, 4]
let mixed = [1, 2, 2.3, "string", null,
             undefined"];

console.log(mixed);
console.log(numbers);
console.log(fruit[2]);
```

index number.

→ ARRAY INDEXING →

e.g. array is ~~immutable~~ —

```
let fruits = ["apple", "mango", "grapes"];
fruits[1] = "banana";
conside.log(fruits);
```

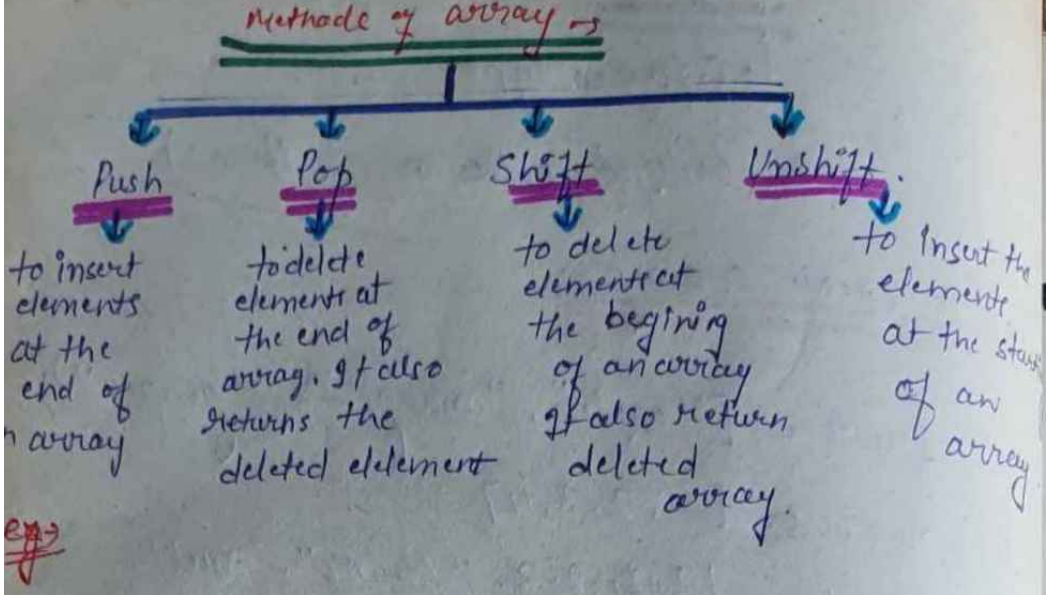
o/p ⇒ ["apple", "banana", "grapes"]

Note → type of array is object.

```
console.log(typeof fruits);   // object.
console.log(Array.isArray(fruit));
```

↳ fun^n that tells the type of reference types.
(method)

Note let obj = {} // object literal.



eg →

```
let fruits = ["apple", "mango", "grapes"];
console.log(fruits);
```

⇒ push :→

```
fruits.push("banana");
console.log(fruits);
```

⇒ pop :→

```
let poped fruit = fruits.pop();
console.log(fruits);
console.log("popped fruits is", poppedfruit);
```

⇒ unshift :→

```
fruits.unshift("banana");
fruits.unshift(
console.log(fruits);
```

⇒ shift :→

```
let removedFruit = fruits.shift();
console.log(fruits);
console.log("removed fruit is", removedFr
```

Note→ push and pop are fast as compare to shift and unshift.

* Primitive Vs Reference data type :→

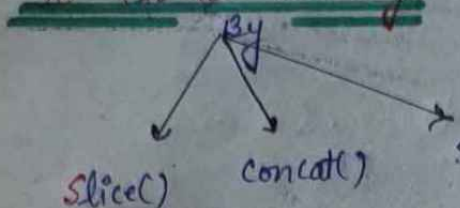
```
let num1 = 6;
let num2 = num1;
console.log("value of num1 is", num1);
console.log("value of num2 is", num2);
num1++;
console.log("after incrementing num1");
console.log("value of num1 is", num1);
console.log("value of num2 is", num2);
```

→ Reference type → e.g. array.

```
let array1 = ["item1, item2"];
let array2 = array1;
console.log("array1", array1);
console.log("array2", array2);
array1.push("item3");
console.log("after pushing element to array1");
console.log("array1", array1);
console.log("array2", array2);
```

Note → primitive data type store in stack.
→ reference data type reuse memory.

* To clone array →



e.g. let array1 = ["item1", "item2"];

let array2 = array1.slice(0);

(or) let array2 = [].concat(array1);

(or) let array2 = [...array1];

Note → To clone and concate items in an array :→

let array3 = array1.slice(0).concat(["item3", "item4"]);

(or) let array3 = [].concat(array1, ["item3", "item4"]);

(or) let array3 = [...array1, "item3", "item4"];

(or) let onMoreArray = ["item5", "item6"];

let array4 = [...array1, ...onMoreArray];

* for loop in Array :→

```
let fruits = ["apple", "mango", "grapes"];
for (let i = 0; i <= fruits.length-1; i++){
    console.log(i); (fruits[i]);
}
```

Note → we should use let for creating array.

→ use const for creating array :→

```
const fruits = ["apple", "mango"];
fruits.push("banana");
console.log(fruits);
```

o/p → ["apple", "mango", "banana"];

* while loop in Array :→

```
const fruit = ["apple", "mango", "grapes"];
let i = 0;
while (i < fruits.length) {
    console.log(fruits[i]);
    i++;
}
console.log(fruits);
```

Note → we can't change const array by using square brackets, but we can apply methods on const array (i.e. push, pop etc.)

* for..of loop in array :→

```
const fruits = ["apple", "mango", "grapes"];
const fruit = [];
for (let fruit of fruits) {
fruits2.push (fruit.toUpperCase());
}
console.log (fruits2);
```

* for...in loop in Array :→ It gives index.

```
const fruits = ["apple", "mango", "orange"];
for (let index in fruits) {
        console.log (fruits[index]);
console.log (index);
}
```

* array destructuring :→

```
const myArray = ["value1", "value2"];
let [myVar1, myVar2] = myArray;
console.log ["value of myVar1 ", myVar1];
console.log ["value of myVar2 ", myVar2];
```

* Note → We can also skip index.

⇒ Rest operator →

```
let [myVar1, myVar2, ...myNewArray]
                    = myArray
```

* OBJECTS → (Reference type.)

→ arrays are good but not sufficient for real world data.
→ objects store key-value pairs.
→ objects don't have index.

⇒ how to create objects :→ (object literals.)

```
const person = {
    name : "Meenakshi",
    favNum : 14,
    hobbies : ["yoga", "dance", "craft-making", "poetry"]
}
console.log(person);
```

(Keys (or) properties → name, favNum, hobbies; values)

⇒ how to access data from objects :→

```
console.log(person.name); // dot notati
console.log(person.hobbies);
```

⇒ To add key-value pair in object :→

```
person.gender = "female";
```

(or)

```
person["gender"] = "female";
```

Note: bracket notation :→

```
console.log(person["name"]);
console.log(person["favNum"]);
```

* difference between dot and bracket notations :→

```
const key = "email";
const person = {
    name : "Meenakshi",
    favNum : 14
    "person hobbies" : ["gardening",
                        "teaching", "learning"];
}

person[key] = "rajputmeenakshixyz@gmail.com";
    console.log(person);
```

⇒ How to iterate object →

for in loop object.keys

eg:

```
for (let key in person) {
    console.log(person[key]);
}
```

→ for key-value pair :—

```
for (let key in person) {
    console.log(`${key} : ${person[key]}`);
(or) console.log(key, ":", person[key]);
}
```

backtick

⇒ console.log (typeof (Object.keys(person)));

const ~~Consoltr~~ val = Array.isArray ((Object.keys (person)));
console.log (val);

---

* Computed properties :⇒

const key1 = "objkey1";
const key2 = "objkey2";

const value1 = "myvalue1";
const value2 = "myvalue2";

⇒ const obj = {
~~objkey1~~ :
[key1] : value1,
[key2] : value2
};

(or) const obj = {};
obj[key1] = value1;
obj[key2] = value2;
console.log (obj);

O/P → const obj = {
objKey1 : "myvalue1",
objKey2 : "myvalue2"
}

* Spread operator in objects :→

```
const obj1 = {
    key1 : "value1",
    key2 : "value 2"
};
const obj2 = {
    key1 = "valueUnique",
    key3 = "value3",
    key4 = "value4"
};
const newObject = {...obj1, ...obj2, key69: "value
const strObject = {..."abc"};
console.log(newObject);
console.log(strobject);
```

Note → integers (numbers) are not iterable.

* Object destructuring :→

```
const band = {
    bandName : "Led zeppelin",
    famousSong : "Stairway to heaven",
    year : 1968,
    anotherfamousSong : "Kashmir",
};
Let {bandName, famousSong, ...restProps} =
    console.log(bandName);
    console.log(restProps);
```

* objects inside array :→

```
Const users = [
  {userId: 1, firstName: 'Meenakshi', gender: 'female'},
  {userId: 2, firstName: 'Mishti', gender: 'female'},
  {userId: 3, firstName: 'Kunj', gender: 'male'},
]
```

loop→

```
for(let user of users){
    console.log(user.firstName);
}
```

o/p→ Meenakshi
Mishti
Kunj

* nested destructuring :→

→
```
const [user1, user2, user3] = users;
console.log(user1);
```

→
```
const [{firstName}, , {gender}] = users;
console.log(firstName);
console.log(gender);
```

→
```
const [{firstName: user1firstName, userId, , {gender:
                                                user3gender}}];
```

to change variable name

```
console.log(user1firstName);
console.log(user3gender);
```

FUNCTIONS :→

keyword — func^n name — body of fun^n

```
function singHappyBirthday() {
    console.log("happy birthday to you .... ");
}
```

parameters

```
function sum3Nums(num1, num2, num3){
    return num1 + num2 + num3;
}
```

argument

```
const returnedvalue = sum3Nums(2, 3, 5);
console.log(returnedvalue);
```

function calling / invoke →

```
Sum3Nums(5, 6, 7);
SingHappyBirthday();
```

Examples →

→
```
function isEven(number) {
    return number % 2 === 0;
}
console.log(isEven(4));
```

→
```
function firstChar(anystring) {
    return anystring[0];
}
console.log(firstChar("abc"));
```

```
function findTarget (array, target ){
    for( let i = 0;  i < array.length;  i++){
     if (array[i] === target ){
            return i ;
        }
    }
    return -1;
}
const myArray = [1, 3, 8, 90]
 const ans = findTarget (myArray, 4);
    console. log(ans);
```

* Function Expression :-> when we assign a function in a variable.

e.g. anonymous function.

```
const singHBday = function() {
    console. log ("happy birthday");
}
```

e.g.

```
const SumNums = function (num1 , num2 , num3);
    { console. log (num1 + num2 + num3 );
    }
```

dry = don't repeat yourself

* Arrow Function :→

eg→
```
Const singHBdy = () ⇒ {
    console.log ("Happy Birthday");
}
```

e.g→
```
Const sum3Nums = (num1, num2, num3) ⇒ {
    return num1 + num2 + num3;
}
Const ans = sum3Nums (2, 3, 4);
    Console. log (ans);
}
```

Note → we can remove parenthesis () if function takes one parameter.

eg:
```
Const isEven = number ⇒ {
    return number % 2 === 0;
}
console. log (isEven(3));
```

o/p ⇒ false.

⇒ we also can write :→
```
Const isEven = number ⇒ number % 2 ===
    console. log (isEven(6));
```

o/p = true

* hoisting :→ we can call function befor declaration. But it is not posible in function expression or in arrow function.

e.g.

```
hello();

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

Note→ We can access variable using var keyword before declaration, but output will be undefined.

→ In let and const it will not be possible.

e.g.

```
console.log(hello);
var hello = "hello world";
```

o/p → undefined

* Function inside function :→

```
function appC1 {
    const myFunc = () => {
    console.log ("hello from myfunc");
    }
const  add2 = (num1, num2) => {
        return num1+num2;
    }
const mult = (num1 , num2) => num1 * num2
    Console . log ("Inside app");
    myFunc ();
    console. log (add2 (2,3));
    Consol. log (mult (2,3));
}
app ();
```

* Lexical Scope :→

```
const myVar = "value1";
function myAppC) {
    function myfunc() {
    const myFunc2 = ()=> {
    console. log ("inside myFunc", myVar);
    } myFunc2();
    } console. log (myVar);
        myFunc();
}
```

* block scope Vs function scope :→

let & const are block scope

var is function scope.

Note→ we can access var globally (outside the block). But we can't access let and const outside the block.

e.g.

```
function myApp() {
    if (true) {
        var/const/let function firstName = "Meenakshi";
        console.log(firstName);
    }
    console.log(firstName);
}
```

var ✓ const ✗ let ✗

block

* default parameters :→

e.g.

```
function addTwo(a, b = 0) {
    return a+b;
}
const ans = addTwo(4);
console.log(ans);
```

* rest parameters :→

```
function addAll(...numbers) {
    let total = 0;
    for (let number of numbers) {
        total = total + number;
    } return total;
}
const ans = addAll(4,5,4,2,10);
console.log(ans);
```

* **parameter destructuring :→** It is used with objects.

```
const person = {
    firstName = "Meenakshi";
    profession = "Trainee";
    Bday = 14
}

function printDetails({firstName, profession, Bd
    {
        console.log (firstName);
        console.log (profession);
        console.log (Bday);
    }
```

* **Callback functions :→**

→ we can pass function in a function.

e.g.

```
function myfunc2 (name) {
    console.log ("Inside my func 2");
    console.log ("your name is ${name}")
}
function myFunc (callback) {
    const data = {}
    console.log ("hello there I am a
        func and I can--");
    callback ("Meenakshi");
}
myFunc (myFunc2);
```

* function returning function :→

eg:

```
function myFunc() {
    function hello(){
     return "hello world";
    }
    return hello;
}
const ans = myFunc();
console.log(ans());
```

* Important array methods :→

→ forEach
→ map
→ filter
→ reduce.

1.) **forEach** :→ It takes callback func" as an input.

```
const number = [4,2,5,8];
number.forEach( function(number,index) {
    console.log(number * 3, index);
})
```

2.) **map method** :→ It always create new array.

```
Const number = [3,4,6,1,10]
const squareNum = number.map((number,
                              index)=>{
  return `index : ${index}, ${number * number}`
});
  console.log(squarNum);
```

3.) filter method :→ It always returns true or fals

```
Const numbers = [1, 6, 9, 0, 3, 4]
Const evenNumbers = numbers.filter((number)
                                  ⇒ {
    return number % 2 === 0;
});
console.log(evenNumbers);
```

4.) reduce method :→

e.g. To add all numbers :→

```
Const numbers = [3, 7, 6, 5, 4, 10]

Const sum = numbers.reduce((accumulator,
                            currentValue) ⇒ {
    return accumulator + currentvalue;
});
```

* Sort method :→ It changes the array.

e.g. Const numbers = [5, 9, 1200, 410, 3000];
numbers.sort((a,b) => {
return a-b;
});

Note → T.S. sorts the number as strings by coverting in ASCII values.
→ Capital letters will be sort first.

* find method :→

e.g. const myArray = ["Hello", "cat", "dog", "Lion"];
Const ans = myArray.find((string) => string.length===3);
console.log(ans);

* every method :→ It also returns boolean values.

Const number = [2,4,6,8,10];
Const ans = number.every((number) => number % 2===0);
console.log(ans);

* Some method :→

e.g.
```
const numbers = [3, 5, 11, 9];
const ans = numbers.some((number) => number%2==);
console.log(ans);
```

* Fill method :→ It changes the original array.

e.g.
```
const myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9];
myArray.fill(0, 2, 5);
console.log(myArray);
```

* Splice method :→ It is used to insert and delete ~~between~~ in array

e.g.

To delete :→
```
const myArray = ['item1', 'item2', 'item3'];
myArray.splice(1, 1);
// console.log(myArray)
console.log(myArray);
```

To insert :→
```
myArray.splice(1, 0, 'Inserted item');
console.log(myArray);
```

Note → It also return deleted items. we can store this item in another variable.

* **iterables :→**

string, array are iterables because we can apply loop on it.

e.g.
```
const firstName = "Meenakshi";
for (let char of firstName) {
    console.log(char);
}
```

(or)
```
const items = ['item1', 'item2', 'item3'];
for (let item of items) {
    console.log(item);
}
```

---

* **array like object :→**

→ string is an array like object.

e.g.
```
const firstName = "Meenakshi";
console.log(firstName.length); //o/p → 8
Console.log(firstName[2]);     //o/p = e
```

---

* **sets :→** It is iterable to store data. sets also have its own methods. We can't store duplicate items in sets. We can't access sets index. It is not ordered.

e.g. **empty set :-**
```
const numbers = new Set();
numbers.add(1);
numbers.add(items);
```

Note → we can use if condition and loop in sets.

⇒ We can extract set from array :→

```
const myArray = [1, 2, 4, 5, 6, 9];
const uniqueElements = new Set(myArray);
```

* Map object :→ It is also iterable i.e. we can use loop in Map object. It stores data in ordered and in key-value pair like object. Duplicate keys are not allowed like objects. In Map, we can use anything (array, number, string etc.). Mostly string or symbol are used as key in Map object.

eg.

```
const person = new Map();
person.set('firstName', 'Meenakshi');
person.set(1, 'one');
console.log(person.keys());
console.log(person.get(1));
```

~~* close using~~

* Optional chaining :→ It is used in nested objects.

eg.

```
let user = {
  firstName : "Meenakshi",
  address : { houseNo: '153' }
}
console.log(user?.firstName);
console.log(user?.address?.houseNo);
```

Meenakshi Lodhi Rajput

* methods :→ function inside object is called method.

e.g.

```
function personInfo() {
    console.log('person name is ${this.firstName} and Id is ${this.Id}');
}

const person1 = {
    firstName : "Meenakshi",
        Id    : 101,
    about     : personInfo
}

const person2 = {
    firstName : "Mishti",
        Id    : 102,
    about     : personInfo
}

const person3 = {
    firstName : "Tanvi",
        Id    : 103,
    about     : personInfo
}

person1.about();
person2.about();
person3.about();
```

object

* Console.log(this) console.log(window)

⇒ We can call fn by using .call() :→

e.g.

```
function hello() {
    console.log ("hello world");
}

hello.call();          // function calling.
```

* apply() :→

e.g.

```
function about (hobby, favMusician) {
console.log (this.firstName, this.age, hobby,
                                favMusician);
}
const user1 = {
    firstName : "Meenakshi",
        age :  23,
    }
    const user2 = {
        firstName : "Itakshi",
            age   : 24,
        }

about.apply(user1, ["Yoga", "Poetry"])
```

* bind() :→

```
const func = about.bind (User2, "Yoga", "Poetry");
func();
```

* new keyword :→

```
function createUser (firstName, age) {
    this.firstName = firstName;
    this.age = age;
}
CreateUser prototype.about = function () {
    console.log (this.firstName, thise age);
}
const user1 = new createUser ("Meenakshi", 23);
    User1.about();
```

Functions Up TRAINEE :→