1991 - Linux Started
1995 - JS started

Control Structure
- if else - often used
- for - abstractions
- Switch - better way available in JS
- while - Normal use
- do while - Normal use

Operators
- Arithmatics + , - , * , / , % , ++ , -- , += , -= , *= , /*= , == , === , !
- Logical && , || , !.
- Bitwise | , & , ^
- Ternary ?: [a ? b : c]
- Relation *

History of JS
- improved performance.
- flexibility

Tech debt
+ → a+a = aa [concate & addition]
== [equality check]
null → type of null = object [Type of null is shown as object due to tech debt]

jQuery → iteration
backbone.js → good web application
Angular JS → used in typescript
React JS → heavily used by company having web application
Node JS → JS framework to run outside the browser JRE(VM)
Gatsby → build on top of React
Next.js → framework for creating website faster
vue.js → open source, angular + React, built by community

JS is ~~Static~~ dynamic type ~~language~~. variable

JS Variables ~~[var]~~

Don't have to compile

Container which holds data

Categories:

Primitive	Non-Primitive
String [String]	object {}, → Object
number [Number]	array [] → Array
boolean [Boolean]	
null	
undefined	
Symbol (ES6)	

In JS everything is an object.

Keywords for variable declaration

- var (ES5) → functional scope – allows redeclaration
- const } (ES6) → lexical scope – cannot reassign the value
- let } → doesn't allows redeclaration in node

In const, you can not reassign ~~pr~~ primitives but can change non-primitive values.

```
Const arrVal = {};
arrVal.nam = "a";
```
✓

```
Const a = 2;
a = true;
```
✗

pass by reference

```
var num = 10
Var value = "Call"
```

num → 10 value → Call

Var is used in ES5
Const, let used in ES6

num + value
→ 10 Call
typeof value
→ String

JS does typecasting
10 + "Value" = String ("10") + "Value"

var num = 10;
var num2 = 12;
num + num2
→ 22

Anything which is inside the " " is string in JS

Var num2 = "12"
num + num2
→ "1012"

num * num2 [This is for *, /, -, %]
→ 120

Var cost = "100" → String to solve this we typecast as
var cost = Number ("100"); OR var cost = (+"100");
cost
→ 100 → Number

cost
→ 100 → Number

100 + + "0.18" → cast that to a number
→ 100.18

```
var cost = "100";
var tax = "0.18";
cost * tax
→ 18
cost + tax
→ "100.18"
(+cost) + (+tax)
→ 100.18
cost + (+tax)
→ "1000.18"
Number(cost) + Number(tax)
→ 100.18
var booked = true  [true will converted into 1]
var cost = 100
cost + booked
→ 101
var cost = true
cost + booked
→ 2
var cost = false
cost + booked
→ 1
cost * booked
→ 0
var cost = true
cost * booked
→ 1
```

var name = null
var exp = 12
name + exp
→ null12

var name2 = null [converts null to zero]
typeof name2
→ object
nam2 + exp
→ 12

var name3 = undefined
name3 + exp
→ NaN

undefined + undefined
→ NaN

undefined + null
→ NaN

undefined + 2
→ NaN

undefined + "2"
→ "undefined 2"

null + null
→ 0

undefined + true
→ NaN

undefined / undefined
→ NaN

undefined / null
→ NaN

null / null [0/0 = ∞]
→ NaN

in + operation Concatenation will take precedence over addition

null + true
→ 1

null - true
→ -1

"A" - null
→ NaN

null - undefined
→ NaN

"A" - undefined
→ NaN

Logical Operators [left to right]

&&
true && false → false
true && true → true
false && true → false
false && false → false

||
true || false → true
true || true → true
false || true → true
false || true → false

!
!0 → 1
!1 → 0

it we have — Result into — will ends to

Number → 0, NaN = false [Falsy value]
String → "" = false [Falsy value]
Undefined → false [Falsy value
Null → false [Falsy value]
false → false

Page No.

var y = 6, z = 7
y && z → truthy && truthy ⇒ true
→ 7

(0, NAN, "", Null, Undefined) → Other than this values are always truthy value → true

Boolean("")
→ false

Boolean(0)
→ false

Boolean(NAN)
→ false

Boolean(undefined)
→ false

Boolean("hell")
→ true

Boolean(1)
→ true

Boolean("null")
→ true

In Logical operators, if value is falsy, then result will be false Otherwise true.

Other that that if we try to convert this into boolean, then it will give true value.

var y=6 , z=7
z && y
→ 6

y=0, z=7
y && z → false && 7
→ 0
Here first condition is false therefore it will not check for next one.

2 && 7
→ 7

2 && 7 && 5
→ 5

2 && 7 && 5 && 34
→ 34

Boolean
if(!(a && b && c && d && e))
{ } → Give boolean value as output
var value = a && b && c && d && e
=

It will stop whereever it get false.

If last value is zero, so it will evaluate to false

if (34 && 7 && 5 && 34) {console.log("Hi")}
→ ~~console~~ Hi

if (34 && 0 && 5 && 34) {console.log("Hi")}
→ Nothing will display. If we have else condition, then that will be executed

RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI
JUHU VERSOVA LINK ROAD, VERSOVA, ANDHERI (WEST), MUMBAI - 53.

```
if (34 && 0 && 5 && 7 && 9)
   { console.log ("Hi") }
else
   { console.log ("Hello")}
```
→ Hello

```
if (-34 && 5 && 0 && 7 && 9)
   { console.log ("Hi")}
else
   {console.log ("Hello")}
```
→ Hello

Primitive Type

Logical AND (&&)

"Hello" && 1 && 5 && 34
→ true && true && true && true
→ true && true
→ true

" " && 4 && "Major" && null
→ false && true && true && false
→ false && false
→ false

When internally doing conditional check, JS will convert the value to boolean

Logical OR (||)

0 || "Class" || 28 || null
→ false || true || true || false
→ true || true
→ true

Non-Primitive type

- we don't have falsy values
- [], {} are always truthy values

```
if ([]&&5)
  { console.log ("a")}
else
  { console.log ("b")}
```
→ a

```
if ([] && "")
  { console.log ("a")}
else
  { console.log ("b")}
```
→ b

```
if ([] && {})
  { console.log ("a")}
else
  { console.log ("b")}
```
→ a

You can change any value into boolean value
If we add '!' in font of any variable, JS engine will convert that into a boolean value & invert the value

JS explicitly converts value into boolean

```
var value=8;
!value;
```
→ false

```
!!value; — Reverse the result
```
→ true

All ES6 code is converted to ES5
ES6 code is not fully supported in browser
Browser internally converts let into var

(!! "Hello" && !! 1 && !! 5 && !! 34) ← Typecasting
→ true
JS will do this itself even if we are not using it like this & decides the value is truthy & falsy based on true & false.

var valuex = [];
!! valuex;
→ true

var valuex = {};
!! valuex;
→ true

In JS, we have dynamic type variables.

var value = undefined;
!! value
→ false

Primitive holds a memory location where the value is stored
Engine will reallocate new memory & it will assign to that

In non-primitive, value is passed by refference

type is evaluated at runtime
∴ JS is a dynamic language.

2 Numbers → addition
1num + 1bool → add
undef + null → NaN
undef + bool → NaN
null + true → 1
1 + false → 1
undef + true → NaN
~~1 + fa~~ true + true → true

1 + ""
→ "1"

Any non-primitive in JS is always a concatenation.
JS internally will convert into string type.

[] + []
→ ""

"" + ""
→ ""

[1] + [2]
→ "12"

[1] + [2,3,5,7]
→ "12,3,5,7"

"1" + "2,3,5,7"
→ "12,3,5,7"

{} + {}
→ "[object Object][object Object]"

({}).toString()
→ "[object Object]"

whenever we add dot with variable it becomes an object.
JS wrap that into an object.

{} + []
→ 0

{} + 1
→ 1

[] + {}
→ "[object Object]"

arr.length means last index + 1
its not mean length

RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI
JUHU VERSOVA LINK ROAD, VERSOVA, ANDHERI (WEST), MUMBAI - 53.

[] + 1
→ "1"

How to declare array & object in JS
var arr = [];
typeof arr
→ "object" because it is a container

Array is nothing but a specialized object in JS
we can Read data of array using index
In JS array is nothing but an object in which index are nothing but a key of array.

Whenever we try to add a dot after a number / boolean / string, it will become a string class, number class

var arr = [];
console.log (arr);
→ []
console.log (arr.length)
→ 0

var arr = [1,2]
console.log (arr.length)
→ 2
var arr = [2,3,4,5,null,true]
→ 6

var arr = [2,3,3,null,true,[],{}];
console.log (arr.length);
→ 7

'.' converts into a class

```
console.log (arr);
→ (7) [2, 3, 3, null, true, Array(0), {...}]

var arr = [2,3,3,null,true,[],{}]
arr[4] = 67;        → replace the value at index 4
console.log (arr);
→ (7) [2, 3, 3, null, 67, [], {}]

arr[3] = "Hello";
console.log (arr);
→ (7) [2, 3, 3, "Hello", 67, [], {}]
```

Array doesn't have fixed length

```
arr[300] = "Long Array";
console.log (arr);
→ (301) [2, 3, 3, "Hello", 67, Array(0), {...}, empty x 293, "Long Array"]

console.log (arr.length);
→ 301

arr[5000] = "Very long array";
console.log (arr);
→ (5001) [...]

arr["oh"] = "Really";
console.log (arr.length);
→ 5001
console.log (arr);
→ It will show oh is the key
```

So array is an object is valid.

JS will look for only numeric index in case of array

arr[5000] → Read the array
→ "Very long array"

arr[0]
→ 2

arr[300] → Read the content in the array
→ "Long Array"

arr["oh"]
→ "Really"

arr["0"] → we can use string value also to
→ 2 read content in that array

arr["300"] In JS keys are always strings.
→ "Long Array"

index are nothing but a key but we don't have to define that key.
There is an object we have to add the key, without key we do not assign the value to it.

arr["5000"]
→ "Very long array"

typeof arr
→ "object"

```
var obj = {}
typeof obj;
→ "object"

var helo = {name:"Akansha", loc:"Mumbai"}
helo
→ {name:"Akansha", location:"Mumbai"}
    location :"Mumbai"
    name:"Akansha"

helo.location
→ "Mumbai"

helo.name
→ "Akansha"

helo.exp = 12          } get added to array
helo.role = "engineer"
helko
→ {name:"Akansha", location:"Mumbai", exp:12, role:"eng."}

helo.know = ["C","html","css","MySQL"]
helo
→ {name:"Akansha", loc:"Mumbai", exp:12, role:"eng.",
        know: Array[5]}
```

name, loc, exp, role } Primitive

know → non primitive

hello.know
→ Read the array.

keys : values
String Pr, Nonpr.

JSON is mode of transport like xml.

helo.available : false
helo
→ new key added having value false.

helo.toString() always gives this value.
→ "[object Object]" → truthy value

helo.know
→ (4) ["c", "html", "css", "MySQL"]

var know = helo.know
know
→ (4) ["c", "html", "css", "MySQL"]

know[0]
→ "c"

know[3]
→ "MySQL"

OR

helo.know[2]
→ "css"

helo.know[3]
→ "MySQL"

helo.work = { one : 1, two : 2 }
helo.work
→ {one : 1, two : 2}

helo.work.one
→ 1

We do array lookup using index

To lookup object → 2 ways [. OR []]

helo.name	helo["name"]
→ "Akansha"	→ "Akansha"

helo.["work"].one	helo.work.["one"]
→ 1	→ 1

But in array it allows only bracket notation

helo.know.0	helo.know[0]
→ give error	→ "C"

In array keys is always string ie. it can hold anything.

helo.full name
→ give error

To handle this, use []
helo.["full name"] = "Akansha Gorivale"
→ "Akansha Gorivale"

helo.["full name"]
→ "Akansha Gorivale"

[Arrays are referenced using [] notation.
Normal words using . notation.]

helo.full-name = "Akansha Gorivale"
→ "Akansha Gorivale"

underscore is treated as part of word whereas - is considered for joining the words.

helo ["full-name"] = "Akansha Gorivale"
→ "Akansha Gorivale"

JS variables → let & const at the runtime changes the values type whenever the value assign to it.

Functional Scope
In JS variables get hoisted → means whenever we run a JS function, the variable defined in the function will get basically get hoisted at the top

Lexical Scope
Any block which is inside functional scope
For eg., SD card has lexical scope of camera ~~inside the camera~~
it cannot come out the camera & occupy the space
This is the functional scope

Phone → functional scope
SD inside it → lexical scope

cd /desktop/missingSkill/

Functional scope:
var a=5;
Console.log (a);
→ 5 → Global functional Scope

Similarly we can do this also:

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

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

```
a=20;
console.log(a);
var a=10;          //var declaration get hoisted
```
→ 20

In this cass what JS engine will do:

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

it will again redeclare itself at the Top

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

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

Whenever we declare a variable JS assigns undefined variable.

```
console.log(a);
a=10;
console.log(a);
```

```
var a;
var a;
a=20
```
→ undefined, 10

(internally it will work like this something →)

```
var a;          //var hoisted
var a;
console.log(a);
a=10;
console.log(a);
a=20;
```
→ undefined, 10

JS internally used, undefine value a to a variable unless you assign any value to variable.

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

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

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

[declaration & assignment]

Only var dec get hoisted not assignment

Page No.

```
var a;
var a;
a=10;
console.log(a+b);  //number+undefined → NaN
a=20;
console.log(a);
var b;
```

→ NaN, 20

```
var a;
var a;
a=10;
console.log(a+b);
a=20
console.log(a);
var b = "ok"
```

→ NaN, 20

```
var a;
var a;
var b = ~~10~~;
~~a~~ a=10;
console.log(a+b);
a= 20
console.log(a);
~~var~~ b = "ok";
```

→ NaN, 20

Let

```
let a;
let a;
```

→ error

```
const b;
a=10;
console.log(a+b);
a=20
console.log(a);
b="ok";
```

→ error at line 2 because in ES6 we cannot redeclare it, we can reassign it

```
let a;
Const b; → error
a=10;
Console.log(a+b);
a=20
console.log(a)
b="ok";
```

→ error in 2nd line because whenever we declare const type it has to be declared as well as assigned/initialized.

```
let a;
Const b=1;
a=10;
console.log(a+b)
a=20
console.log(a)
b="ok"          → error
```

→ executed till line 6 & error at last line

```
11
20
```

Assignment cannot be done to constant variable.

```
let a;
a=10;
console.log(a+b); → error
a=20;
Console.log(a)
Const b="ok";
```

→ error at line 3 because

JS will consider whole file as a global function

Page No.

In ES6 const & let cannot get hoisted.
It cannot access b before initialization

```
let a;
a=10;
console.log(a+b)    //Num+undefined → NaN
a=20
Console.log(a)
var b="ok";    //only var dec get hoisted
```

→ NaN, 20

```
let a;
a=10;
Console.log(a+b)    → error
a=20
Console.log(a)
let b="ok"
```

→ error at line 3
cannot access b before initialization

```
var a;
a=10;
console.log(a+b)
a=20
console.log(a)
var b="ok"
```

→ NaN
20

```
var a;
var b = "ok"
a=10
console.log (a+b)
a=20
console.log (a)
```
→ 10 ok , 20

```
var a;
var b ="ok"
a=10
console.log (a+b)
a=20
function hello() {
    console. log (a) }
hello();
```
→ Look
20

```
var a;
var b ="ok"
console.log (a+b)  //undefined + "ok" = concat
a=20
function hello () {
    console. log (a)}
hello()
a=10;
```
→ undefined ok
20

```
var a;
var b = "ok"
Console.log (1, a+b)
a=20
function hello() {
    var a = 3;
    Console.log (2, a);
}
hello();
Console.log (3, a)
a=10
Console.log (4, a)
```

→ 1 undefinedok
2 3
3 20
4 10

Whenever we declare & assign var inside a function, it will work inside the function only. It doesn't have any scope outside the function. If var is declared outside the function then that variable get over-written.

```
var a;
var b = "ok"
Console.log (1, a+b)
a = 20
function hello() {
    a = 3
    Console.log (2, a);
}
```

```
hello();
Console.log(3,a)
a=10
console.log(4,a)
```

→ undefined ok
2 3
3 3
4 10

[because variable has a functional scope & functional scope is defined using var keyword

In the code, function portion is a functional scope & overall code is global functional scope
Any code will have 1 global functional scope.

```
var a=10;
var a=5;   → Rewrite the value
var b="ok"
Console.log(1,a+b)
a=20
function hello(){
    console.log(2,a)
    var a=3   - // var get hoisted
}
hello()
Console.log(3,a)
a=10
Console.log(4,a)
```

→ 15 ok
2 undefined
3 20
4 10

[Variables get hoisted only at the functional scope

Global scope → Parents functional scope.

Page No.

```
var a = 10
var a=5
var b = "ok"
console.log (1, a+b)
a=20
function hello() {
    var a;          // become a functional scope be cause
    console.log (2, a)   var keyword is used.
    a=3;
}
function hello2(){
    console.log (2, a) //callback to global fun scope.
    a=3;               //updating global variable
}
hello2();
hello();
console.log (3, a)
a=10
Console.log (4, a)
```

global scope

local scope

→
1 Sok
2 20
2 undefined
3 3
4 10

Whenever we don't use var keyword, JS will basically callback to its global functional scope From local to global scope.

RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI
JUHU VERSOVA LINK ROAD, VERSOVA, ANDHERI (WEST), MUMBAI - 53.

In ES 5 var get hoisted at the top of the functional scope.

```
var a=10;
var a=5
var b ="ok"
console.log (1, a+b)
a = 20
function hello(){
    var a
    console.log (2,a)
    a=3                    // destroy the var
}
function hello2() {
    consde. log(2,a)
    var a=5                // having local scope
}
hello2();
hello();
console. log(3, a)         //look for global scope.
a=10
console. log(4,a)
```

→
1 5 ok
2 undefined
~~2~~ undefined
3 20
4 10

```
var a =10
var a=5
var h="ok"
console.log (1, a+b)
a=20
function hello(){
    var a
    console.log (2,a)
    a=3
}
function hello2(){
    a = a
    console.log (2.1, a)
}
hello2()
hello()
console.log (3,a)
a=10
console.log (4,a)
```

→ 1 5ok
2.1 20
2 undefined
3 20
4 10

RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI
JUHU VERSOVA LINK ROAD, VERSOVA, ANDHERI (WEST), MUMBAI - 53.

```
var a = 10
var a = 5
var b = "ok"
console.log (1, a+b)
a = 20
function hello(){
    var a
    Console.log (2, a)
    a = 3
}
function hello2(){
    a = a+a
    console.log(2.1, a)
}
hello2()
hello()
console.log (3, a)
a = 10
Console.log (4, a)
```

a will get modified here as 40

→
```
1 5ok
2.1 40
2 undefined
3 40
4 10
```

```
var a=10
var a=5
var b = "ok"
console.log (1, a+b) //undef+undef = NaN
a=20
function hello(){
    var a
    console.log (2, a+a);
}
function hello2(){
    a=a+a
    console.log (2.1, a)
}
hello2()
hello()
console.log (3, a)
a=10
console.log (4, a)
```

→
```
1 5ok
2.1 40
2 NaN
3 40
4 10
```

- Hoisting means var comes at top of it scope at runtime
- At runtime it decides whether to hoist the var or not
- ES5, var will get hoist to its nearest functional scope.

RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI
JUHU VERSOVA LINK ROAD, VERSOVA, ANDHERI (WEST), MUMBAI - 53.

Functional Scope < global / local ES5 hoisted at Functional Scope

```
var a=10
var a=5
Var b = "ok"
console.log(1,a+b)
a=20
function hello(){
  var a = "7"
  console.log(2,a+a)
}
function hello2(){
    a=a+a
    console.log(2.1,a)
}
hello2()
hello()
console.log(3,a)
a=10
console.log(4,a)
```

→
```
1  5ok
2.1  40
2  77
3  40
4  10
```

ES6 variable will not get hoisted
It will not access before it is declared
Will throw an error

Lexical Scope

If function size is keep on increasing then, var will still get hoisted at the top of the functional scope.

In JS, var is like a container
At the runtime it decides a type of var
[adapts to the environment]

```
var a = 10;
var a = 5;
Var b = "ok"
consde.log (1, a+b)
a=20
function hello(){
    var a = "7"
    consde.log (2, a+a)
    a=3
}
function hello2(){
    a = a+a                    // Move to nearest fun scope
    Console.log (2.1, a)       i.e. global fun scope
    var c;
}
hello2()
hello()
console.log (3, a)
a=10
Console.log (4, a)
```

→ 1 50k
2.1 40
2 77
3 40
4 10

~~Ehar~~ Twist in hello2()

```
function hello2(){
    a=a+a
    console.log(2.1, a)
    var c                    ← declared here but doesn't assign any value in the current functional scope
    function hello-inside(){
        a=5
        console.log("inside", a)
        c=9
    }
    console.log("outside", c)
    hello-inside()
}
```

→ 1 50k
2.1 40
outside undefined — because ←
inside 5
77
5
10

[a Functional scope does not go to the child Functional scope]

[child cannot access assignment if var is not defined]

Change in hello2 & hello & declaration

```
function hello2(){
  var a = a+a   //undefined+undef → NaN
  console.log(2.1, a)
  var c
  function hello-inside(){
    a=5
    console.log("inside", a)
    c=9
  }
  console.log("outside", c)
  hello-inside()
  console.log("outside 2", c)
}
function hello(){
  var a = "7"
  console.log(2, a++a)
  a=3
}

var a=5
var b="6"
console.log(1, a++b)
a=20
```

→ 1 11
2.1 NaN
Outside undefined
inside 5
outside 2.9
2.77
3 20
4 10

Change in hello2()

```
function hello2(){
  var a = a+a
  c=11
  console.log(2.1, a)
  var c
  function hellow_inside(){
    var a=5
    console.log("inside", a)
    c=9+b
  }
  console.log("outside", c)
  hello_inside()
  console.log("outside2", c)
}
```

→

1	11
2.1	NaN
inside	5
outside	11
outside2	96
3	20
4	10
2	77

Functional scope starts with a function
Lexical scope starts with {}

Lexical Scope

```
let a=4
if (a){ // Boolean (4) → True
    let b=9
    console.log (a+b)
    b=a+b
  }
```

→ 13

```
let a=4
if (a){
    let b=9
    let a=9
    console.log (a+b)
    b=a+b
}
```

→ 18

```
let a=4
if (a){
   let b=9
   let a=9
   console.log (a+b)
   b=a+b
}
function hello (){
  if (a){
       console.log (a+b)
     }
else
```

Both functional & lexical has a local & global

```
{
}
hello();
```

→ 18
Error : b is not defined

If in prev code we change to var b = 9

→ 18
22 [b will get moisted]

How execution happens

→ First if variable is ES6, it will check in lexical scope first, then parent's lexical scope., functional scope, parent's scope

```
let a = 4    var b
if (a) {
    let b = b = 9
        let a = 9
        console.log (a+b)
        b = a+b
}
function hello() {
    let b = 5
    if (a) { console.log (a+b)
             b = 20 }
    else { let b:  }
    Console.log (b)  }
hello()
```

→ 18
9
20

```
let a=4
var b
if a(){
        var b=9
        let a=9
        console.log(a+b)
        b=a+b
}
function hello(){
 let b=5
 if (a){
      console.log(a+b) — error
      let b=20 ——→ does not get hoisted &
      {                having lexical scope.
else
    {
      let b
    }
    console.log(b)
}hello()
```

→ Print 18
Error at line 12
: Cannot access b. before initialization

Page No.

Lexical is always a part of functional

change in hello function

```
function hello(){
    let b = 5
    if (a){
    {   let b = 20
        console.log(a+b) }
    else
     { let b
     } }
    console.log(b)
}
hello();
```

if we dont declare b here like b = 20, we get error

b does not get hoisted & we can't use b before its declaration

→ 18
24
5

→ add 'let b = 5' before last console.log

→ 18
24
5

If we change this to var
→ 18
24
5

If we add 'console.log(b)' here we get
18
24
5
5

make 'b=5' → 'var b=5'

→ 18
24
5
18

in if block make 'b=9' → 'var b=9'

→ 18
24
5
18

Now replace "var b=9" to "let b=9"

→ 18
24
5
undefined → because b has lexical scope

Now replace "let b=9" to "const b=9"

→ 18
error → because we can't update const again

Do 'var b' to 'const b' [2nd line of code] & make const b=9 to let again

→ error → const has not been initialized

make 'const b' → 'const b=40'

→ 18
24
5
40

RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI
JUHU VERSOVA LINK ROAD, VERSOVA, ANDHERI (WEST), MUMBAI - 53.

make 'let b=9' to 'var b=9'

→ 18 error
24 because b
5 is already
40 declared

if we declare a var inside 'if' cond it will get hoisted to its nearest functional scope

make 'var b=9' to 'let b=9'

→ 18
24
5
40

if we do var b ~~then~~ & inside that let b,
it will work but in reverse
if ~~then~~ let b & inside that var b,
it wont work
var & in lexical scope i can write same var with let because var get hoisted

let cannot be redeclared
∴ it will show error

```
let a=4
let b= 40
let c= 20
if(a) {
    let b = 9
    let a = 9
```

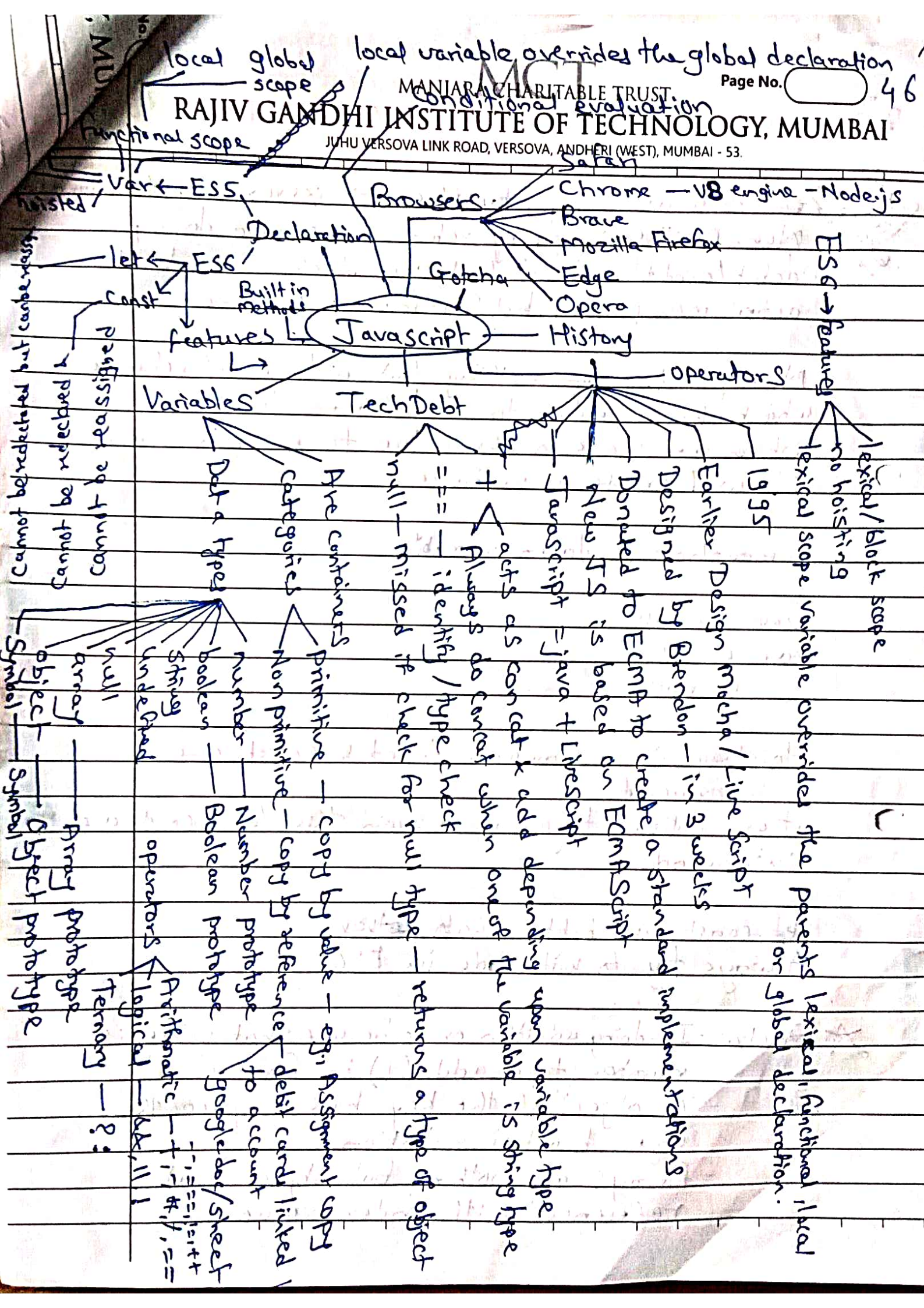
JS variables

Primitive types are copy by value
Non primitive types are copy by reference


```
    console.log (a+b)
    b = a+b
}
function hello(){
  if (a){
        c = 30
        b = 20
        console.log (a+b)
    }else{
          let b
    }
    var b=5
    function inside(){
      console.log ("inside", a+c)
    }
    console.log (b)
    }inside();
}
hello();
consde.log (b,c)
```

→

18
24
5
inside 34
40, 30


local global
scope
local variable overrides the global declaration
Conditional evaluation
Functional scope
Var ← ES5
Declaration
Safari
Browsers
Chrome — V8 engine - Node.js
Brave
mozilla Firefox
Edge
Opera
let ← ES6
Const
Built in methods
Gotcha
Javascript
History
features
ES6 → features
Variables
TechDebt
operators
Cannot be redeclared but can be reassigned
Cannot be redeclared
Cannot be reassigned
Data type
Categories
Pre contained
Symbol
Object
Array
Undefined
String
Boolean
Number
Non primitive
Primitive
Arrays are basically an object
Blocks do not create their own scope
Array — Prototype
Object — Prototype
Symbol — Symbol
Terrary — ? :
Arithmetic operators
Logical operators
Comparison operators
Assignment operators
Bitwise operators
Type conversion
Type checking
Type coercion
Dynamic typing
Loosely typed
Designed by Brendan Eich in 10 days
Designed to create a standard ECMAScript implementations
Earlier Design Mocha / Live Script
Javascript = Java + Livescript
Let is block scoped
Donated to ECMA
ES5
Lexical / Block scope
No hoisting
Lexical scope variable overrides the parents lexical - Functional - local or global declaration.

Read the examples

Primitive → Copy by value → xerox copy of assign
Non-Primitive → copy by reference → google docs, debit/credit cards of bank acc which doesn't hold actual money, they hold reference to the money.

Non primitive are nothing but a kind of reference which holds value to a different location.
When we reassign the value to non primitive, it will create a reference.

Screenshot is taken in mobile.

```
var a = null
var c = a
    c = null
```

Hoisting is a drawback ∴ let & const are introduced.
it will callback to global, to avoid this let & const are introduced.

let, const are better than var
Always try to write code in ES6

Gotcha – If doing addition or to be sure that value is number type add (+) in front of it
- In object if the key has a space then use bracket notation
- Arrays are nothing but object but referenced with index value

CBV applicable for Primitive
CBR applicable for Non-primitive

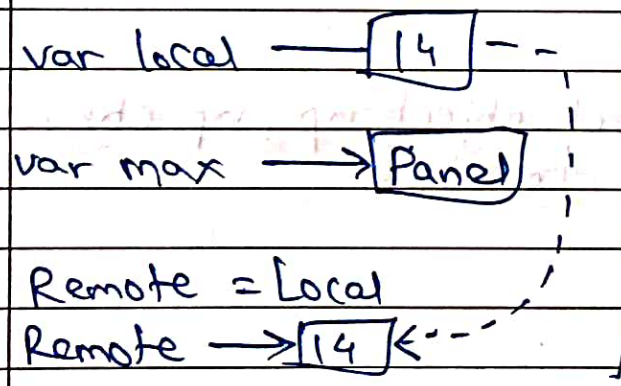
RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI
JUHU VERSOVA LINK ROAD, VERSOVA, ANDHERI (WEST), MUMBAI - 53.

* Conditional evaluation

Falsy values	Truthy values
0, null, undefined, NaN, "", false	→ All non-primitives are truthy → Any primitive which is not falsy

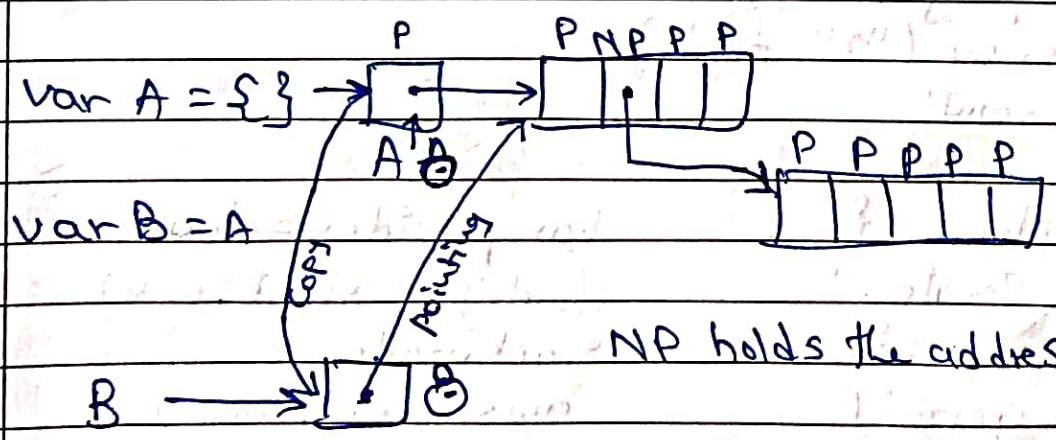
* Built in methods
  - console.log
  - typeof

Copy by value & Copy by Reference



Local = 20
Remote = 14 → already have a copy

whenever i make changes to local it will not change the copied value of remote



NP holds the address of the block

whenever i make changes to A, it will also change value of B OR whenever i make changes to B it will also reflect to A

```
var stack = {
    node : 1
    php : 1
    js : 1
    css : [5,6,7,8,9] }
console.log ("1", stack)
var tech = stack;
console.log ("2", tech)
```

→ copy of stack, print entire object.

```
tech.php = 10
console.log ("3", stack.go)
```

→ 10

// Type of stack is object, object np, np cbr, tech is pointing to stack

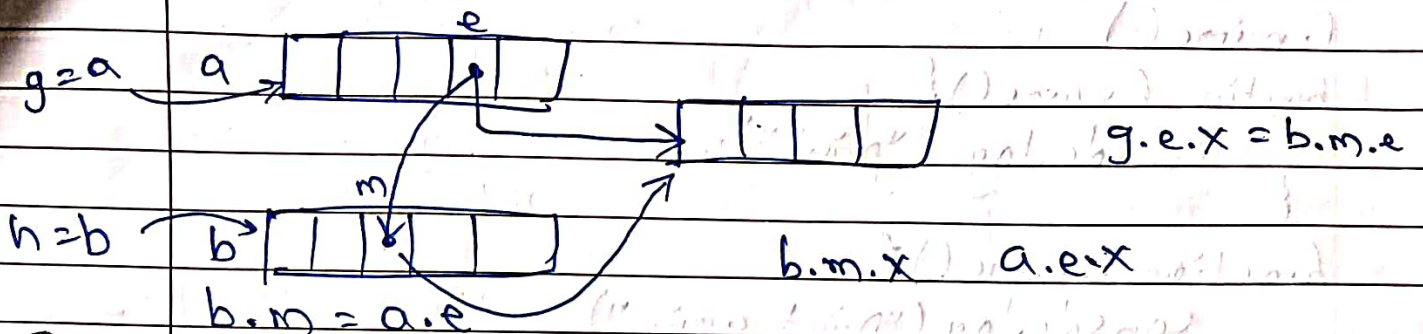
```
tech.node = "34.4"
tech.css[3] = "Hello"
console.log ("1", stack)
console.log ("2", tech)
```

→ Same result

```
var cloud = {
    google : 1
    safari : 1
    opera : 1
    firefox : 1
}
console.log ("3", cloud)

var provider = cloud;
cloud.js = stack.css;
stack.node = {
    aws : cloud.aws }
console.log ("4", provider)
```

cloud.js[1] = "Arrow"
console.log("4", stack, provider)



Javascript Functions

```
function fname(){
    console.log("print")
}
fname();
```

→ print

Specifically in ES5 functions behaves exactly like variable's behaviour

Container, hoisted, scope local & global, Object, override

```
function fname(){
    Console.log("print again")
}
```

Add to prev code

```
fname();
```

→ print again

↑

Object, override

function can be called before it is been declared

Hoisting, override

```
f_name()
function f_name(){
    console.log("print")
}
function f_name(){
    console.log("print again")
}
```

→ print again

```
var f_name = null
!! f_name && f_name();
```

True — This will get called

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

```
Typeof function(){}
```

→ "function"

```
var f_name = 10
!! f_name && (typeof f_name === "function") && f_name();
```

True — False

→ not run the function

It give safety to a function

This way we safeguard

```
function fname(){
  console.log("10");
}
!! fname && (typeof fname === "function") && fname();
```

→ 10

```
fname.hello = "World"        — Add Key
console.log("fname", fname)
fname.key = "change
console.log("fname", fname.hello)
```

```
function outsidecall(){
   Console.log("from outside");
}
outsidecall();
```

```
function outsidecall(){
   console.log("from outside")
   fname()
   function fname(){
   Console.log("from inside");
  }
}

Outside call(); → ~~inside~~ outside inside
```

next → fname() → from outside from inside - 10

function call will look for a global scope.

```
function f_name(){
    console.log("20");
}
function main(){
    function output(){
        console.log("from inside");
        f_name();
        function f_name(){
            console.log("20");
        }
    }
    console.log("from outside");
    f_name();
}
main();
```

→ from outside
20

```
function f_name(){
    console.log("20");
}
function main(){
    function output(){
        console.log("from inside");
        f_name();
        function f_name(){
```

```
        console.log ("200");
    }
}
console.log ("from outside");
f_name();
output();
}
main();
```

→
```
from outside
20
from inside
200
```

```
function fname(){
    console.log ("20");
}
function main(){
    function output (){
        console.log ("from inside");
        fname();
    }
    console.log ("from outside");
    f_name();
    output();
    function fname (){
        console.log ("200");
    }
}
main();
```

→ From outside
200
From inside
200

* In JS, whenever we define a function inside a function, the scope of the function is only limited to that function.
Therefore, we cannot call it from outside the function. It is also known as 'Closure'.

Variable primitives & Non-primitives.

```
var sport = "chess";
function printSport (){
  var sport = "cricket"
    console.log ("sport", sport);
}
 console.log ("outside sport", sport);
printSport();
```

→ outside sport chess
sport cricket

```
var sport = "chess"  //String → copy by value
function printSport (param){
    var sport = param;
    console.log ("sport", sport);
}
```

```
printSport (sport);
console.log ("outside sport", sport);
```

→ Sport Chess
outside sport Chess

```
var sport = "Chess"
function printSport (abc){   //var abc=sport
    abc = "Cricket";
    console.log ("Sport", abc);
}
printSport (sport);
console.log ("outside sport", sport);
```

→ Sport Cricket
outside Sport Chess

```
var sport = "Chess";
var sport2 = "Kabbdi";
function printSport (abc) {
    console.log ("sport" = abc + abc);
}
printSport (sport);
printSport (sport2);
console.log ("outside sport", sport);
console.log ("outside sport", sport2);
```

→ Sport ChessChess
Sport KabbdiKabbdi
outside Sport Chess
outside Sport Kabbdi

```
var Sport = "Chess";
var sport2 = "Kabbdi";
function printSport (sport) {
    console.log ("1. sport", sport);
    function demand (abc) {
        abc = "No Demand"
        Consdle.log ("2. Demand == ", abc);
    }
    demand (Sport);
    Sport = "Football";
    console.log ("3. Value", sport);
}
printSport (sport);
console.log ("4. outside sport", sport);
```

→ 1. Sport Chess
2. Demand == No Demand
3. Value Football
4. outside Sport Chess

In primitives, whenever you pass a variable to a parameter, it copies the value & override the value then change will always remains till the function.

```
var mart={                    //Object → (copy by reference
    Items :"Grocery",
    Checked :1,
    Open :true
}
var mart2 =mart;
mart2. open =false;
console. log (mart);
```

→ {Item :"Grocery", checked:1, open :false}

* In JS except main primitives, everything is an object. It also applies to function as well.

```
function a(){}
a
```

→ f.a(){}

```
a.prototype
```

→ (constructor :f)

```
a. check
```

→ true

```
function hello(somevalue){
    Console.log ("Print"+somevalue);
}
var val =20;
hello(val);
```

→ Print 20

First class citizen
- Function can be assigned to a variable.
- Function can be return from ~~retur~~ another function.
- FRN (Function returns a function)

Declaration & assignment

```
travel && (typeof travel === "function") && travel();
let travel = function(destination){
    return destination;
}
const result = travel("Goa");
result = travel("Mumbai");
```

→ Error because travel is not defined here
∴ cannot access ~~bef~~ before initialization.

```
function(a,b){
    console.log(a+b);
})(10,2)
```
} Cannot call it twice.

→ 12

immediately invoke function expression

Function Properties:
- Normal declaration using function name
- Hoisting
- Override function
- Container (object)
- Scope Local & Global
- Subset of object

RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI
JUHU VERSOVA LINK ROAD, VERSOVA, ANDHERI (WEST), MUMBAI - 53.

```
var hello= function(){
    console.log("printing hello");
}
hello();
hello();
hello();
```

→ printing hello
printing hello
printing hello

Function
- First class citizen — can be assigned to a variable and passed & returned a function converted into a prototype constructor
- Declaration — gets hoisted, override
- expression — assigned, not hoisted, anonymous
- ~~Ho~~ Higher order function — function in a parameter, function returns a function
- Inline function
- Arrow fn (ES6)
- Local / Global scope — Local scope override global scope works exactly same a variable.
- Derived from an object — Acts as a Container, Always a truthy value
- Can be converted into a prototype constructor by using new keyword

Built in Constructor / Object Constructor / Object prototype

- String
- Date
- Array
- Boolean
- Function
- Object
- Math
- Number
- RegEx

Array, Object & String Prototype

```
var fruits = {Mango, Apple, Banana, Guava};
fruits (fruits.length-1) = "Grapes";
Console.log (fruits);
```

→ {Mango, Apple, Banana, Guava, Grapes}

```
var data = { "Name", true, null, {1, "test", {user:
    "Link"}}, {name: "Object", list: [2,3,4,5]}},
  function (param)
    { console.log ("Prints", param)}
},
data()("Some value");
```

→ Prints some value.

Built-in methods that we can operate on datatypes:

- .toString()
- .trim()
- .charAt()
- .split()
- .substr()
- .replace()

```
:(function (callback,next){
  return function (value){   // === final call
    callback (next)(value);
  }
}}{ function (p2){
    return function (n1){
      p2 (n1);
    }
},function (p3) {
    return console.log ("function =>" ,p2) };
}}{"Some value with inner reference"};

;function (callback,next) {
    return function (value){
    callback (next)(value);
  }
}}}{ function (p2) {
    return function (n1){
      p2(n1);
    }
},function (p3) }{
    console.log ("function =>" ,p3)
}}("Pointer p= pointer");
```

→ Function => Some value with inner reference
Function => pointer p= pointer.

Page No.
Date.

Built-in functions & ES6

1] setTimeout()

- Delay the code for specific period of time.

```
const timerFn = function(){
    console.log("Printing after 2 sec");
}
console.log("Running 1");
setTimeout(timerFn, 2000);
Console.log("Running 2");
```

→ Running 1
Running 2
Printing after 2 sec - [This will print after 2 sec]

2] setInterval()

- It's like a timer loop. It runs continuously.

```
console.log("Running 1");
setInterval=function(){
    Console.log("Printing after 5 sec");
},5000);
console.log("Running 2");
```

→ Running 1
Running 2
Printing after 5 sec
Printing after 5 sec
Printing after 5 sec
Printing after 5 sec

} It will continuously running after every 5 sec.

3] parseInt()
- Convert to number type

```
const num = "12.5";
let intNum = parseInt(num);
console.log(intNum);
```
→ 12

4] parseFloat()

```
const num = "12.5";
const price = 25000;
let intNum = parseFloat(num);
console.log(num, intNum, intNum=price, num=price);
```
→ 12.5 12.5 . 312500 312500

5] JSON object
- It helps in converting an object into JSON & JSON into object.
- It is data used outside the program.

```
JSON.stringify({}); } converts object to JSON
```
→ "{}"

```
JSON.parse("{}") } converts JSON to object
```
→ {}

```
var a = JSON.stringify({});
typeof a
```
→ "String"

In JSON ~~stri~~ object, key should be a string.

Page No.
Date

```
var b = JSON.parse("{}");
typeof var b
```
→ "object"

6] Object.~~as~~assign()
- Creates a new object & copies all primitive & non-primitives are still pass by reference.

Trivia & Closing Thoughts

*
```
function a(){
    return function(){
        return function(){
            return function(){
                return function(){
                    return function(){
                        return function(){
                            z = "Final value";
                        }
                    }
                }
            }
        }
    }
}
let result = a()()()()()().z
console.log(result);
```
→ Final value

we can also use arrow before "{}"

Page No.
Date.

* a().b().c().d().e();

→
```
function a(){
    return{
        b:function(){
            return{
                c:function(){
                    return{
                        d:function(){
                            return{
                                e:function(){
                                console.log("a");
                                }
                            }
                        }
                    }
                }
            }
        }
    }
a().b().c().d().e();
```

* a()().b()().c.d().e.z;

→
```
c:{
    d:function(){
        return{
            e:{
                z="final";
            }
        }
    }
}
```

* a(1).b(2).c(3)(4).d(5).final;

JS Built-in methods
- console.log
- typeof
- parseInt
- parseFloat
- setTimeout
- setInterval
- clearInterval
- ~~set~~clearTimeout

Built-in constructor
- Array
  - → push
  - → pop
  - → indexOf
  - → shift
  - → unshift
  - → find
  - → isArray
  - → slice
  - → filter
  - → sort
  - → forEach
  - → map
  - → includes
  - → length

Trivia
- a()a()()()
- a().b().c()
- a()().c().d().e

ES6
- Spread Operator
- Destructuring
- String literals
- Arrow function
- let const