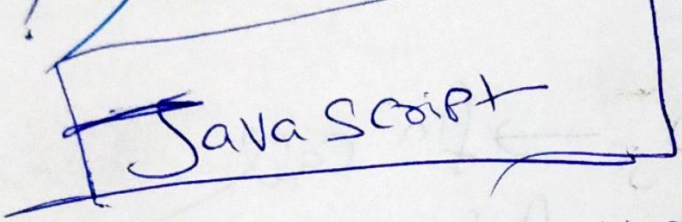


Building Blocks of JS — Variables, datatype, statements, functions.

Variables. — Identifiers used to store data and info related to application. Variables are defined using following keywords: —

Let: — The variables declared with let have block scope. Variables declared with let must be declared before use.

→ A block is defined as a set of lines of code enclosed within a pair of Parentheses {}. It can be a for loop, if stmt, function etc.

Variables declared ~~within~~ inside a block {} cant be accessed from outside the block:

```
{ let x=2;
}
```

Global scope: — Variables declared with var have Global scope.

```
{
  var x = 2;
}
```

here x can be accessed outside block.

Note:- Variables defined with let cannot be redeclared.

```
let x = "Smith";
let x = 0;  // we cant do this.
```

- Variables defined with var can be redeclared.

```
var x = 10;
{
  var x = 2;  allowed.
}
```

```
<!doctype html>
<html>
<body>
<h2> Redeclaring Variable using Var </h2>
<p id = "demo" > </p>
<script>
var x = 10;
{
var x = 2;
}
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

o/p — Redeclaring a Variable using var.

2

Eg

```
var x = 2;  // allowed
let x = 3;  // not allowed

{
  let x = 2;  // allowed
  let x = 3;  // not allowed.
}
```

```
{
  let x = 2;  // allowed
  var x = 3;  // not allowed
}
```

Note:— Redeclaring a variable with let in another block is allowed.

```
let x = 2;  // allowed
{
  let x = 3;  // allowed
}
{
  let x = 4;  // allowed.
}
```

Const:— This is used when containing value is constant and will not undergo change.

```
Const Pi = 3.14;
```

Note:— A variable declared with const keyword cannot be reassigned.

```
Const Pi = 3.141;
Pi = 3.12;  // error.
Pi = Pi + 10;  // error.
```

Scope:— Scope in JS tells us which variables will be accessible at a given point. 2 types: global, local.

Global:— Any variable declared aside any function.

is accessible anywhere in code even in the functions and is in global scope.

```
const global = "Hi ! I am Gopal";
```

Local scope :- Accessible only in a specific part of code.

2 types — function scope
— Block scope

function scope — When a variable is declared within a function, it is accessible only within the function. U cannot access the variable once u are out of fn.

```
function HelloWorld () {
const hello = "welcome to JS"; // this is fn scoped.

Console.log (hello);

}
HelloWorld ();

Console.log (hello ); // gives error as hello is not defined.
```

Block scope — When a variable is declared using const or let keyword, within a block of curly brace, it is accessible only within that curly brace.

```
{
  Const hello = "welcome"; //Block scope.
  Console.log(hello);
}
console.log(hello); // error as hello is not defined.
```

Note:- multiple variables can be declared in a single statement.

```
var x=1, y=2, z=3;
```

Note:- 1) Variables should have purposeful names.
2) It can contain $ and _ but not hyphen(-).
3) use Camelcase. firstName, lastName.

Pg:-92 Diff. let, Const, var.

Datatypes

- Primitive types
  - Number
  - String
  - Boolean
  - null
  - undefined
- Composite types.
  - Array
  - objects.

Number:- A primitive datatype to handle any type of no:- integer or float.

We can Perform mathematical operations: — +, −, *, /, %

```
let Pi = 3.14;
let infiniteNumber = 12/0;
const notANum = 'here'/2;
```

gt will return NaN (not a no), an invalid expression.

String:— one or more characters enclosed within quotes.

Both single and double quotes can be used.

```
let name = "JavaScript";
let fullName = "JS" + "Rocks";
let fullName = 'JS' + 'Rocks';
```

• Boolean :— A primitive datatype that takes only 2 values.

true / false. ← small

```
let isCarFlag = true;
let BigNum = num1 > 1000;
```

null → for unknown value or empty value.

```
let yearOfManufacture = null;
let num1;
Console.log(num1); //undefined.
```

Object — Complex data for handling complex data structures, including data of different types.

```
Var person = { name: "peter",
               age: 24,
               designation: "Software Engineer",
               Allocated: true }
```

Arrays — This is a subtype of an object to handle list of items of same primitive type.

```
Var cities = ["Hyderabad", "Newyork", "BBSR"]
```

JS symbols — It is new to JS and a new type of primitive datatype introduced as part of ECMA Script 2015.

- It represents unique & immutable primitive value that can be used as an identifier.

type of operator :- It can be applied to any value and it returns a string indicating the datatype of the value.

```
Console.log (typeof 19); // number
Console.log (typeof "javascript"); // string
Console.log (typeof true); // boolean
Console.log (typeof undefined); // undefined
```

JS Datatype Conversion:

- It is done by following ways:—
  - JS function.
  - Automatically by JS :: - It is done when a mismatch is found.

Some Common type Conversion using function:—

To Convert to a string :— Any variable can be converted to a string using following options.

- String() → global method to convert nos, booleans, literals, expressions, dates to strings.

```
String(false) //returns false
String(Date()) → returns current date in string format
```

- ToString() → This method does conversion on any no & converts to string.

```
(498).toString() //"498"
```

- Number() → converts a variable or value into a no.

```
Number("3.14").   3.14
Number("  ")      o/p — 0
Number("99 88").  NaN
Number("Sun")     NaN
```

Number(false) //returns 0

Number(true) //returns 1

Operators :- operate on 1 or more operands to give a result. These are applied to variables to perform value manipulations. Based on no. of operands they work upon

Unary, binary, ternary :-

```
2++ ; // Unary operator.
2+3; // Binary operator.
(4>3) ? true : false ; ternary operator.
```

Types of operators

Arithmetic operators — +, -, *, /, %, ++, --

```
let num1 = 6 ;
let num2 = 4 ;
num1 * 4 ;
num1 / num2 ;
let rem = num1 % num2; // modulus.
num1++ ; // post-increment
++num1 ; // pre-increment
num1-- ; // post-decrement
--num1 ; // pre-decrement
let x = 100 + 50 ;
document.write(x) ;
```

```
let x = 5 ;
let y = 2 ;
let z = x + y ;
document.write(z) ;
```

<script>

```
let n=5;
let z = n++;
document.write(z);
document.write(n);
let y = n++ + ++n;
document.write(y);
```

O/P — 5, 6, 13

Note — ++ increment (++) operator is used to increase the value of variable by 1.

If the operator is used before the operand, the value is increased by 1 and then returned but if the operator is after the operand then the value is 1st returned & then incremented.

```
let n=10;
Console.log(n++);
Console.log(n);
```

O/P — 10, 11

```
let n=10;
Console.log(++n);
Console.log(n);
```

11, 11

Assignment operators —

```
var num = 4;
num += 2;   // num = num + 2;
num -= 1;   // num = num - 1;
num *= 5;   // num = num * 5;
num /= 2;   // num = num / 2;
num %= 2;   // num = num % 2;
```

Addition assignment

Subtraction assignment
</script>

Comparison operators

5 === "5" // strict equality, false.
5 == "5" // Equality, true.
5 !== "5" // strict - non equality, true.
5 != "5" // non-equality, false.

Strict equality (===) : It checks whether its 2 operands are equal returning a boolean result. It considers operands of different types to be different.

Console.log (1 === 1); true.
Console.log ('hello' === 'hello'); true.
Console.log ('1' === 1); false.
Console.log (0 === false); o/p - false.

Strict Inequality (!==) : It checks whether its 2 operands are not equal returning a Boolean result. It considers operands of different types to be different.

Console.log (1 !== 1); o/p - false
Console.log ('hello' !== 'hello'); false
Console.log ('1' !== 1); true.
Console.log (0 !== false); true.

<, <=, >, >=, !=, =

Logical operators — AND, OR, NOT

```
var x=4;
var y=2;
(x<10 && y>0);      true.
(x==5 || y==5);     false.
!(x==y);            true.
```

<55/>

Type Coercion: — process of automatic conversion of values from one datatype to another.

Eg— no to string, string to no, boolean to no.

HTML element → is defined by a start tag, some content & an end tag.

<tagname> content </tagname>

→ it refers to everything from start tag to end tag.

Block level elements → always starts on a new line & automatically browser adds some space before & after the element

Inline element → does not start on a new line.

- An inline element only takes up as much width as necessary.

Eg— span,

Preformatted text:— <Pre> tag defines preformatted text. The element is displayed in a fixed width font and the text preserves both spaces and linebreaks.

```
<body>
<pre>
gud
morning to u
hiii
</pre>
</body>
```

o/p—

gud
morning to u
hiii

Eg—2

```
<Pre>
gud
morning
</Pre>
```

o/p—

gud
morning
to

<ins> tag :— It defines a text that has been inserted into a document. Browsers will usually underline inserted text.

```
<p> my fav Color <del> blue </del>
<ins> red </ins> !</p>
```

my fav Color ~~blue~~ red !

<mark> tag defines text that should be marked or highlighted.

<P> Good <mark> morning </mark> today </P>

Good morning today

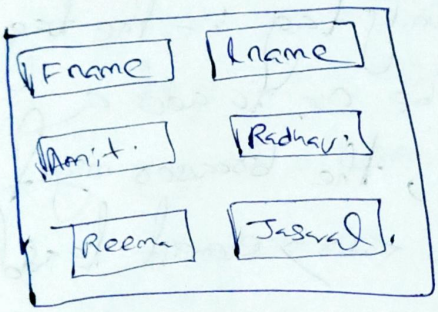
highlighted text

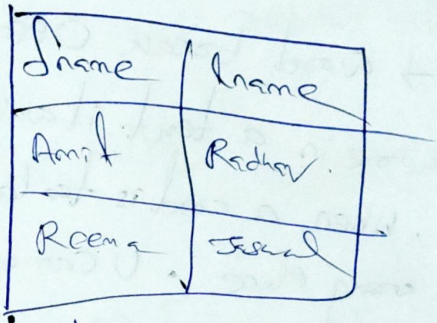
<UL start = N> •

<OL start = N> 1. . . .

---

border:





border-collapse: separate

border-collapse: collapse

- border
- border-collapse
- padding
- border-spacing

<head>
<style>

table, td, th { border: 1px solid black; }

#table1 {
border-collapse: separate; }

#table2 { border-collapse: collapse; }
</style> </head>

<body>
<table id = "table1">
- - - -

</table>
<table id = "table2">
</table>

<wbr> → word break Opportunity tag :- the tag specifies where in a text it would be ok to add a line break. When a word is too long, the browser might break it at wrong place. U can use <wbr> element to add word break opportunity

<html>
<body>
<h1> A v. good mng </h1>
<h2> <wbr> hour U </h2>
</body>

Comments :— Single line Comment //
multiline comment :— /* */

Statements — made up of Variables, Value, operators, Expressions, keywords, Comments.

Variable declaration

let num1, num2;

Assignment — let sum = num1 + num2;

Return statements: — return sum;

Break/Continue — Return stmt: — it stops execution of debugger.

a function & returns a value

Note: — Break stmt used to jump out of a loop

```
<!Doctype html>
<html>
<body>
<h1> — </h1>
<p> — <p>
<p id="demo"> </p>
<script>
function myFunction() {
    return Math.PI;
}
```

return value ;

break :- break stmt helps to jump out of a loop even it is encountered.

```
<script>                <p id="demo"></p>
let text = " ";
for (let i=0; i<10; i++) {
if (i===3) { break; }
text += "no is" + i + "<br>";
```

o/p — 0
1
2

document.getElementById("demo").innerHTML = text;

</script>

Continue:- statement breaks one iteration in the loop if a specified condition occurs, and continues with the next iteration in the loop.

```
<p id="demo"></p>
<script>
let text = "";
for (let i = 0; i < 10; i++)
{
if (i === 5) { continue; }
text += "no is" + i + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>
```

o/p —

0
1
2
3
4
6
7
8
9

if...else.

<script>

const hour = new Date().getHours();

let greeting;

if (hour < 18) {

greeting = "Good day";

}

else {

greeting = "Good evening";

}

document.getElementById("demo").innerHTML =

greeting;

</script>

JS Switch stmt. — use it to select one of many codeblocks to be executed.

```
switch (expression) {
  case x:
    // codeblock
    break;
  case y:
    // codeblock
    break;
  default:
    // codeblock
}
```

working. —

→ Switch expression is evaluated once.

- Value of the expression is compared with value of every case.
- If there is a match the concerned block of code is executed.

- If no match, the default code block is executed.

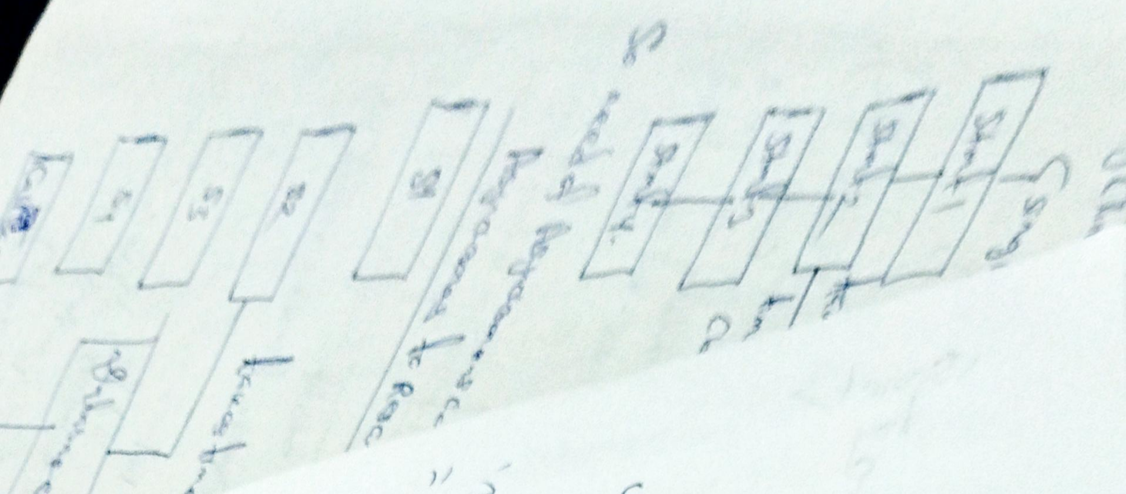
for loop: loops through a block of code a no. of times

```
<p id = "myid"></p>
<script>
const colors = ["Red", "violet", "purple"];
let text = "";
for (let i = 0; i < colors.length; i++)
{
text += colors[i] + "<br>";
}
document.getElementById("myid").
innerHTML = text;
</script>
```

for/in loop — loops through the properties of an object

```
for (key in object) {
// code block }
```

```
const person = {fname: "John", lname:
"Doe"
age: 35 };
```



```
let text = " ";
for (let x in person) {
    text += person[x];
}
```

Try catch statement :- It is used to handle any error in try block code.

try --- catch construct has 2 main blocks :- try & catch.

```
try {
    //code            ──────→   throw {
}                                 error : "Throwing error"
catch (error)                   }}
{
    //error handling logic....
}
```

— try statement

- try statement defines the code block to run or try.
- throw statement defines a custom error.
- catch stmt defines a code block to handle any error.
- finally stmt. defines a code block to run regardless of result of try block
- catch & finally are optional, but u must use one of them

```
&lt;body&gt;
    &lt;p id="demo"&gt; &lt;/p&gt;
```

```
<script>
try
{
addalert ("welcome guest");
}
Catch (err) {
document.getElementById ("demo").innerHTML
err.message
}
</script>
</body>
</html>
```

<p> enter a no beh.

JS functions :—

A JS function is a block of code designed to perform a particular task.

```
function myFunction (P1, P2)
{
return P1 * P2;
}
```

→ A JS function is defined with keyword function, followed by name & ().

→ Function names can accept letters, digits, underscores & dollar sign (same rules as variables)

→ The function accepts parameters called by JS objects.

→ If return stmt is not explicitly required, then undefined is returned as value.

```
var startPos = { xPos: 20, yPos: 20 };
var currentPos = { xPos: 40, yPos: 40 };
currentPos.xPos, currentPos.yPos
startPos.xPos  startPos.yPos
function calculateDistanceTravelled (startPos, currentPos)
{
var distance =
Math.sqrt ( Math.pow (currentPos.xPos - startPos.xPos, 2)
+ Math.pow (currentPos.yPos - startPos.yPos, 2))
return Math.ceil (distance);
}
document.write ("you have travelled " +
calculateDistanceTravelled (startPos, currentPos)
+ " km");
```

- Function as expression —

```
var square = function (x) { return x * x; };
function calculateDistanceTravelled (startPos, currentPos)
{
var distance = Math.sqrt ( square (currentPos.xPos -
startPos.xPos) + square (currentPos.yPos -
startPos.yPos));
```

return Math.ceil(distance);

var startPos = {xpos: 20, ypos: 20};

var currentPos = {xpos: 40, ypos: 40};

d.w("Traveled distance is " + calculateDistanceTraveled(startPos, currentPos) + "km");

Note:- What is the difference betn function declaration vs. function expression?

Different Aspects of function:-

Parameters:- Values defined as part of function declaration.

Argument:- Values that we use at the time of calling the function.

Note:- We can pass 0 or more parameters to the function.

Nested Scopes:- When a function is defined inside another function, the inner function can access the variable of the outer function. This is called lexical scoping & the inner function is called a closure.

Array:- List of values represented with an index starting from 0.

```
let city0 = "Hyderabad";
let city1 = "Newyork";
let city2 = "Delhi";
let cities = ["Hyderabad", "Newyork", "Delhi"];
```

console.log (cities [0]);

console.log (cities [1]);

let name = [ ]; empty array.

let names = ['JS', 'HTML', 'CSS'];

Syntax

Const array_name = [item1, item2, ...];

A declaration can span multiple lines:

```
Const cars = [
    "BMW",
    "Volvo",
    "Mazda"
    ];
```

U can also create the array & then provide the elements.

```
Const cars = [ ];
cars[0] = "BMW";
cars[1] = "Volvo";
cars[2] = "Mazda";
```

Access ~~Const Cars~~ let car = cars[0];

Changing value of element cars[0] = "myCar";

Const len = cars.length;

Using keyword new:

```
<p id="demo"></p>
<script>
Const cars = new Array ("Saab", "Volvo", "BMW");
document.getElementById("demo").innerHTML = cars;
</script>
```

Converting an Array to a string:

```
<script>
const cities = ["Hyderabad", "Pune", "Bombay"];
document.getElementById("demo").innerHTML =
cities.toString();
</script>
```

```
<p id="demo"></p>
<script>
const person = ["Ajay", "Binoy", 64];
document.getElementById("demo").innerHTML =
person[0];
</script>
```

Array as objects.

```
<p id="demo"></p>
<script>
const person = {firstname: "John", lastname: "Doe",
age: 51};
document.getElementById("demo").innerHTML =
Person.firstname;
</script>
```

Accessing last array element

~~let person~~

```
const fruits = ["mango", "orange", "grapes"];
let f = fruits[fruits.length - 1];
```

Looping Array elements –

```
<p id="demo"></p>
<script>
const fruits = ["mango", "orange", "grapes"];
let flen = fruits.length;
let text = "<ul>";
for (let i=0; i<flen; i++) {
text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";
document.getElementById("demo").innerHTML = text;
</script>
```

o/p – mango
orange
grapes.

for each –

```
const fruits = [ . . . ];
let text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";
```

```
function myFunction(value)
{
text += "<li>" +
value + "</li>";
}
```

Adding New Elements to Array

```
num = [1,2,3,4,5];
n = num.push(6,7,8);
Console.log(num); //[1,2,3,4,5,6,7,8]
Console.log(n); // 8
```

Take an Array : Apple, orange, mango.
Create button named Tryme.
If u click Tryme, banana to be added.

```
<button onclick="myfunction()">Tryme</button>
<p id="demo"></p>
<script>
  Const fruits = ["Apple", "orange", "mango"];
document.getElementById("demo").innerHTML = fruits;
  function myfunction(){
      fruits.Push("Banana");
document.getElementById("demo").innerHTML =
                                    fruits;
}</script>
```

or

```
fruits[fruits.length] = "lemon";
```

```
const fruits = ["mango", "Banana", "orange"];
fruits[6] = "lemon";
```

O/P — mango
Banana
orange
undefined
undefined
undefined
lemon.

Joining Arrays using Concat

```
arr1 = [1,2,3];
arr2 = [4,5];
arr3 = [6,7,8];
arr = arr1.concat(arr2, arr3);
Console.log(arr);
```

Removing elements from array —

using Pop —

```
num = [1,3,3,4,5];
n = num.pop();
Console.log(num); //[1,2,3,4]
Console.log(n); //5
```

Removing first element using shift

```
num = [1,2,3,4,5];
n = num.shift(a);
Console.log(num);
//[2,3,4,5]
Console.log(n); //1.
```

Slice method —

Create a new array using elements from existing array. The original array is not impacted with this.

```
num$ = [1,2,3,4,5];
arr1 = num.slice();
Console.log(arr1);
```

O/P — [1,2,3,4,5];

Ex-2

num = [1, 2, 3, 4, 5];
0 1 2 3 4

array2 = num.slice(1, 3);

Console.log(array2);

O/P — [2, 3]

Ex-3

array3 = num.slice(2);

Console.log(array3);

O/P — [3, 4, 5]

Const fruits = ["Banana", "orange", "Lemon", "Apple", "mango"];
0 1 2 3 4

Const citrus = fruit.slice(1, 3);

O/P — Orange lemon.

Const n = fruits.slice(-1, -3);

O/P — Lemon, apple

Splice

Const numbers = [1, 2, 3, 4, 5];
2nd

Lets remove last 3 elements and restore.

numbers.splice(2, 3);

Console.log(numbers);

Const deleted = numbers.splice(2, 3);

Console.log(deleted);

O/P — 3, 4, 5

Const deleted = numbers.splice(2, 3);

splice (start: number, deleteCount: number, ...items:

```
const numbers = [1,2,3,4,5];
                 0 1 2
const deleted = numbers.splice(2,3,6,7);
console.log(numbers); // [1,2,6,7]
console.log(deleted); //[3,4,5]
```

If we don't want to remove any thing - set 2nd value = 0

```
const numbers = [1,2,3,4,5];
const deleted = numbers.splice(2,0,6,7)
console.log(numbers); //[1,2,6,7,3,4,5]
console.log(deleted); // []
```

```
num = [1,2,3,4,5];
       0 1 2 3 4
num.splice(2,1,11,12,13);
```

o/p — 1,2,11,12,13,4

```
const num = [1,2,3,4,5];
             0 1 2
const del = num.splice(2,1,11,12,13);
document.write(num); o/p — 1,2,11,12,13,4,5
document.write(del); o/p — 3
```

Push:

```
num = [1,2,3,4,5];
n = num.push(6,7,8);
console.log(num); // [1,2,3,4,5,6,7,8].
```

Concat:

```
arr1 = [1,2,3];
arr2 = [4,5];
arr3 = [6,7,8];
arr = arr1.concat(arr2,arr3);
console.log(arr); // [1,2,3,4,5,6,7,8]
```

Traditional looping methods

```
array = [1,2,3,4,5,6];
for (index=0; index<array.length; index++)
    {
    console.log(array[index]);
    }
```

using do while

```
do{ console.log(array[index]
    index++;
} while (index<array.length)
```

for each — to call a function for each element of an array.

```
array.forEach(function(element){
    console.log(element);
});
```

Higher order fn

A function that takes another function as an argument or returns a function from it is called HOF.

```
function x() {
    console.log("Namaste");
}

function y(x) {
    x();
}
```

x → called back fn
y → Higher order fn.
x() → called back fn.

```
const radius = [3, 1, 2, 4];
```

radius of 4 circles.

```
const calculateArea = function (radius) {
    const output = [];
    for (let i = 0; i < radius.length; i++) {
        output.push(Math.PI * radius[i] * radius[i]);
    }
    return output;
};

console.log(calculateArea(radius));
```

o/p — array of area.

```
const calculateCircumference = function (radius) {
    const output = [];
    for (let i = 0; i < radius.length; i++) {
```

```
    output.push ( 2 * Math.PI * radius[i] );
  }
  return output;
};
console.log ( calculateCircumference (radius) );
const area = function(radius) {
    return Math.PI * radius * radius;
};
```

```
const radius = [3,1,2,4];
const area = function (radius) {
    return Math.PI * radius * radius;
};
const calculate = function (radius, logic) {
  const output = [];
  for (let i=0; i<radius.length; i++) {
    output.push ( logic (radius[i]) );
  }
  return output;
};
console.log (calculate (rad
```

map - map method calls the function for each element & returns an array of elements = m

O/P:-
2,3,4,5

```
<p id = "demo" ></p>
<script>
  Const num = [4,9,16,25];
document.getElementById("demo").innerHTML =
                numbers.map(Math.sqrt);
</script>
</body> </html>
```

Arrow function.

```
<p id = "myid" > </p>
<script>
  let myFunction = (a,b) => a*b;
document.getElementById("myid").innerHTML =
                myfunction(5,6);
</script>
</body>
```

O/P - 20

O/P - Hello Universe.

```
<script>
  let hello = " ";
  hello = (val) => "Hello " + val;
document.getElementById("myid").innerHTML =
                hello("Universe");
</script>
```

```
let num = [10,20,30];
Square = num.map((n,index) =>
    {
    return n*n;
    })
document.write(Square);
```

o/p — 100, 400, 900.

reduce

```
let num1 = [10,20,30];
Square = num1.reduce((sum,n) => {
    return sum + n*n;
});
```

filter method :—

— Each time const value will be set to each element of array

```
const numbers = [1,-1,2,3];
let sum = 0;
for (let n of numbers)
    sum += n;
    console.log(sum);
```

```
const sum = numbers.reduce((accumulator, currentValue) => {
    return accumulator + currentValue;
}, 0);
```

Callback fn, Initial value for accumulator

```
console.log(sum);
```

// acc = 0, cv = 1 ⇒ a = 1

// acc = 1, cv = -1 ⇒ a = 0

// acc = 0, cv = 2 ⇒ a = 2

// acc = 2, cv = 3 ⇒ a = 5

o/p — 5

If o skipped o accumulator set to 1st ele.

let num = [10, 20, 30];

square = num.reduce ((sum, n) => {

return sum + n + n;

});

Console.log (square)

Acc cv

10 + 400 = 410

Acc

410 410 + 900 = 1310

filter

Const words = ["spray", "elite", "elegant", "vandeofhl"];

Const result = words.filter ((word) => word.length > 6);

Console.log (result);

o/p — elegant, vandeofhl

Removing last element with splice.

num = [1, 2, 3, 4, 5];

n = num.shift (a);

Console.log (num); // [2, 3, 4, 5]

Cosle.log (n); // 1

Document object model

When a webpage is loaded, the browser creates a Dom of page. It is constructed as a tree of objects

<books>

<book>

<author> Carson </author>

<Pubdate> 05/01/2001 </Pubdate>

</book>

<Pubinfo>

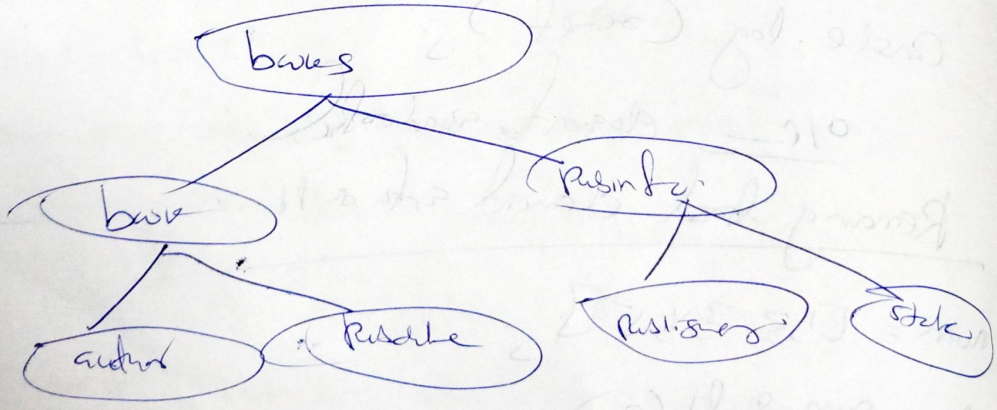
<Publisher> MSPress </Publisher>

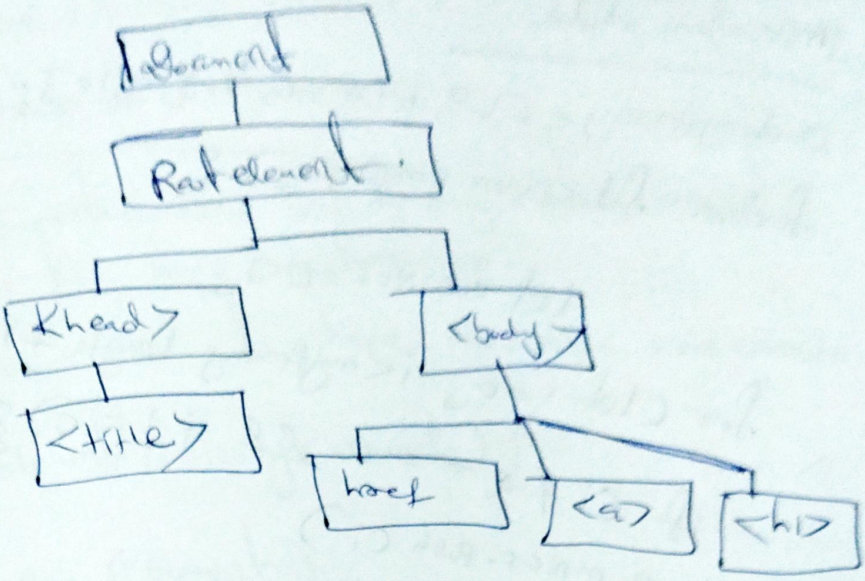
<State> WB </State>

</Pubinfo>

</books>

Dom represents the content of xml or html document as a tree structure.





- firstChild property returns the first child node of a node.

node value ⇒ sets or returns value of a node.

- getAttribute() returns value of an element's attribute.

setAttribute() ⇒ sets a new value to attribute.

hasAttribute() ⇒ returns true if a node has attribute otherwise false.

removeAttribute()

Functional Programming with JS -

(FP)

FP is a programming paradigm that defines a way programmers thinks for solving problems. It's a pattern for developing programmers using pure functions that avoid shared state & mutable data. FP is language independent.

a form of declarative programming.

Imperative programming :- how U do something.

declarative " :- what U do or something.

Imperative style

```
const myArray = [1,2,3,4,5,6,7,8,9,10];
function fetchEven (myArray) {
    let evenNos = [];
  for (let i=0; i<myArray.length+1;i++){
    if (i%2===0 && i!=0){
      evenNos.push(i)
    }
  };
};
return evenNos.
};
Console.log (fetchEven (myArray));
    // logs [2,4,6,8,10]
```

Declarative style—

```
Const myArray = [1,2,3,4,5,6,7,8,9,10];
function fetchEvenD (myArray)
{
return myArray.filter (num => num%2===0)
}
Console.log (fetchEvenD (myArray))
    // logs [2,4,6,8,10]
```

→ The declarative / functional approach. Process Blocker code.

Tenets of functional programming —

pure functions — For the same set of arguments passed, the function returns the same result always.

The pure function will not have observable side effects

```
function addElementToArray(element) {
    array.push(element);
}
```

```
var triple = function(x) {
    return x * 3;
}

var waffle = triple
waffle(30);
```

Filter is a function on the array that accepts another fn as its argument.

```
var animals = [
  { name: 'fluffykins', species: 'rabbit' },
  { name: 'caro', species: 'dog' },
  { name: 'Hamilton', species: 'dog' },
  { name: 'Harold', 'species': 'fish' },
  { name: 'ursula', species: 'cat' },
  { name: 'Jimmy', species: 'fish' } ]
```

Let we want to filter out dogs.

```
Var dogs = [ ]
for (var i=0 ; i<animals.length ; i++){
  if (animals[i] . species === 'dog')
    dogs.push (animals[i])
}
```

```
Var dogs = animals.filter (function (animal){
return animals.species === 'dog'

})
```

Functions like this that u send into other functions are called Called back functions.

```
Var isDog = function (animal){
    return animal.species === 'dogs'
}
Var dogs = animals.filter (isDog)
```

```
Var otherAnimals = animals.reject (isDog)
```

In Javascript, functions are values & u can exploit this by dividing ur code into small simple functions & Composing them together using higher order fn.

HOF → A function that takes other functions as arguments.

map transform data.

Var animals = [- - -] Same array.

We want to get an array of all the names of all the animals

Solving using the for loop -

```
Var names = []
for (var i=0; i<animals.length; i++) {
  names.push (animals[i].name)
}
Console.log (names);
```

Note - Map is a function on the array object.

```
Var names = animals.map (function (animal) {
  return animal.name + ' is a ' + animal.species;
});
```

Arrow fn → A new shorter & better syntax for fn's in ECMAScript 6.

Note -

- Filter expects its callback fn to return a true or false value that determine whether or not the item should be included in the array or not.
- map includes all items in array but it expects callback fn to return a transformed object.

that it will return new array instead of the original

Using map

Var names = animals.map(function(animal){ return animal.name })

using Arrow fn

Var names = animals.map((animal) =>

animal.name)

here u can get rid of { } and return stmt

Var names = animals.map((x) => x.name)

reduce — map, filter, reject.

animal → animal names

→ to express any list transformation

Var orders = [{amount:250}, {amount:400}, {amount:100}, {amount:325}]

Var totalAmount = 0

for (var i = 0; i < orders.length; i++)

{ totalAmount += orders[i].amount }

Console.log(totalAmount) 1075

var totalAmount

= orders.reduce(function (sum, order) {
Console.log("hello", sum, order)
return sum + order.amount
}, 0)
Console.log(totalAmount)

o/p — hello 0 {amount 250}
hello 250 {amount: 400}
hello 650 {amount: 100}
hello 750 {amount: 325}

1075

pure function —

- the function always returns same o/p for a given i/p
- No side effects.

```
function salut() {
  c.l. ("hello friend");
}
salut();
```

```
function salut() {
  return "hello friend";
}
salut();
o/p — hello friend.
```

```
function f(x) => y
function f(n) => n * 2
f(5)  // 10
```

```
function salut() {
  return "hello";
}
salut();
```

```
let name = "Shiv"
function sault() {
return "Hello ${name}"
}
sault()
name = "kendall"
sault()
name = "Tom"
sault()
```

→ different result.

→ different results each time.

→ it depends on external variable.

To qualify a function as pure function, it must have the following characteristics:—

- For the same set of arguments, the function returns the same results always. The pure function will not have observable side effect.
- For the same set of arguments, the provided function returns the same results always.

Write a fn to calculate the area of a sphere.

Area of sphere = $4\pi r^2$

```
let PI = 3.14;
const CalculateAreaOfSphere = (radiusValue) =>
    4 * PI * radiusValue * radiusValue;
CalculateAreaOfSphere(10); //returns 1256
```