Javascript

- Developed by Brandon Eich in 1995.
- Add Interactivity to Netscape.
- Mocha > Livescript > Javascript.
- Standardised by ECMA International in 1997 and ECMAScript or ES.
- It is single threaded. i.e. (1 callstack & 1 Heap)

* Variable isn't type bound but the value is type bound.
* Variable declaration:

var a = 10;
a = "Hello";
a = true;

* let :- can't declare the variable defined with 'let' keyword but can update it.
* const :- can't be reassigned and redeclared.

6 Primitive

i) number - 64 bit
ii) string
iii) boolean
iv) null
v) undefined
vi) symbol

1. Object = {}; key : value,

- Number.MAX_VALUE < infinity.
not a number = NaN

* General rules for constructing variable names.
  - can contain letters, digits, underscores & dollar sign.
  - must begin with letter, $, _
  - case sensitive
  - can not contain spaces.
  - keywords can not be used.

* Type of :-

typeof 10;

typeof

* typeof null; => object ⟶ Bug
* typeof NaN; => number ⟶ Bug

// line comment /* ... */ block cm

* Operators

Arithmatic : + - * / %

Assignment : = += *= etc

Urinary : ++ -- pre/Post

Comparison : >, <, >=, <=, !=, ==, ===, !==

Logical : ff, ||, !

Bitwise :- >>, <<, &, |, >>>

* Type Coercion :-

"1" + 1 ; output → "11"

"1" - 1 ; ——— 0

"1" * 5 ; ——— 5

"1" > 0 ; ——— true.

* == vs ===

1 == "1" ; ⇒ true → type not considered

1 === "1" ; ⇒ false → type considered

* Conditionals :-

if -- elseif -- else

var a = 0;

if (a > 0) {
  console.log("positive");
} else {
  console.log("zero");
}

when condition not given.
0, null, undefined are false.

* loops :-

* while loop

```
var i = 1;
while (i<=5){
    console.log(i);
    i++;
}
```

* for loop

```
for (i=1; i<=5; i++){
    console.log(i);
}
```

* do while. ← runs atleast one time & then checks condition.

```
do{
    console.log(i);
    i++;
}while (i<5);
```

* switch case

```
let variablename = "value";   ← desired value.
switch (variablename) {
    case "condition":
        console.log("condition statement");
        break;
    case "value":
        console.log("desired value");
        break;
    default:
        console.log("If nothing is true then execute me");
}
```

* functions :-

```
function showAlert(...) {        ← parameters, no need to declare just type varname.
    alert("Hey!!");
}
showAlert(...);   ← calling.
          ↑ attribute.
```

Variable length Arguments.

if don't give value to variable in JS it gives value as "undefined".

* ~~function~~ Hoisting.. * variable hoisting.

```
function hoistDemo(){
    console.log(i);
    var i=10;
}
hoistDemo
```

=> undefined

Bcoz of hoisting JS get know variable is created but doesn't give its value.

Similarly function hoisting using function before creating it.

Scope.

~~var~~ ~~global~~

```
var name = "global";
console.log(name);
```

global

```
function scopeDemo(){
    var name="functional or lexical";
    console.log(name);
}
scopeDemo();
console.log(name);
```

lexical

* Execution context.
* ~~Execution Stack~~.

Hard refresh : Ctrl+Shift+r

Global : GEC is the default execution context where all js code that is not inside any function. There can be only one GEC in a program.

Functional : FEC is new execution context created whenever a function call encounters.

* Function Expression :

Named F.E

→ var variableName = function funcName(n){
    ...;
    return ans;
};

calling ⇒ console.log(variableName(5));

* cannot call by function Name but can use within function.

Anonymous F.E

— var varName = function (n){
    ...;
    ...;
    return ans;
};

It has funcname same as varName.

* Function Declaration

Hoisting works

works everywhere.

Function Expression.

Hoisting Doesn't work.

when we want to use function only after creating

Passing function as arguments :-

console.log (funct1 (funct2))

* funct2 will execute after the execution of funct1.
* funct2 will called as callback function.

* Arrays :- In JS can be heterogeneous.

- var varName = [1,2,3,4,5] ;
- var varName = new Array (1,2,3,4,5);
- index start from 0. Array(size)

varName [0] ⇒ 1

varName [5] ⇒ Undefined.

- varName[0] = 10 ; ⇒ [10,2,3,4,5]

varName[7] = 20 ; ⇒ [10,2,3,4,5, empty x2, 20]

varName.length ; ⇒ 8

empty spaces doesn't occupy memory.

* function on Arrays :-

var arr = [1,2,3,4]

arr.push(10) ; ⇒ [1,2,3,4,10]

arr.pop() ; ⇒ [1,2,3,4]

arr.shift() ; [2,3,4]

arr.unshift(1) ; [1,2,3,4]

Splice :- splice (startindex, deletioncount, elements to be inserted)

arr.splice(1,1) ⇒ [1,3,4]

arr.splice(1,0,2) ⇒ [1,2,3,4]

arr.splice(1,2,20,30) ⇒ [1,20,30,4]

arr.splice(1) ⇒ [1,20]

slices a given part of an array and returns that sliced part as a new array.

* Iterate over array :-

```
var arr = [1,2,3,4,5];
for (var i=0; i<= arr.length; i++) {
    console.log(arr[i]);
};
```

(can pass index & arr)

```
function print(element) {
    console.log(element);
}
arr.forEach(print); — No need to pass args.
```

* Object :- {key:value,} — Property

```
var obj1 = { key1 : 'val1', key2 : val2 }
obj1.key1 = value;   or obj["key"]
```

- var obj = {};
- var obj = new Object();

⚠ where key name is invalid you cannot use obj.key . you must use obj["key"]
(.notation) ([] notation)

* delete obj.key; or delete obj["key"]

* Iterate over object.

```
for (var prop in objname) {
    console.log(prop, student[prop]);
}
```

OR

* Object.keys(objName);
* Object.getOwnPropertyNames(objName);

* Nested Objects.

```
var obj1 = { "key1" : "val1",
             key2 : {key: 'val2',
                     keyb: 'key2'
                     }
           };

obj1.key2.keyb",
obj1.key2['keyb']
```

* Arrays are objects with additional properties.
  keys are indices.
  Object does not have length property
  arr.key = 'xyz' but doesn't count in length.
  only counts when key is int.

* Iterating over arrray using for in.

* Timing Events
  - setTimeout(function name only, number);
    miliseconds
    1000 ms = 1 sec
    will run after specified time.
  - setInterval(functName, 1000);
    will run repeatedly after every specified time.
  - clearInterval(var_name));
    you need to use variable when creating interval method to use the clearInterval method.

* Js is pass by value except objects & arrays

* Deep Copy :- Values copied to new variable & disconnects from old variable ~~references for object~~ primitive.

* Shallow copy :- new variable & old variable values ~~and~~ are connected.

* Spread operator :-
var obj1 = {... obj2} ; ~~Deep copy~~
Deep copy
creates new object.

* var obj1 = Object.assign({}, source)
~~Deep copy~~ Deep copy. creates new object.

* var obj1 = obj2 ;
shallow copy gives same reference.

* Arrow function :-

var ans = (x, y) => {
return x*y;
}

if only one statement you can remove { } brackets & return keyword.

var ans = (x, y) => x * y;

* Map Method :- does not modify original array.
it returns new array value.

let arr = [1, 2, 3];
let Modified = arr.map(function modify(i) {
return i * 2;
});

Reduce Method :- Combines the elements in a sequence and give us reduced single value.

arr.reduce(function, initialvalue);

filter Method :-

arr.filter(funName);

* for...in → returns list of key
for...of → returns list of values.

DOM

* Document Object Model - (Logical structure of webpage.
The topmost object in the DOM is the browser's window. After the window, DOM is the document displayed in the browser's window.

- document.documentElement :- To access whole html
- document.head :- To access head
- document.body :- To access body tag

- window.document = document

- screen = window.screen

- function test(){
  return this;
}
console.log(test()==window);

⇒ True

* Fetching Element.
* By id

var ele = document.getElementById('id');
ele.style.color = 'red'; ⇒ ele becomes of red color.

- By tags:

```
var ele2 = document.getElementsByTagName('tagName');
ele2[1].innerText = "Changed Content";
```

⇒ Text changes ↖ innerHTML → gets or sets text.

- By class:

```
var ele3 = document.getElementsByClassName('class');
ele3[2].style.backgroundColor = 'Cyan';
```

- By CSS selectors:

```
var ele4 = document.querySelector('#idname');
```

* will return first matching result in case of class.

* querySelectorAll('.className') for all.

* Event Handling :-

```
<button id="btn" onclick="alert('Hello!!')">
    SAY HELLO! </button>
```

(on + eventname)

OR

```
var xyz = document.getElementById('btn');
xyz.addEventListener('click', function(){});
```

(Nameonly)

— can add multiple listeners in case second.

* Script tag :- you should add at body end.
else html code not render properly.

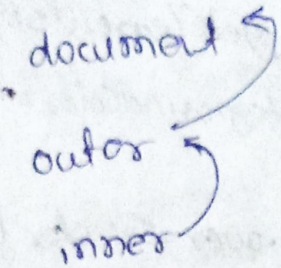
mouse events :- click, mouseover, mouseout, keypress, keydown, keyup

will not detect tab, alt, ctrl, shift, up & down, space etc case keydown for above.

* event.keyCode → pressed output a => 65, b = 66 likewise

* Pass 'event' as argument.

* Propogation of Event.



→ to stop propagation.

event.stopPropogation();

* Strict Mode

varName=0; —will have global scope

- it should throw error generally.
to do that
⇒ "use strict";

use can use strict mode for specific functions only.

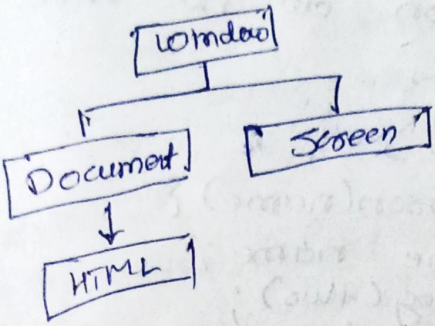
* eval( ); evaluates operations.

eval(('4*5'));

eval(opt+" "+operator+" "+opt2);

* window.scrollTo(x, y); x/y px.
* window.pageYOffset; → will position,
* window.scrollBy(x, y); x/y px steps.

BOM => Browser Object Model.



* IIFE : Immediately Invoked function Expression.

```
(function{
  ...
  ...
})();
```

var inside it will have block level scope.

* Closures : function + lexical environment in which function was created.
- can use the variables of lexical environment though the function was executed.

```
var i = 10;
function outer(){
  var j = 20;
  console.log(i, j);
  var inner = function(){
    var k = 30;
    console.log(j, k);
  }
  return inner;
}
var inner = outer(); inner();
```

* Arrow functions :

var multiply = (x, y) => { return x * y };

var multiply = (x, y) => x * y; - if single statement

var square = x => x * x ; - if single arg

arrow function do not have their own binding.

```
function Person(name) {
    this.name = name;
    console.log(this);
    /* setTimeout(function() {
        console.log(this);
    }, 1000); */
    setTimeout(() => console.log(this), 100);
}
var p = new person("sandip");
```

window object

Person {name: "sandip"}

* "this" keyword :- look at function call at (where) & how & where)

refer to an object

- Rule 1* :- 'new' keyword :

```
function vehicle() {
    console.log(this);
}
new vehicle();
```

output: vehicle {}, vehicle {}

1. create a new object {}
2. {} link to prototype of vehicle constructor func.
3. vehicle() with 'this' equal to vehicle object
4. return {} ... vehicle function.

Rule 2nd :- Explicit binding rule.
Where we tell what will the value of this Arg.

~~func.call(thisArg);~~ func.call(thisArg);
func.apply(thisArg);

or. const john = {
name: 'John'};

function ask(){
console.log(this, this.name);
}
ask();
=> window{...}

ask.call(john);
=> {name: "John"} "John"

ask.apply(john);
=> {name: "John"} "John".

~~Hard binding~~
~~Local binding~~

~~ask~~ ~~var localask = john.name;~~

~~localask();~~
=> ~~Hello~~ john
window{ }

ask

~~Hard binding~~

var raj = {name: 'Raj', greet: function(){ console.log("Hello", this) }};

raj.greet() => Hello {name: "Raj", greet: ƒ}

var localAsk = raj.greet;
localAsk() => Hello window{...}

Hard Binding

thisArg.

```
var localGreet = raj.greet.bind(raj);
```

this

```
localGreet() => Hello {name: "Raj", greet: f}
```

- 3rd Rule:- Implicit Binding

```
var raj = {
    name: 'Raj',
    greet: function () {
        console.log('Hello', this);
    }
};
```

Implicit.

```
raj.greet()
=> Hello {name: "Raj", greet: f}
```

```
var localAskFunc = raj.greet;
localAskFunc()
```

—— called in global context.

```
=> Hello window {---}
```

④ 4th Rule : Default Binding.

```
function ask() {
    console.log(this, this.name);
}
ask()
```

```
=> window {---}
   undefined
```

* name property = '' => empty string.
  in new versions of JS.

```
var person = {
    name: "John",
    ask: function() {
        console.log(this);
    }
}
new (person.ask.bind(person))();
```

=> ask{}
ask {}

Precedence :- Rule 1 new > Explicit Binding > Implicit Binding > default Binding.

* Function To Create Objects

```
function createStudent(name, rollNo, marks){
    var student = {};
    student.name = name;
    student.rollNo = rollNo;
    student.marks = marks;
    return student;
}
var student1 = createStudent("Sandip", 1, 30);
student1;
```

=> {name: 'Sandip', rollNo: 1, marks: 30}

OR using Constructor.

```
function CreateStudent(name, rollNo){
    this.name = name;
    this.rollNo = rollNo;
}
```

Calling as constructor → `var student1 = new CreateStudent("Sandip", 1, 30);`

Calling as simple func → `var student2 = CreateStudent("XYZ", 2, 20);`

`student1;` => CreateStudent{name: 'Sandip', rollNo: 1}

`student2;` undefined.

Bcoz simple calling only gives return value as output.

* Adding Behaviour to Objects:

```
function Vehicle(no, price){
    this.no = no;
    this.price = price;
    this.getPrice = function(){
        return this.price;
    }
}
var vehicle1 = new Vehicle(2,500);
var vehicle2 = new Vehicle(3,300);
vehicles;
```

⇒ Vehicle{ no:2, price:500, getPrice: f }

⇒ Vehicle{ no:3, price:300, getPrice: f }

getPrice: f get created for every object though it is not necessary as it will take lot of space ie memory wastage.

* Prototype: Every function you are creating it creates 2 objects. 1 is function itself & 2nd is prototype.

Vehicle.prototype;

Useful when using func as constructor.

⇒ ▼ {constructor: f}
  ▼ constructor: f Vehicle(no, price)
  ▼ __proto__ : object

Vehicle.prototype.constructor;

⇒ f Vehicle(no, price){

}

* Why Prototypes: ~~Should have To create only one object~~ To avoid creation of new object for same function while creating its parent functions object.

- To share data & behaviour without creating new copy.

```
function Vehicle (new, price){
    this.new = new;
    this.price = price;
}
Vehicle.prototype.getPrice = function(){
        return this.price;
    };
var vehicle1 = new Vehicle(2,500);
```

* More Properties around Prototypes.

__proto__ or dundur proto — ES6 deprecated

vehicle1.__proto__ ;

⇒ { getPrice: f , constructor: f }

vehicle1.__proto__ === Vehicle.prototype

⇒ true.

equal

Object.getPrototypeOf(vehicle1);

* to check is prototype or not

Vehicle.prototype.isPrototypeOf(vehicle1);

⇒ true.

* hasOwnProperty,

vehicle1.hasOwnProperty('price');

⇒ true

vehicles.hasOwnProperty('getPrice');

⇒ false.

vehicle1 looks in → Vehicle.prototype looks in → Object.prototype

* object vs Object → all objects in JS inherit

↳ constructor function Object

datatype.

{ }