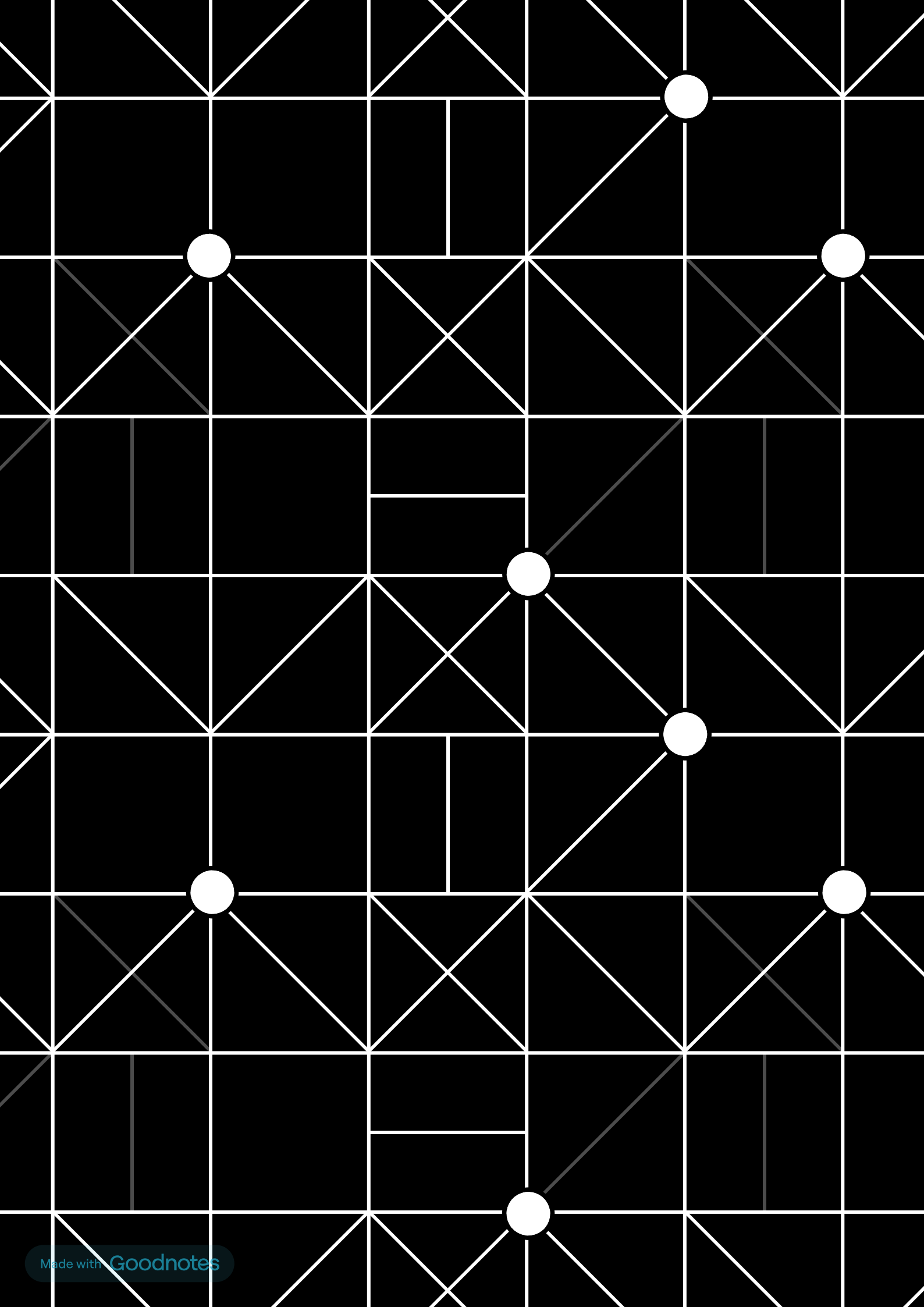
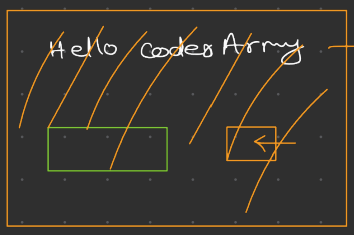


Javascript :- It is a programming language

Calculator
↳ Html
↳ CSS
↳ JS



→ orange
yellow

JS Html
manipulates
↓
change

→ JS is made in 10 days.

1995: Birth of JS

→ Netscape created JS

→ initially called mocha, then renamed Livescript, and finally Javascript.

→ Developed for adding interactivity to webpages within Netscape Navigator.
↳ the dominant web browser of that time.

1996: microsoft and Jscript

→ microsoft developed its version of javascript called Jscript to work with Internet Explorer.

→ Netscape vs. microsoft led to rapid developments and inconsistencies across browsers.

1997: ECMAScript standardization

→ To unify different implementations, Javascript was standardized by ECMA international as ECMAScript.

→ ECMAScript (ES1) was released in June 1997.

→ established the foundation for javascript's consistent behaviour across browsers.

chrome → V8 engine

mozilla → spider monkey

JS code → JS engine (→ V8 engine) code → machine code (0101---)

C++ → compiler → machine code (0101--)

Node.js: [V8 engine + function] → node

Var, let and const

var → outdated, don't use it.

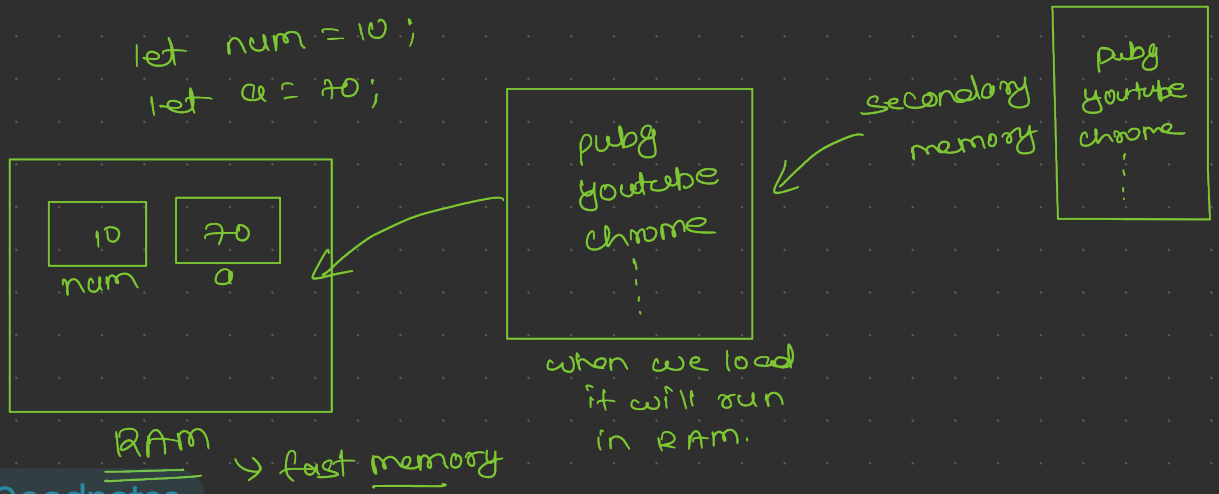
```
let num = 10;
num = 20;
```

✓

```
const str = "mohit";
str = "sohit";
```

→ error.

we can't change the value of a const variable.



Datatypes in JS

Javascript has several built-in data types that are divided into two categories :-

① primitive data types

② Non-primitive data types. — Array, object, functions & many more.

(1) primitive Data types :- These are the basic building blocks in Javascript.

Number :- Represents both integer and floating-point numbers.

```
let account_balance = 786;
console.log(typeof account_balance); → number
```

string :- Represents textual data, enclosed in single(' '), double (" ") or template literals (` `).

```
let str = "Rohit";
console.log(typeof str); → string
```

```
let comment = 'Rohit is a bad boy';
console.log(typeof comment); → string
```

boolean :- Represents a true or false value.

```
let isAdmin = true;
console.log(typeof isAdmin); → boolean
```

undefined :- A variable that has been declared but not assigned a value.

```
let x;
console.log(x); → undefined
console.log(typeof x); → undefined
```

Null :- Represents the intentional absence of any object.

```
let balance = null;
console.log(balance) → null
console.log(typeof balance); → object
```

error in JS.

BigInt :- Represents integers larger than the Number.MAX_SAFE_INTEGER.

```
let bigNumber = 12345678912345678999n;
```

Number :- 64 bit → allocated to memory in JS.

let a = 4893612985120;

↳ suppose it requires 68 bit to convert in Binary.

↓

But we have 64 bits so, some data gets lost.

So, To overcome from this scenario JS introduced BigInt.

let a = 12345678912345678n;

↑ indications of bigint.

Q. what is the largest & smallest Number we can store in 64 bit.

7 ← 1 1 1 — 3bit (largest)

0 ← 0 0 0 — 3bit (smallest)

Now, the question is how can we store -ve Number.

suppose, we have 3 bit

0 1 1 → (3) → largest in 3bits.

↑ sign bit
0 (+ve)
1 (-ve)

1 1 1 → (-3) → smallest in 3bits

110 ⇒ (-2)

010 ⇒ (2)

Here, we gets 2 representation of 0.

0 0 0 +0 = 0

1 0 0 -0 = 0

In 3 bit

$2 \times 2 \times 2$

$2^3 =$ & number / 2 — 4 +ve (0, 1, 2, 3)

(+3, -3)

\ 4 -ve (0, -1, -2, -3)

$(2^{3-1}-1)$ +ve number

$-(2^{3-1}-1)$ -ve number

$\rightarrow (2^{n-1}-1)$ +ve

$-(2^{n-1}-1)$ -ve

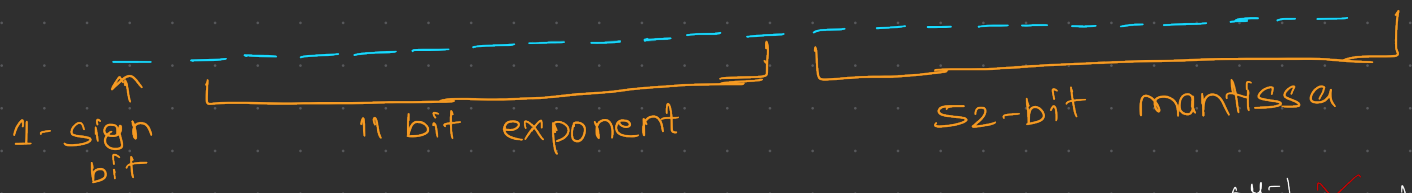
n = no. of bits

4 bits

$\_\ \_\ \_\ \_ \rightarrow 2^3-1 = 7$

$(2^3-1) = -7$

Now, in 64 bit



lets suppose we have to store 42.75

↳ in 64bit → 75.

$2^{64-1}-1$ → largest

$-(2^{64-1}-1)$ → smallest

2	42	
2	21	0
2	10	1
2	5	0
2	2	1
2	1	0
	0	1

42 = 101010.11

$0.75 \times 2 = 1.5$

$0.5 \times 2 = 1.0$

$0.0 \times 2 = 0$

↳ whenever get 0 stop.

101010.11

1.0101011×2^{5} → exponent part.

(2^5 bcz it is in binary).

0 10000000100 01010110000000...

(sign-bit) | exponent part | (mantissa)

↓

It stores the part that comes after decimal.

Now, in exponent part, we can't store direct 5 in binary form.

↓

Here, we add bias

1023 + 5 = 1028 → convert it in binary & then store it in exponent part.

① convert in Binary.

② move decimal to left side, till leftmost 1.

③ After decimal, mentissa part

④ whatever is in the power of 2, that is our exponent part.

⑤ Now, add 1023 to power. and convert it in binary then, store it in exponent part.

Actual Number is getting stored in mentissa part.

1. 1 1 1 1 _ _ _ .... _ _ _ .. 52 times

by default present.

$(2^{53} - 1)$ → largest = Number. MAX_SAFE_INTEGER

$-(2^{53} - 1)$ → smallest = Number. MIN_SAFE_INTEGER

Q. why we have not did $(2^{63} - 1)$.

→ when we do this we are unable to store decimal Numbers.

eg:- 0.75.

0.11

1.1×2^{-1} → exponent. $1023 - 1 = 1022$

↳ mentissa.

0.000010101

1.0101×2^{-5} — exponent. $1023 - 5 = 1018$

mentissa.

Q. why we add 1023. not any other number.

exponent has 11 bits

_ _ _ _ _ _ _ _ _ _ _

exponent (11 bits)

lets suppose we have to store 20.

20 = 10100

0 0 0 0 0 0 1 0 1 0 0

↑ sign bit.

1 0 0 0 0 0 1 0 1 0 0 = -20

2^{11}

-ve ↙ ↘ +ve

$-(2^{10} - 1)$ $\quad$ $2^{10} - 1$

-1023 $\quad$ 1023

Now, they decided to drop the logic of sign bit.

so, $-1023 + 1023 = 0$.

$1023 + 1023 = 2046$

↑

They have decided to convert all these in (+ve).

so, we add 1023 to the exponent. ←

(2.) Non-primitive Data types

Array :- used to store ordered collections of values.

```
let arr = [10, 20, 30.2, "rohit", "mohit"];
console.log(arr); → [10, 20, 30.2, 'rohit', 'mohit']
console.log(typeof arr); → object.
```

object :- Represents collections of properties, where each property is a key-value pair.

```
let obj = {
    user_name : "Rohit",
    account_number: 31242314213
    balance : 420
}
console.log(typeof obj); → object.
```

Function :- A block of code designed to perform a particular task

```
let fun = function() {
        console ("Hello Coder Army");
    }
console.log(typeof fun); → function.
```

Type conversions

String to Number.

```
let account_balance = "100";
let num = Number(account_balance);
console.log(typeof account_balance); → string
Console.log(typeof num); → number

let account = "100xs";
console.log(Number(account)); → NaN.
```

Boolean to Number

```
let x = true;
console.log(Number(x)); → 1.
```

Null to Number

```
let x1 = null;
console.log(Number(x1)); → 0.
```

Undefined to Number

```
let x2;
console.log(Number(x2)); → NaN
```

Number to string

```
let ab = 20;
console.log(String(ab)); → '20'
```

Boolean to string

```
let ax = true;
console.log(String(ax)); → 'true'
```

String to Boolean

```
let abc = "str";
Console.log(Boolean(abc)); → true
console.log(Boolean(" ")); → false
console.log(Boolean(" ")); → true
```

Operators

1. Arithmetic Operators

Used to perform mathematical calculations.

Operator	Description	Example
+	Addition	5+3 → 8
-	Subtraction	5-3 → 2
*	multiplication	5*3 → 15
/	Division	10/2 → 5
%	modulus (remainder)	10%3 → 1
++	Increment	x++ or ++x
--	Decrement	x-- or --x
**	Exponentiation	2**3 → 8

Divide multiply :- L to R
Add subtract :- L to R
} → order of priority

```
Console.log(6*3+18/6-9); → baat mar dega company aise likhe ton.
Console.log(((6*(3+18))/6)-9)); → Better way to write.
```

```
let sum = 20;
Console.log(sum++); → 20 (post increment)
Console.log(sum--); → 21 (post decrement)
```

```
let num = 23;
++num; (pre increment)
Console.log(num); → 24
Console.log(--num); → 23 (pre decrement)
```

(2.) Assignment operator.

Used to assign values to variables.

operator	Description	Example
=	Assign	x = 5
+=	Add and assign	x += 3 → x = x + 3
-=	Subtract and assign	x -= 3 → x = x - 3
*=	multiply and assign	x *= 4 → x = x * 4
/=	Divide and assign	x /= 2 → x = x / 2
%=	modulus and assign	x %= 3 → x = x % 3
**=	Exponentiation & assign	x **= 2 → x = x ** 2

(3.) Comparison operator

```
let a1 = 10;
let a2 = 20;
console.log(a1 == a2)  false
console.log(a1 > a2)   false
console.log(a1 < a2)   true
console.log(a1 >= a2)  false
console.log(a1 <= a2)  true
```

```
let num = 10;
let str = "10";
```
here, string gets converted into number

```
console.log(num == str);  → true
console.log(num == "20");  → false
str = "30x";
console.log(a1 < str);  → false
```
str is converted in (NaN).

(===) first check type if it's same then compare the values inside it.

```
let a1 = 10;
let str = "10";
console.log(a1 === str);  → false
let a2 = 10;
console.log(a1 === a2);  → true
```

null == undefined → true.

null === undefined → false.

null is only equivalent to undefined.

console.log (null == 0); → false

in below cases null gets converted to 0.

console.log (null < 0) 0<0 false

console.log (null > 0) 0>0 false.

console.log (null <= 0) 0<=0 - true

console.log (null >= 0) 0>=0 true

NaN == NaN
↓
false.

undefined comparison :-

console.log (undefined == 0); false, bcz it's only equivalent to null.

console.log (undefined < 0); NaN<0 (false)

console.log (undefined > 0); NaN>0 (false)

console.log (undefind <= 0); NaN<=0 (false)

console.log (undefined >= 0); NaN>=0 (false)

```
let str3 = "mohit";
let str4 = "mohit";
console.log(Number(str3) == Number(str4));
```

NaN == NaN → false.

```
let abc1 = 123;
let abc2 = "123";
let abc3 = 123;
console.log(abc1 == abc2 == abc3) → false.
```

Here, first is compare abc1 == abc2 which gives true, then it compare true == abc3. it gives false.

```
abc3 = true;
console.log(abc1 == abc2 == abc3); → true.
```

(4.) Logical operator

operator	Description	Syntax example	Result
&&	logical AND:- Returns true if both operands are true. otherwise false.	true && true 5>3 && 3<2 false && false	true false false.
||	logical OR :- Returns false if both operand are false. Otherwise true.	true || false false || true false || false true || true	true true false true.
!	logical NOT:- Inverts the boolean value of the operand.	!true !false	false true

(5.) Bitwise operator

operator	Name	Description	Example
&	Bitwise AND	Performs a bitwise AND operation on each pair of bits. Result is 1 if both bits are 1.	5&3 → 0101 & 0011 = 0001 → 1.
|	Bitwise OR	Performs a bitwise OR operation on each pair of bits. Result is 1 if atleast one bit is 1.	5|3 → 0101 | 0011 = 0111 = 7.
^	Bitwise XOR	Performs a bitwise XOR. Result is 1 if bits are different.	5^3 → 0101 ^ 0011 = 0110 = 6
~	Bitwise NOT	Inverts all the bits (1 to 0, 0 to 1)	~5 → ~0101 → 1010 ⇒ 10
<<	left shift	shifts bits to the left by the specified no. of positions.	5<<1 → 0101 becomes 1010 → 10.
>>	Right shift	shifts bits to the right, maintaining the sign.	5>>1 → 0101 becomes 0010 → 2.

memory in JS

Primitive data type :- Immutable — value can't be changed

Non-primitive data type :- mutable — value can be changed.

```
let obj1 = {
        id : 20,
        naming: "rohit"
    }
let obj2 = obj1;
console.log (obj 2);   { id : 20, naming : "rohit" }
obj2. id = 30;
console.log(obj1);
console.log (obj2);   > { id : 30, naming : 'rohit' }
```

↳ Both obj1 & obj 2 is modified.

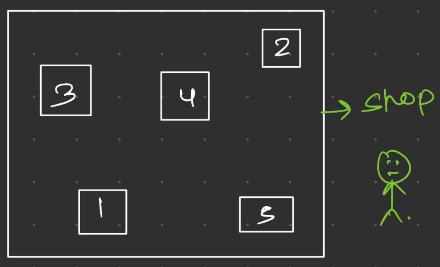
stack and heap memory

Suppose we have 5 packets of sugar. How can we store it in the

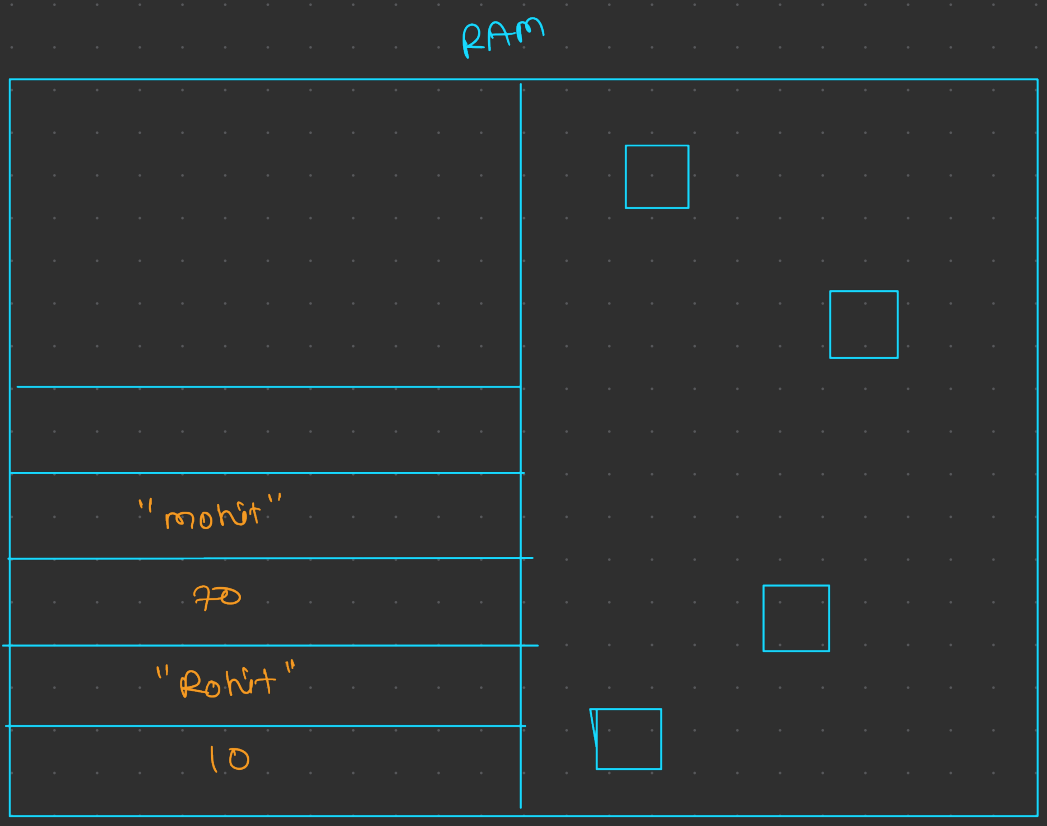
→ packets of sugar.

① 

Here, we have kept it side by side.

② 

Here, we have kept randomly anywhere, where the space is free.



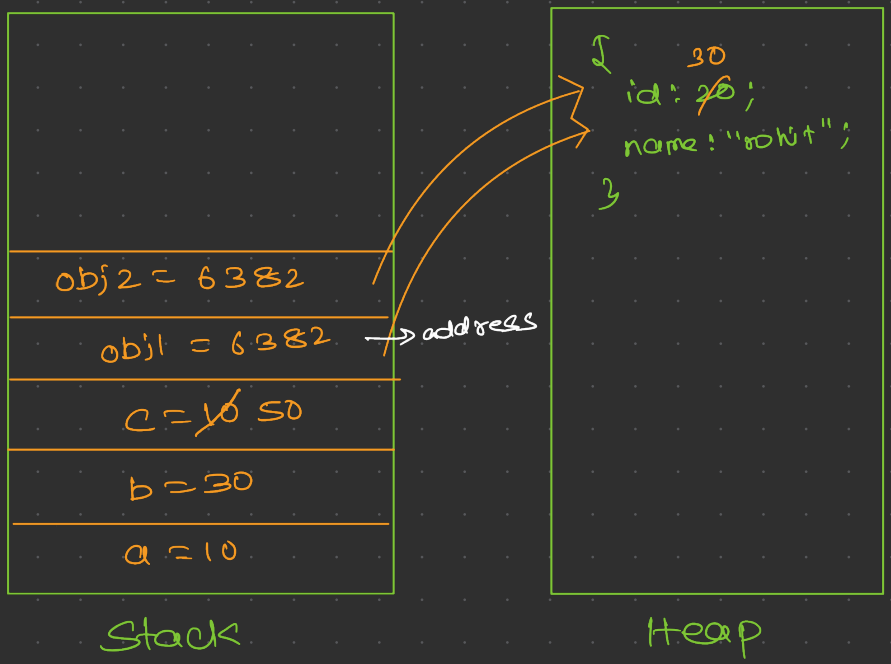

```
let a = 10;
let b = 30;
let c = a;
c = 50
```

(Call by value)

↑ Here changes affect only in c. not in a.

```
let obj1 = {
    id : 20;
    name : "Rohit"
}
let obj2 = obj1;
obj2.id = 30;
```

→ Here, it changes id to 30. As obj1 and obj2 is pointing to the same location. Then the values that changes got affected in both obj1 and obj2.



Why we need address?

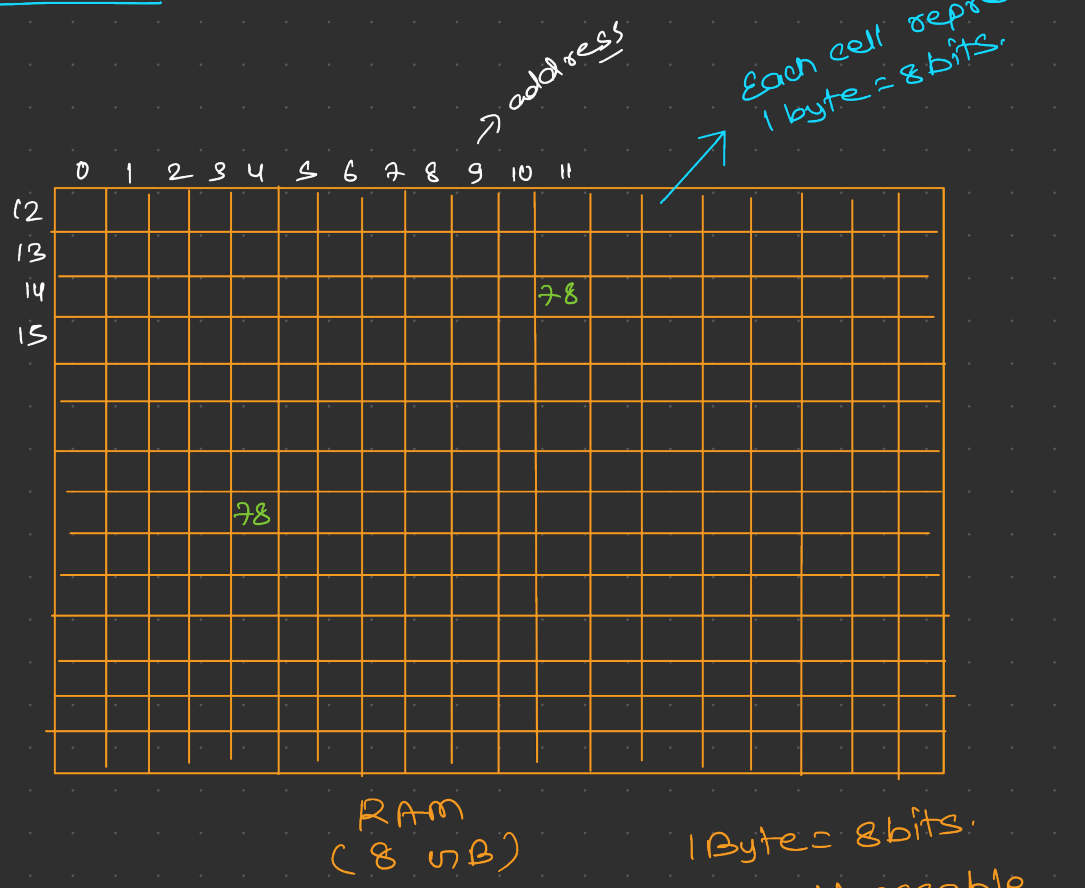
1 GB = 1024 MB
= 2^{10} MB.

1 MB = 2^{10} KB.

1 KB = 2^{10} Bytes

1 GB = 2^{30} bytes

8 GB = 2^{33} bytes



RAM
(8 GB)

1 Byte = 8 bits.

Byte addressable
↓
each byte gets an address.

Scenario 1:- We want to access 78.

If we don't know the address then we have to search one by one.

↓

It takes a lot of time.

↓

So, To overcome from this scenario, we need address of that variable.

Scenario 2:- Now we have 78 at two places.

↓

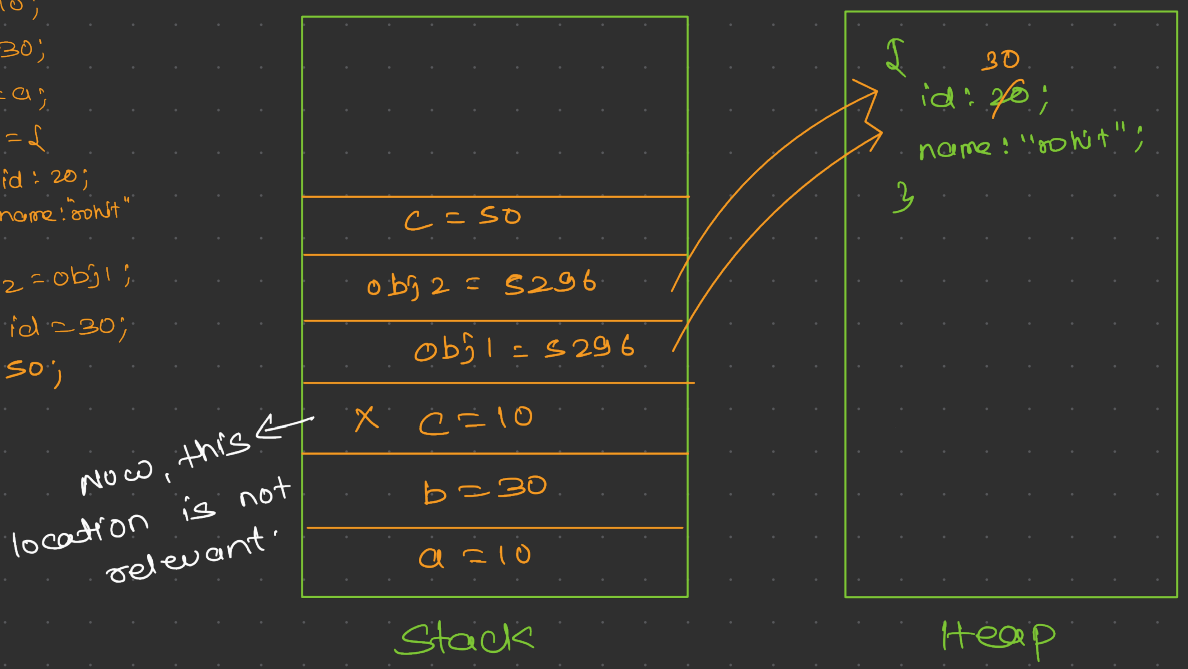
We get confused that which 78 we have to access.

↓

To overcome from this situation we need address.

Ques:- why primitive data type is immutable ?

```
let a = 10;
let b = 30;
let c = a;
let obj1 = {
    id : 20;
    name : "Rohit"
}
let obj2 = obj1;
obj2.id = 30;
c = 50;
```



In JS, when we try to change or modify the value of that variable. It creates new memory location for that variable.

↑
This is why it is immutable.

what is the reason behind doing this?

Suppose c takes memory of 8 bytes.

Now, we want to store

c = "Rohit Bhaiya Zindabaad";

↑
It is taking 20 bytes.

So, we are unable to store it at the old location as there we have only 8 bytes.

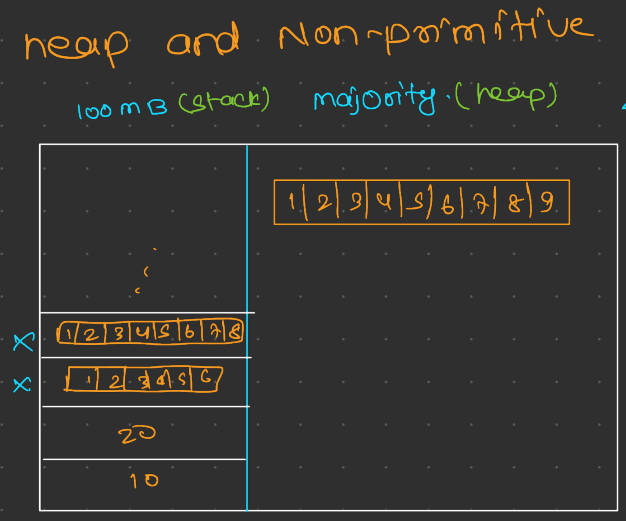
↑
This is why we give it a new location.

This is not in C++ / Java. because if we have variable of int data type. Then, we can change the value of that variable to integer only.

↓
So, we don't need this type of memory management there.

Here, Question arises that why we haven't store primitive data type in heap and Non-primitive data type in stack

Ans:-



Stack gets very less memory in RAM.

Now, if we store 10 elements in array in stack.

and we added some elements to it.

↑

So, it gets new memory location in stack.

Then, stack gets full quickly.

So, we store Non primitive data type in heap.

and primitive in stack.

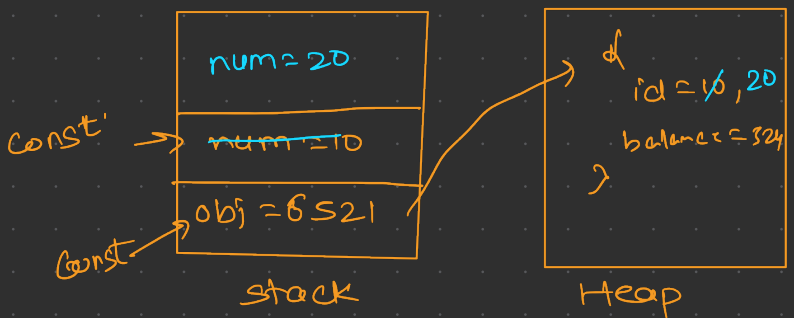
↑

it takes a small amount of memory.

```
// primitive data type
const num = 10;
num = 20;
console.log(num);
```

→ Here, it gives error bcz we can't change value of a const variable.

```
// Non - primitive
const obj = {
    id: 10,
    balance: 234
}
obj.id = 20;
console.log(obj);
```



Here, the address of obj is not changing. so, it is not giving error.

String in JS

→ ways to declare string.

```
let singlequote = 'Hello, world';
let doublequote = "Hello, world";
let backtricks = `Hello, world`;

let price = 80;
console.log(`Price of the tomato is ${price}`);
```

↓

Price of the tomato is 80

→ String concatenation

```
let s1 = "hello";
let s2 = " world";
let s3 = s1 + s2;
console.log(s3); → "hello world"
```

→ length of the string (str.length)

```
console.log(s1.length); → 5
console.log(s2.length); → 6
console.log(s3.length); → 11
```

→ Access character

```
let special = "rohit";
console.log(special[0]); → 'r'
console.log(special.charAt(3)); → 'i'
```

→ toLowercase(), toUppercase() :- it returns a new string.

```
console.log(special.toLowercase()); → 'rohit'
console.log(special.toUppercase()); → 'ROHIT'
```

→ Searching in strings

- indexOf(substring) :- returns the index of first character of first occurrence of a substring if it exists, otherwise returns -1.

```
let hero = "Hello coder coder Army"
console.log(hero.indexOf("coder")); → 6.
```

- lastIndexOf(substring) :- returns the first index of last occurrence of a substring if it exists, otherwise returns -1.

```
console.log(hero.lastIndexOf("coder")); → 12
```

- includes(substring) :- check if a substring exists.

```
console.log(hero.includes("coder")); → true
```

→ Extracting substrings

- slice(start, end) :- Extracts part of a string from start index to end. end is not included.

```
let newstr = "HelloDon";
console.log(newstr.slice(0,3)); → 'Hel'
```

- substring(start, end) :- similar to slice but doesn't accept negative indexes.

```
  0 1 2 3 4 5 6 7
  H e l l o D o n
 -7-6-5-4-3-2-1 0
```

```
console.log(newstr.substring(0,3)); → 'Hel'
console.log(newstr.slice(-6,5));
console.log(newstr.slice(-2,4));
```

↓

it doesn't work as starting index is coming after ending index. so, it's not possible.

→ Replacing contents

- replace(oldSubstring, newsubstring) :- Replaces the first match.

```
let str10 = "hello world, world acaha hai";
console.log(str10.replace("world", "duniyaa"));
```

↓

"hello duniyaa, world achha hai"

- replaceAll(oldsubstring, newsubstring) :- Replaces all the matches.

```
console.log(str10.replaceAll("world", "duniyaa"));
```

↓

'hello duniyaa, duniyaa achha hai'

→ Splitting strings

- split(delimeter) :- splits a string into array based on a delimeter.

```
let str = "money! honey! sunny! funny!";
console.log(str.split("! "));
```

↓

['money', 'honey', 'Sunny', 'funny']

→ Trimming

- trim() :- removes whitespaces from both ends.
- trimStart() | trimEnd() :- removes whitespaces from the start or the end.

New way to create string

```
let lateststring = new String("hello coder Army");
console.log(typeof lateststring);
```

→ object

Now, this string will takes place in heap instead of stack.

Number in JS

```
let num1 = 231;
let num2 = new Number(231);
console.log(typeof num2); → object
let num3 = new Number(231);
console.log(num1 == num2) → true
```

Here, datatype of num2 and num1 is not same. So, JS firstly convert num2 in number and then compare it with num1.

```
console.log(num2 == num3) → false.
```

Here, datatype of num2 and num3 is same i.e object. And we know that the object is getting memory in heap. So, it will compare address and the address is not same.

- `toFixed()`:- Formats a number with a fixed no. of decimal places.

```
let num = 231.689;
console.log(num.toFixed(1)); → 231.6
```

- `toPrecision()`:- Formats a number to specified length

```
console.log(num.toPrecision(5)); → 231.69
```

- `toString()`:- converts a number to string.

```
let num = 42;
console.log(num.toString()); → '42'
```

- `toExponential()`:- Converts a number into its exponential notation and returns a string representation

```
let num = 12345.6789;
console.log(num.toExponential(2)); → "1.23e+4"
```

Math object

JS provides a Math object for advanced mathematical Calculations:-

```
console.log(Math.E); → 2.71
Console.log(Math.LN10); → 1
Console.log(Math.PI); → 3.14....
```

Math.round() → Rounds to the nearest integer.

```
console.log(Math.round(4.6)); → 5
```

Math.floor() → rounds down

```
console.log(Math.floor(4.9)); → 4
```

Math.ceil() → rounds up

```
console.log(Math.ceil(4.1)); → 5.
```

```
console.log(Math.random()); //it generate value between
0<=value<1
console.log(Math.random() * 10); // 0<=value<=9
console.log(Math.floor(Math.random() * 10));//it generate 0,1,2,
3,4,5,6,7,8,9
console.log(Math.floor(Math.random() * 10) + 1); // it generate
from 1,2,3,4,5,6,7,8,9,10

console.log(Math.floor(Math.random() * 10) + 11); // 0-9 + 11

// generalised formula
// min = 40 , max = 50;
// console.log(Math.floor(Math.random() * (max-min+1) + min));

console.log(Math.floor(Math.random() * (40-30+1)) + 30);
```

Array in JS

Arrays are versatile in JS, a dynamic data structure that store ordered collections of values. These values can be of any type, including numbers, strings, objects and even other arrays.

```
let arr = [2, 3, 4, 1, 8, 9, "rohit", true];
```

length :- returns the number of elements in the array.

```
let arr = [1,2,3];
console.log(arr.length); → 3
```

accessing array elements

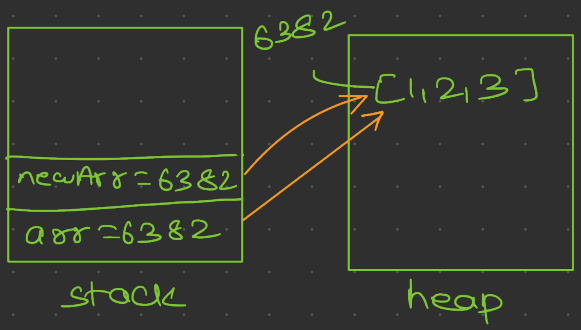
```
console.log(arr[0]); → 1
console.log(arr.at(1)); → 2
```

↳ it also takes -ve indexes.

```
console.log(arr.at(-1)); → at returns last element (3).
```

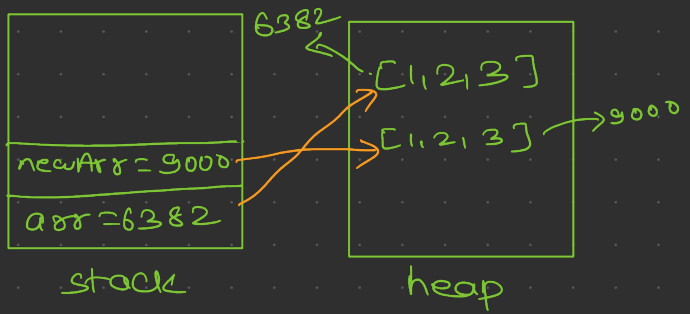
Copying arrays.

```
const arr = [1,2,3];
const newArr = arr; → here, reference of arr is getting copied to newArr.
console.log(newArr == arr); → true
```



Now, I want newArr to points at different address.

```
const newArr = structuredClone(arr); (! important)
Console.log(newArr == arr); → false.
```



push() :- Adds an element at the end

```
const arr = [10,20,30];
arr.push(40);
arr.push(50);
console.log(arr); → [10,20,30,40,50]
```

pop() :- removes the last element.

```
const arr = [10,20,30];
arr.pop();
console.log(arr); → [10,20]
```

unshift() :- Adds an element at the beginning.

```
const arr = [10,20,30];
arr.unshift(50);
console.log(arr); → [50,10,20,30]
```

shift() :- Removes the first element

```
const arr = [10,20,30];
arr.shift();
console.log(arr); → [20,30]
```

delete :- delete element of a particular index, it has a disadvantage that it deletes the element but the space is reserved for that element.

try not to use it.

```
const arr = [10,20,30,40];
delete arr[0];
console.log(arr); → [<1 empty item>,20,30,40]
console.log(arr[0]); → undefined.
```

indexOf() :- returns the index of first occurrence.

```
const arr = [2,4,6,8,10,8];
console.log(arr.indexOf(8)); → 3
```

lastIndexOf() :- returns the last index of the element.

```
console.log(arr.lastIndexOf(8)); → 5.
```

includes() :- checks if an element exists.

```
const arr = [2,4,6,8,10];
console.log(arr.includes(10)); → true.
```

returns true if element exists, otherwise false

slice() :- Extracts a portion of the array without modifying it.

```
const arr = [2,4,6,8,10,12,14];
console.log(arr.splice(2,5)); → [6,8,10]
```

↓

returns an array from index 2 to 4.
5th index is not included

splice() :- method in JS is a powerful array method used for modifying an array by adding, removing or replacing elements.

```
const arr = [2,4,6,8,10,12,14];
console.log(arr.splice(2,4)); → [6,8,10,12]
```

↓

returns an array from index 2, and count 5 elements from there and changes the original array by removing these 5 elements.

```
console.log(arr); → [2,4,14]
```

splice(starting index, total-element-delete, add value)

```
console.log(arr);
arr.splice(2,0,"saurav",450,true);
console.log(arr); → [2,4,"saurav",450,true,14]
```

toString() :- converts an array to a string of (comma separated) array values.

```
const arr = [2,4,6,8,10];
console.log(arr.toString()); → 2,4,6,8,10
console.log(typeof arr.toString()); → string
```

join() :- joins all array elements into a string.

```
const arr = [2,4,6,8,10];
console.log(arr.join("*")); → 2*4*6*8*10
console.log(arr.join(" ")); → 2 4 6 8 10
```

concat():- This method creates a new Array by merging (concatenating) existing arrays.

```
let arr1 = [2,35,6,11];
let arr2 = [5,12,19,20];
let arr3 = arr1.concat(arr2);
console.log(arr3); → [2,35,6,11,5,12,19,20]
let arr4 = [23,432,1123,31];
let arr5 = arr1.concat(arr2,arr4);
console.log(arr5); → [2,35,6,11,5,12,19,20,
                      23,432,1123,31]
```

when we push

```
arr1.push(arr4); → it behaves like 2d array.
console.log(arr1); → [2,35,6,11,[23,432,1123,31]]
```

flat():- This method creates a new array with sub-array elements concatenated to a specific depth.

```
// convert 2d into 1d.
let arr = [[1,2,3],[4,5,6],[7,8,9]];
let newArr = arr.flat();
console.log(newArr); → [1,2,3,4,5,6,7,8,9]

// convert 3d into 1d
let arr = [1,2,[3,4,[5,6]],7,8];
let newArr = arr.flat();
console.log(newArr); → [1,2,[3,4,5,6],7,8]

let result = arr.flat(2);
console.log(result); → [1,2,3,4,5,6,7,8]
```

arr.flat(Infinity)

↓

By using this we can convert any (3d,4d,5d) array into 1d.

isArray():- returns true if it is array, otherwise false.

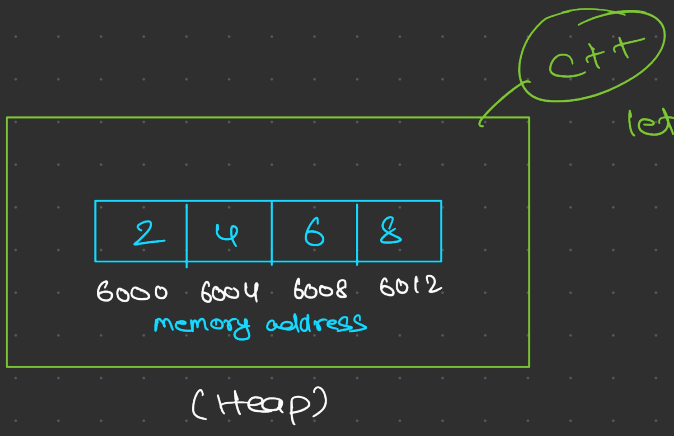
```
let abc = [1,2,3];
console.log(Array.isArray(abc)); → true
```

other way to create Array

Not recommendable

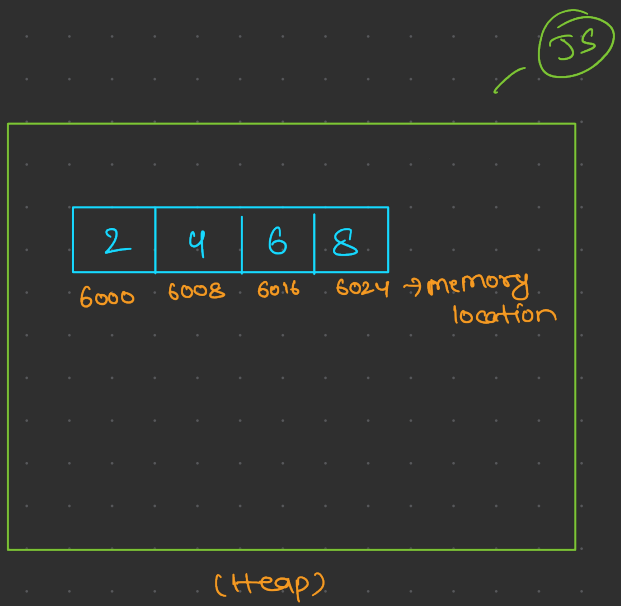
```
let ac = new Array(2,3,10,8);
console.log(ac); → [2,3,10,8]
```

```
let arr = new Array(2); → when we give single value, it indicates to the size of the array.
console.log(arr);
```



let a = [2,4,6,8]
0 1 2 3

Here, if we want to modify the value of 2 then instead of that we only modify integer elements.
↓
the memory size is same
so, In C++, Array takes contiguous memory.



let a = [2,4,6,8]

For storing 2 it takes address from (6000 - 6007).

Now, I want to store "Hello Coder Army" instead of 2.
↓
But it takes more than 8 byte memory.
↓
so, if we go with C++ memory management I will loose some data.
↓
For this reason, In JS array is not taking contiguous memory.

Date in JS

In JS, the Date object is used to work with dates and times. It provides various methods to create, manipulate, and formate dates

```
const d = new Date();
console.log(d); → 2024-11-29T15:34:51.729Z
                  (international time zone).

console.log(d.toDateString()); → Fri Nov 29 2024

console.log(d.toString());
            ↓
  Fri Nov 29 2024 21:06:20 GMT+0530 (Indian
                                  Standard time)

console.log(d.toISOString());
            ↳ 2024-11-29T15:36:49.6592

console.log(typeof d); → object
```

Q. From where it is bringing date for us?

⇒ System clock.

JS is calculating time and date in milliseconds from Jan-1-1970 12:00:00.

```
Const d1 = new Date(1000);
Console.log(d1); → 1970-01-01T00:00:01.0002
```

It calculate 1000 ms from 01-01-1970 12:00:00 and return us a date.

```
const d2 = new Date(2000);
console.log = (d2); → 1970-01-01T 00:00:02.0002
```

Common methods

getFullYear():- Returns the year.

getMonth():- Returns the month (0-11)

getDate():- Returns the day of the month (1-31)

getDay():- Returns the day of the week (0-6, where 0 is sunday)

getHours():- Returns the hour (0-23)

getMinutes():- Returns the minutes (0-59)

getSeconds():- Returns the seconds (0-59)

getMilliseconds():- Returns the milliseconds (0-999)

```
const d = new Date();
console.log(d.getFullYear()); → 2024
console.log(d.getMonth()); → 10 (november)
console.log(d.getDate()); → 29
```

Q. why we have followed the milliseconds approach?

Ans:- milliseconds → Date — 2024-11-29
→ month
→ year.

online ticket book.

problem is we have 1 ticket, and there is 2 person who tried to book tickets at same time.

↓

If we have followed date approach then, we are unable to get result of who booked first bc2 both have booked at same day.

↓

So, we have followed milliseconds approach, as every minute time values matters.

↓

- milliseconds is very helpful in getting minute differences blw the time.
- mathematical calculate is easy in milliseconds in comparison of date

let's take a example.

suppost 2 person booked a ticket.

1st person - 23815679 ms

2nd person - 23815678 ms

and we have only 1 ticket left.

Here, on comparing time for 1st person and 2nd person.

we can see that 2nd person has booked the ticked 1ms earlier than 1st person.

milliseconds is known as timestamps in js.

```
console.log(d.getTime()); → 1732895732567
```

↓

It returns milliseconds from 01-01-1970 12:00:00.

```
const now = Date.now();
console.log(now);
```
→ it returns milliseconds from 01-01-1970 12:00:00

we can create custom Date

```
const d = new Date("2022-10-20");
console.log(d);
```
→ 2022-10-20T00:00:00.0002

when we give date in string format, 1-based counting.

```
const d = new Date("2022-10-20T10:10:12");
console.log(d);
```
→ 2022-10-20T10:10:12.0002.

```
const date = new Date(2024, 4, 28);
console.log(date);
```
→ 2024-05-27T18:30:00.0002

↓

Here, 0 based counting bcz we have entered in number format

```
console.log(date.toDateString());
```
↳ Tue may 28 2024

(year, month, date, hours, minutes, seconds, milliseconds)

↓

we can give all these values

But first 2 value is compulsory, if we give single value JS will think that it is milliseond

Setting date components

```
const d = new Date();
d.setDate(20);
d.setFullYear(2021);
d.setMonth(3);
console.log(d.toString());
```
Here, also (0)based counting for month

↓

Tue Apr 20 2021 21:41:24 GMT+0530 (India Standard Time)

object in js

An object is a collection of key-value pairs, where the keys are called properties (or methods if their values are function). objects are one of the fundamental building blocks of JS and are used to represent real world entities, store data and organize code in a structured manner.

how to create objects

```
const obj = {
    name: "Bhupendar Jogi",
    account_balance : 150,
    gender: "male",
    age : 30
}
console.log(obj);
```

→ { name: 'bhupendar jogi', account_balance: 150, gender: 'male', age: 30 }

we can write our keys in the form of string also

```
const obj = {
    "name": "Bhupendar Jogi",
    "account_balance" : 150,
    "gender": "male",
    "age" : 30
}
console.log(obj);
```

→ { name: 'bhupendar jogi', account_balance: 150, gender: age: 30 }

→ In Backend, keys are always going to be stored in the form of string.

→ we can write the keys with and without strings.

→ we can't use space in keys unless we are declaring it as a string.

account number : 10500 ✗

"account number" : 10500 ✓

How to access object properties

(1.) DOT Notation.

```
console.log( obj.name );   → 'Bhupendar Jogi'
console.log (obj.age) ;    → 30
console.log (obj.gender) ; → 'male'
```

(2.) Bracket Notation

```
console.log (obj["gender"]); → 'male'
console.log (obj["name"]); → 'Bhupendar Jogi'
```

→ when we are accessing the properties using bracket Notation, then must write keys in string.

→ when we have space in the key's name, then we can only access it using bracket Notation not by dot notation.

```
obj.account number  ✗
obj["account_number"]  ✓
```

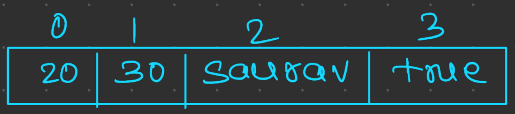
→ If we have number in the keys, then we can access it without(" ") in Bracket Notation.

```
const obj = {
    0: 1,
    1: 2,
    2: 3,
    3: 10
}
```

```
console.log (obj[0]); → 1
console.log (obj[1]); → 2
console.log (obj[2]); → 3
```

→ typeof array is object because javascript is storing it in key:value pairs internally.

```
const arr = [
    0: 20,
    1: 30,
    2: "saurav",
    3: true
]
```



→ we can name our keys undefined and null too.
↳ this is why, at the end it is stored as string.

```
const obj = {
    name: "Bhupendar jogi",
    gender: "male",
    age: 30,
    undefined : 50,
    null : "saurav"
}
```

```
console.log(obj.undefined)
console.log(obj[undefined]);
console.log(obj["undefined"]);
```
} 50

```
console.log(obj.null)
console.log(obj[null]);
console.log(obj["null"]);
```
} 'saurav'

more ways to create object

using the object constructor

```
const person = new Object();
```

we can add, delete, modify the property later.

add

```
person.name = "saurav";
Person.age = 80;
person.gender = "male";
console.log(person); → { name: 'saurav', age: 80, gender: 'male'}
```

delete

```
delete person.age;
console.log(person); → { name: 'saurav', gender: 'male'}
```

modify/update

```
person.name = "gaurav";
```

→ if there exists a property named name, then it will modify the value. , if it doesn't exist then it will add a property named name.

```
console.log(person); → { name: 'gaurav', gender: 'male'}
```

By creating a class (oops progamming)

```
class people {
    constructor (name, age, gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
}

let person1 = new people("rohit", 20, "male");
let person = new people("mohit", 30, "male");
let person = new people("pinki", 19, "female");

console.log(person1); → { name: 'Rohit', age: 20, gender: 'male' }
console.log(person2); → { name: 'mohit', age: 30, gender: 'male' }
console.log(person3); → { name: 'pinki', age: 19, gender: 'female' }
```

this is pointing to its instance.

here, this is referring to person1

here, this is referring to person2

here, this is referring to person3

→ The Benefits of this approach is that we can create multiple instances of it, if we are declaring using previous methods then for every instance we have to write same code.

Common methods for objects

```
let obj = {
    name: "Saurav",
    age: 30,
    account_balance: 150,
    gender: "male"
}
```

Object.keys() :- Returns an array of property names

```
let arr = Object.keys(obj);
console.log(arr); → ['name', 'age', 'account_balance', 'gender']
```

Object.values() :- Returns an array of property values.

```
let values = Object.values(obj);
console.log(values);
        ↓
['saurav', 30, 150, 'male']
```

Object.entries() :- Returns an array of key-value pairs.

```
let arr2 = Object.entries(obj);
console.log(arr2);
```

```
[
  ['name', 'saurav'],
  ['age', 30],
  ['account_balance', 150],
  ['gender', 'male']
]
```

Object.assign() :- Copies properties from one object to another.

```
const obj1 = {
    a: 1,
    b: 2
}
const obj2 = {
    c: 3,
    d: 4
}
```

```
const obj3 = Object.assign(obj1, obj2);
console.log(obj3);
        ↓
{a: 1, b: 2, c: 3, d: 4}
```

But this changed the obj1 also

```
console.log(obj1);
        ↓
{a: 1, b: 2, c: 3, d: 4}
```

→ It happens because in object.assign(), the first parameter it takes is that where it want to stores the object.

→ so, we always use empty object {}.

```
const obj4 = Object.assign({}, obj1, obj2);
console.log(obj4); → {a:1, b:2, c:3, d:4}
console.log(obj1); → {a:1, b:2}
```

→ we can copies the properties of as many as object we want.

```
const obj6 = Object.assign({}, obj1, obj2, obj3, ...)
```

→ If we created an object using object.assign() and then later we changed the property or values of the object, then it doesn't reflect the object from where I have copied the properties.

```
const obj1 = {
  a:1,
  b:2
}

const obj2 = {
  c:3,
  d:4
}

const obj3 = Object.assign({}, obj1, obj2);

obj3.a = 10;
console.log(obj3); → {a:10, b:2, c:3, d:4}
console.log(obj1); → {a:1, b:2}
```

Spread in objects :- The spread operator creates a shallow copy of an object. It opens up all the key:value pairs of the objects & store it.

```
const obj4 = {...obj1, ...obj2};
console.log(obj4); → {a:1, b:2, c:3, d:4}
```

Object.freeze() and Object.seal()

↓ completely freezes an object, making it immutable.

↓ seals an object, preventing the addition or removal of properties but allowing the modification of existing properties' value.

eg:-

```
let obj = {
    name: "Alice",
    age: 25
};
Object.freeze(obj);
obj.age = 30;          Ignored (in strict mode, throws an error)
obj.address = "Bhopal"; Ignored (can't add)
delete obj.name;       Ignored (can't delete)
console.log(obj);      {name: 'Alice', age: 25}
```

eg:-

```
let obj = {
    name: "Bob",
    age: 30
};
Object.seal(obj);
obj.age = 35;          Allowed
obj.address = "Delhi"; Ignored
delete obj.name;       Ignored
console.log(obj)
```

↓

{name: "Bob", age: 35}

Feature	Object.freeze()	Object.seal()
Add new properties	Not allowed	Not allowed
Delete properties	Not allowed	Not allowed
modify existing values	Not allowed	Allowed
change property descriptors	Not allowed	Not allowed
Nested object freezing | sealing	Needs explicit freezing for nested objects	Needs explicit sealing for nested objects

shallow copy - copies the reference.

```
let obj1 = {
    a: 1,
    b: 2
};
let obj2 = obj1;
obj2.a = 10;
console.log(obj2, obj1);
```

→ it changes the values to both the objects.

{a:10, b:2} - obj 2

{a:10, b:2} - obj 1

Deep copy - It shares different addresses.

```
let obj3 = structuredClone(obj1);
obj3.a = 20;
console.log(obj3, obj1); → { a:20, b:2 } { a:10, b:2 }
```

for Nested object.

```
const user = {
    name: "Bhupendar",
    balance: 420,
    address: {
        pincode: 803113,
        city: "Biharsharif"
    }
}
```

it changes the pincode of user also ←

```
const user2 = Object.assign({}, user);
user2.address.pincode = 803110;
console.log(user.address.pincode); → 803110
console.log(user2.address.pincode); → 803110
```

This doesn't change the name of the user. ←

```
user2.name = "Saurav";
console.log(user.name); → Bhupendar
```

Note:-

→ For nested it does shallow copy and for single it does deep copy

→ Spread also does the same.

→ To Avoid this always copy by using structuredClone.

Destructuring of an object

It is a syntax that allows us to unpack properties from objects into distinct variables. It provides a clean and concise way to extract values without directly accessing them via dot notation or bracket notation repeatedly.

Basic object destructuring

```
let obj = {
    name: "Saurav",
    money: 420,
    balance: 30,
    age: 20,
    aadhar: "747398265 43"
}
const { name, balance, age } = obj;
console.log (name, balance, age); → Saurav
                                    30
                                    20
```

Assigning new variables name

We can assign new variable names during destructuring using :

```
const { name: fullName, age: years } = obj; → Now, we can't access
                                                it using name, age.
console.log (fullName, years); Saurav
                               20
```

Destructure with the rest operator

We can use the rest operator (...) to collect the remaining properties into a separate object.

```
const { name, age, ...obj1 } = obj;
console.log (name, age); → Saurav
                           20
console.log (obj1); → { money: 420, balance: 30, aadhar: "747398265" }
```

Destructuring of an array

```
const arr = [ 3, 2, 1, 5, 10 ];
const [first, second] = arr;
console.log (first, second); → 3, 2

const [first, second, , third] = arr;
console.log (first, second, third); → 3, 2, 5

const [first, second, ...third] = arr;
console.log (first, second); → 3, 2
console.log (third); → [1, 5, 10]
```

Destructuring of nested objects

we can destructure properties from nested objects.

```
let obj = {
    name: "Saurav",
    age: 20,
    aadhar: "7u7398uAS3",
    address: {
        pincode: 803113,
        city: "Biharsharif",
        state: "Bihar"
    }
};

const { address: { city } } = obj;
console.log(city); → Biharsharif
const { address: { pincode, state } } = obj;
console.log (pincode, state); → 803113  Bihar.
```

```
let obj = {
    name: "Saurav",
    age: 20,
    arr: [90, 40, 60, 80];
    aadhar: "7u7398uAS3",
    address: {
        pincode: 803113,
        city: "Biharsharif",
        state: "Bihar"
    }
};
```

I want to destructure the first element of the arr

```
const { arr: [first] } = obj;
console.log (first); → 90.
```

Prototype chaining

Interview

```
let obj = {
    name : "Saurav",
    amount: 420,
    greet: function(){
        return 10;
    }
}
console.log( obj. toString() );
```

→ we haven't created a function named toString() than how i am accessing it

→ In array, we haven't created functions like push(), pop(), indexOf() .... etc.

→ Then the question arises that from where it is coming!

ans!- inbuilt function.

→ interviewer will kick you out if we give this answer.

Right answer!- prototype.

prototype chaining in JS is a mechanism that allows objects to inherit properties and methods from other objects.

```
let user1 = {
        name: "Saurav",
        age : 20
    };
let user2 = {
        amount : 150,
        money : 20
    };
user2. name = "mohan" X.
```

we can't do this as name is the property of user1.

So, user2 can't access properties of user1 and viceversa.

→ If we want user2 to access the property of user1 then we have to change the prototype of user2 to user1

```
user2.__proto__ = user1;
console.log(user2.name); → Saurav.
```

→ JS has a object whose name is Array.prototype

↓

It has all the properties like push, pop, includes etc....

Now, if we create an array, it access all the properties like push(), pop(), indexOf() etc...

This happens because array inherits the properties from Array.prototype

```
arr.__proto__ == Array.prototype.
```

→ JS has an Object.prototype, it has some properties like toString(), valuesOf() etc...

→ Array.prototype inherits the properties from Object.prototype

```
Array.prototype == Object.prototype
arr.__proto__.__proto__ = Object.prototype
Object.prototype.__proto__ == null
```

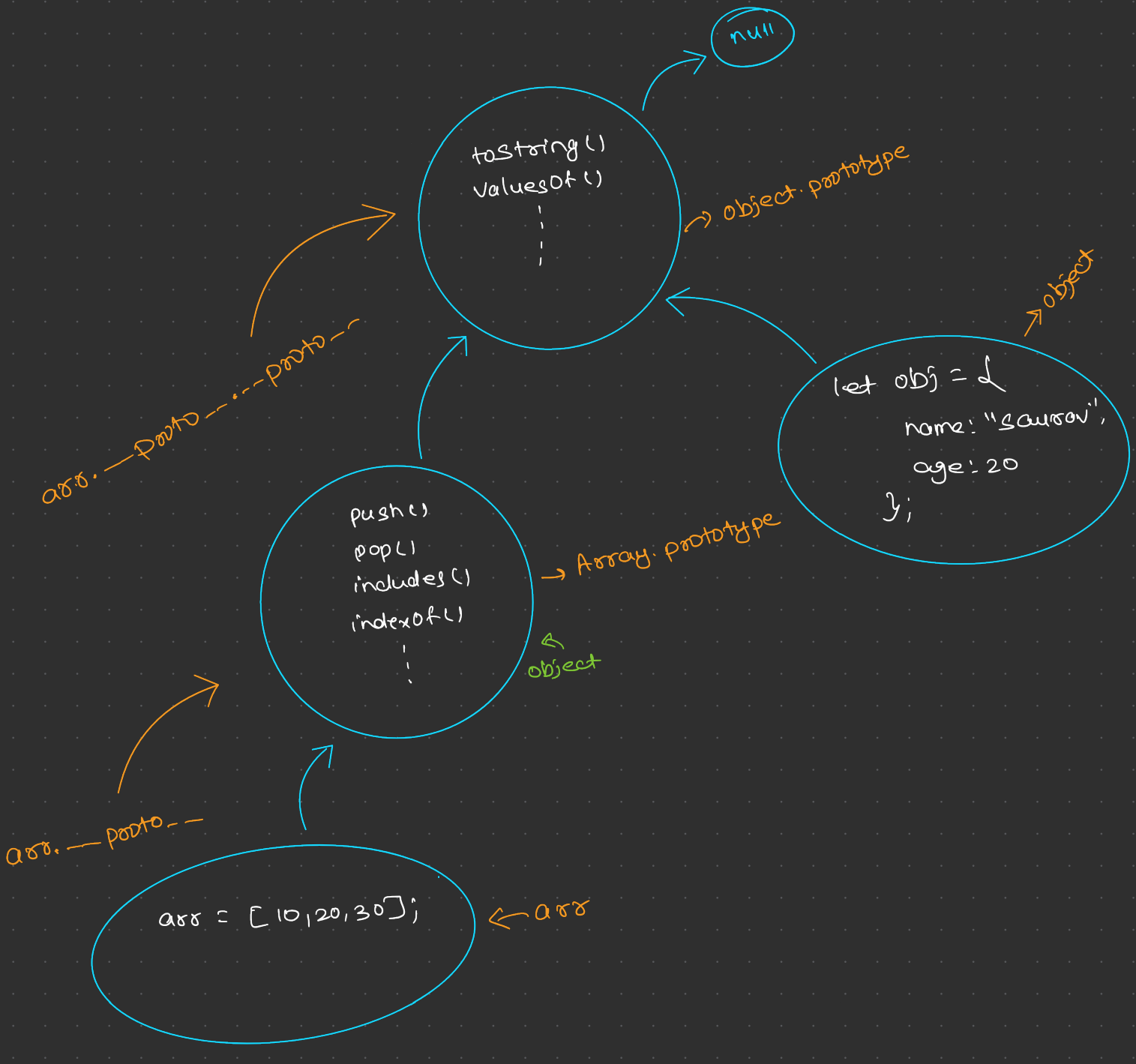
→ whenever we create an object, Object.prototype gets attached to it.

→ whenever we create an array, Array.prototype gets attached to it.

→ All these discussion is known as prototype chaining.

→ This is the reason for why the typeof array is object.

```
let arr = [10,20,30];
console.log(arr.__proto__ == Array.prototype);
console.log(arr.__proto__.__proto__ == Array.prototype.__proto__)
Console.log (arr.__proto__.__proto__ == Object.prototype);
console.log(arr.__proto__.__proto__.__proto__ == null);
```



Function in JS

Function are reusable block of codes that perform specific task.

Function declaration

A function is defined with the function keyword, followed by its name, parenthesis, and a block of code.

```
function greet() {
    console.log("Hello Coder Army");
}
greet(); → function calling.
```

```
// add program          parameter
function sum (a,b) {
    return a+b;
}
let result = sum(3,5);
                ↳ argument
```

Function expression

A function can be assigned to a variable. This is called a function expression.

```
const add = function (a,b) {
    return a+b;
};
console.log (add(2,3)); → 5
```

```
// the statement written below the return keyword will never run.
const fun = function() {
    console.log("Hello Coder Army");
    console.log("mein to Badhiya hoon");
    return "money";
    console.log("Aur kya chal rha hain");
}
                ↑
        this statement never run.
```

Arrow Function

Introduced in ES6, arrow functions provide a shorter syntax. They are especially useful for callbacks and concise one-liners.

```
const sum = (a,b) => {
        return a+b;
    };
console.log(sum(3,4)); → 7
```

→ when we have to write single statement inside block, then it automatically returns the statement.

```
const sum = (a,b) => a+b;
console.log(sum(3,4)); → 7
```

→ when we have single parameter then we don't need to write Parenthesis.

```
const cube = a => a*a*a;
console.log(cube(3)); → 27
```

Rest parameters

The ... Syntax allows us to handle an indefinite number of argument

```
const sum = function(...number){
        let sum=0;
        for(let i=0; i<number.length; i++)
            sum += number[i];
        console.log(sum);
    };
sum(2,3); → 5
sum(2,3,4); → 9
sum(10,18,11,19); → 58
```

↳ it collects all the arguments & store them in an array.

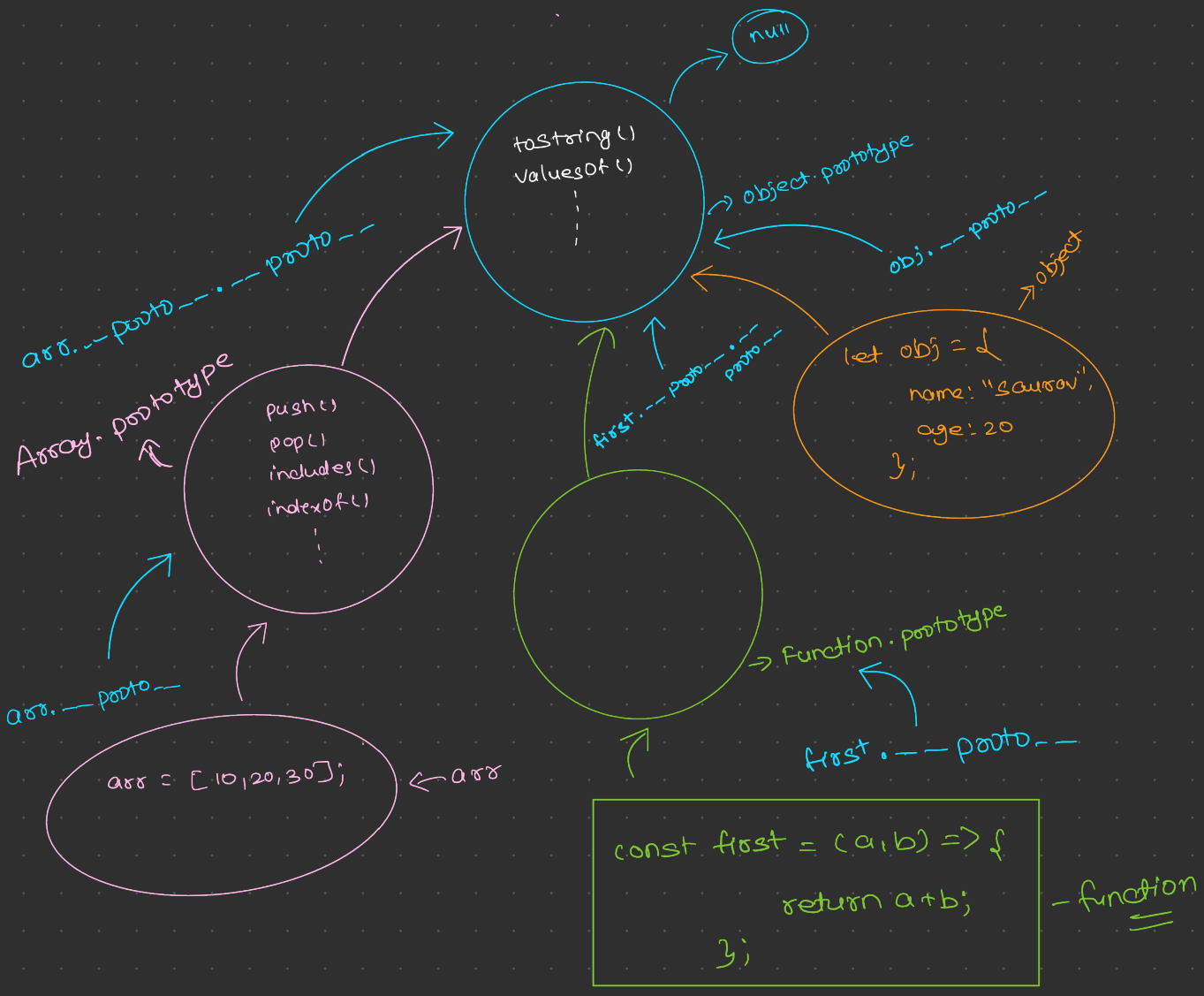
```
let obj = {
    name: "saurav",
    age: 30,
    amount: 420
};
```

// Now, we have to create a function that print name & age of the obj.

```
function fun(obj1){
    console.log(obj.name, obj1.age);
}
fun(obj); → 'saurav', 30
```

// we can destructure it also.

```
function fun2({name, amount}) {
    console.log(name, amount);
};
fun2(obj); → 'Saurav', 420.
```




```
// all below comparison is true
console.log(first.__proto__ == Function.prototype);
console.log(first.__proto__.__proto__ == Function.prototype.__proto__);
console.log(first.__proto__.__proto__ == Object.prototype);
console.log(first.__proto__.__proto__.__proto__ == Object.prototype.__proto__);
console.log(first.__proto__.__proto__.__proto__ == null);
```

Conditional statements

It allows us to perform different actions based on conditions. They help make decision in our code by evaluating expressions and executing corresponding code blocks.

if - statement

The if statement executes a block of code if the specified condition is true.

```
const age = 20;
if (age >= 18) {
    console.log("you are eligible to vote");
}
```

condition

if-else statement

The if-else statement provides an alternative block of code if the condition is false.

```
const age = 16;
if (age >= 18) {
    console.log("you are eligible to vote");
} else {
    console.log("you are not eligible to vote");
}
```

output :- you are not eligible to vote.

if-else if ladder

It evaluates multiple conditions sequentially.

```
let age = 19;
if (age < 18)
    console.log("KID");
else if (age > 45)
    console.log("OLD");
else
    console.log("ADULT");
```

output :- ADULT.

Switch statement

It is used when there are multiple possible values for a variable. It executes the block corresponding to the matching case.

```
switch (new Date().getDay()) {
    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");
}
```

default: ← it runs when None of the above case is true.

Loops in JS

loops in JS are used to repeatedly executes a block of code until a specific condition is met.

For loop

```
for (initialization; condition; increment/decrement) {
    //code to execute
}
```

```
for (let i=1; i<=20; i++) {
    console.log("Hello world");
}
```

prints 'Hello world' 20 times.

while loop

```
while (condition) {
    // code to execute
}
```

```
let i = 0;          → initialization
while (i < 10) {    → condition
    console.log("Hello world");
    i++;            → increment
}
```

prints 'Hello world' 10 times.

Nested loop

We can nest loops inside one another for working with multi-dimensional data structures.

```
const matrix = [
    [1,2,3],
    [4,5,6],
    [7,8,9]
];

outer loop ← for (let i = 0; i < matrix.length; i++)
{
inner loop ← for (let j = 0; j < matrix[i].length; j++)
    {
        console.log(matrix[i][j]);
    }
};
```

output:- 1 2 3 4 5 6 7 8 9

Scope in JS

Scope refers to the accessibility of variables, objects, and functions in different parts of the program. Scope determines where we can access or use a particular variable or function.

Global Scope

Variables declared outside of any function or block have global scope. They can be accessed and modified anywhere in the program.

```
let a = 10;
var b = 20;
const c = 30;
function greet() {
    console.log(a); → 10
    console.log(b); → 20
    console.log(c); → 30
}
greet();
console.log(a); → 10
console.log(b); → 20
console.log(c); → 30
```

local scope (functional scope)

Variables declared inside a function are function-scoped. They can only be accessed from within the function where they defined.

```
function greet() {
    let a = 10;
    var b = 20;
    const c = 30;
    console.log(a); → 10
    console.log(b); → 20
    console.log(c); → 30
}
greet();
console.log(a); → error
console.log(b); → error
console.log(c); → error
```

we can't access any of these variables outside the function. so, the scope of variable is inside the function only.

block scope

variables declared inside a block are block-scoped, they are only accessible within that block.

```
if(true) {
    let a = 10;
    var b = 20;
    const c = 30;
}
console.log(a); → error
console.log(b); → 20
console.log(c); → error
```

```
var amount = 400;
var amount = 20;
```
] This doesn't give error

```
console.log(b);
var b = 20;
```
] → we can use variable declared by var, before its declaration.

Reasons for not using var

→ Variables declared with var are not block-scoped, it ignore the block boundaries (like if, for or {}) and are accessible outside their block.

→ The var keyword allows the same variable to be re-declared within the same scope, which can lead to unexpected behaviour

→ Variables declared with var are accessible before their declaration due to hoisting, which can lead to confusing code:

```
console.log(x); → undefined
var x = 10;
```

Function can be accessed before declaration.

```
greet(); → 'Hello world'
const meet = function greet() {
                console.log("Hello world");
             }
```

Function expression can't be accessed before declaration

```
meet(); → error.
const meet = function() {
                console.log("Hello meet");
             }
```

This is why because the function is inside a variable and that variable can't be accessed before declaration.

Advanced loops

```
let obj = {
    name : "rohan",
    age : 23,
    gender : "male",
    city : "kotdwar"
}
```

for-in-loop

It is used to iterate over the keys of an object.

```
for (let key in obj)
{
    console.log(key); → name age gender city.
}
```

We can access their corresponding values also

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

We can access keys with Object.keys(obj) and with for in loop also. Then, what's the difference?

```
let obj2 = Object.create(obj);
obj2.money = 420;
obj2.id = "Roh";
console.log(object.keys(obj2)); →['money', 'id']
for(let key in obj2)
    console.log(key); → money
                        id
                        name
                        age
                        gender
                        city
```

Object.keys(obj2) :- It only access the keys of the object (obj2).

for-in-loop :- It access both the object and inherited keys.

when we create object it inherits the property from Object.prototype then why the keys of Object.prototype didn't get accessed by for-in-loop as it access the inherited property also.

In JS, writable, enumerable and configurable are property attributes that define the behaviour of an object's properties.

```
let obj = {};
obj.name = "rohit";
console.log( Object.getOwnPropertyDescriptor(obj, 'name'));
```

↓

```
{
    value: 'rohit',
    writable: true,
    enumerable: true,
    configurable: true
}
```

→ By default these three gets added & value of these are true.

writable

Determines if the value of the property can be changed.

- If `writable: true` the property value can be reassigned.
- If `writable: false` the property value is read-only

Attempts to change it fail:

→ In Non-strict mode, the operation silently fails.

→ In strict mode, it throws a typeError.

```
let obj = {};
Object.defineProperty(obj, 'name', {
        value: 'rohit';
        writable: false,
        enumerable: true,
        configurable: true
});
console.log (obj.name); → rohit
obj.name = "mohit";
console.log (obj.name); → rohit
```

↑ writable: false

↑ we can't reassigned the value.

Configurable

Determines if the property's descriptor itself can be modified or if the property can be deleted.

- If configurable : true
  - → The property can be deleted using the delete operator.
  - → The property's attributes (writable, enumerable, configurable) can be changed.
- If configurable : false
  - → The property cannot be deleted.
  - → The property's attributes cannot be changed, except for writable (if it's true, it can be made false)

```
let obj = { };
Object.defineProperty(obj, 'name', {
        value: "rohit",
        writable: true,
        enumerable: true,
        configurable: false
    })
obj.name = "mohit";
console.log(obj.name); → mohit
Object.defineProperty(obj, 'name', {
        writable: false
    })
obj.name = "saurav";
console.log(obj.name); → mohit
```

Here, writable: true → we can modify the values.

Here, writable: false, configurable: false, we can't change the value of writable but we can change the writable from true to false. so, Here it becomes false and the property value is not changed.

```
let obj = { };
Object.defineProperty(obj, 'name', {
        value: "rohit",
        writable: false,
        enumerable: true,
        configurable: false
    })
obj.name = "mohit";
console.log(obj.name); → rohit
```

Here, writable: false → we can't change the values.

```
Object.defineProperty(obj, 'name', {
        writable: true
    }) → it gives error, As we can't change the value of writable from false to true when configurable is false.
```

```
Const customer = {
        name: "saurav",
        age: 21
        account_number : 123456789,
        account_balance: 4820
    }
```

Question:- I don't want that value of name and account_number gets changed.

```
Object.defineProperty(customer, 'name', {
        writable: false,   → value of name can't be changed.
        configurable: false → value of writable can't be changed.
    });
Object.defineProperty(customer, 'account_number', {
        writable: false,   → value of account_numbers can't be changed.
        configurable: false → value of writable can't be changed.
    });
```

Enumerable

Determines if the property will show up in enumeration operations like:

- for ... in
- Object.keys()
- Object.entries()
- If enumerable: true, the property is visible during iteration.
- If enumerable: false, the property is hidden from enumeration methods but can still be accessed directly.

```
Const customer = {
        name: "Saurav",
        age: 21
        account_number : 123456789,
        account_balance: 4820
    }
for ( let key in customer)
        console.log (key); → name age account_number
                             account_balance.

Object.defineProperty (customer, 'name', {
            enumerable: false
    }
for (let key in customer)
        console.log (key); → age account_number account_balance.

let customer2 = Object.create (customer);
customer2.city = "Haridwar";
customer2.place = "Delhi";

for (let key in customer2)
        console.log (key); → place age account_number account_balance
                             city.
```

for...in → it access only those property whose enumerable is true, and even if enumerable is true for inherited properties, it will access it.

Question :- why don't it access object.prototype properties.

```
console.log(Object.getOwnPropertyDescriptor(Object.prototype, 'toString'));
```

↓

```
{
    value: [Function: toString],
    writable: true,
    enumerable: false,
    configurable: true
}
```

enumerable is false so, for...in unable to access it.

Question:- why don't we use for...in loop with Arrays.

```
const arr = [10,20,30,40];
for(let key in arr)
    console.log(key); → 0 1 2 3
```

// we know that, at the end array is an object.
// we can add new properties (key to it.

↓ name of the key is index.

```
arr.name = "saurav";
arr.age = "30";
for(let key in arr)
    console.log(key, arr[key]); → it points name and age also
```

↑ we know that it is not an index.

↓

```
0 10
1 20
2 30
3 40
name saurav
age 30
```

for...in → it doesn't see that it is an array, it has indexes. it only sees the enumerable of the key and by default enumerable : true for everyone, so, it access all the keys.

↑ That's the reason why we are not using for...in for arrays instead of it use normal loop.

```
const customer = {
    name: "Saurav",
    age: 21
    account_number : 123456789,
    account_balance: 4820
}

for (let key in customer)
    console.log (key);
```

name
age
account_number
account_balance

```
Object.defineProperty (Object.prototype, 'toString', {
                enumerable : true
})

for (let key in customer)
    console.log (key);
```

name
age
account_number
account_balance
toString

for of loop

The for...of loop is used to iterate over iterable objects such as arrays, strings or other objects that implement the iterable protocol. It provides a simple and clean way to access each value in a iterable

```
for(variable of iterableObjects){
    //code to be executed.
}
```

variable :- it stores the value of each element in the iterable during each iteration.

Iterating over Arrays

```
const arr = [10,20,11,18,13];
for(let value of arr)
    console.log(value); → 10,20,11,18,13.
```

Iterating over strings

```
let str = "saurav";
for(let value of str)
    console.log(value); → s, a, u, r, a, v
```

Question:- why we are not using for...of in plain objects.

→ for...of loop is not used for objects because objects are not iterable by default.

→ The for...of loop is designed to work with iterable objects, which implement the Symbol.iterator method as part of the iterable protocol.

→ plain objects, like key:value do not have this method.

iterable :- it is an object that defines how to sequentially access its elements via an iterator.

How to iterate over objects using for...of ?

If we want to iterate over the keys, values or both of an object we can use :-

(i) Object.keys() :- It makes an array of keys of the object and then we can iterate over it.

```
const obj = {
    name: "Saurav",
    age: 21,
    city: "New york"
}
for (const key of Object.keys(obj))
    console.log(key); → name age city
```

(ii) Object.values :- It makes an array of values of the object and then we can iterate over it.

```
for (const value of Object.values(obj))
    Console.log(value); → Saurav 21 New york
```

(iii) Object.entries :- It makes an array of key:value of the object.

```
for (const [key, value] of Object.entries(obj))
    console.log(`${key}: ${value}`);
```

↓

name : Saurav
age : 21
city : New york

callback function

A callback function is a function that is passed as an argument to another function and is executed later.

```
function name ( callback) {
        console.log ("Hello, I am name");
        callback();
    }
function greet () {
        console.log("Hello, I am callback function");
    }
name(greet);
```

Here, greet is a function that is passed as an argument of another function (name). so, greet is a callback function.

```
// we can direct define the function in argument.
name( function greet(){
        console.log ("Hello, I am callback function");
    });

// we can use arrow function in argument.
name( () => {
        console.log("Hello, I am callback function");
    });

// we can use function expression also.
const fun = function() {
        console.log ("Hello, I am callback function");
    };
name(fun);
```

for Each

→ The forEach method is an array method used to execute a provided function once for each array element.

```
array.forEach(callback(currentValue, index, array));
```

↑ a function that is executed on each element of the array.

```
let arr = [10, 20, 30, 40, 50];
arr.forEach( function (number) {
        console.log (number);
});
```
→ 10 20 30 40 50

```
arr.forEach((num, index) => console.log(num, index));
```

10 0
20 1
30 2
40 3
50 4

reference of the arr.

```
arr.forEach ( (num, index, a) => {
        a[index] = num * 2;
});
console.log(arr);
```
→ 20 40 60 80 100

filter

The filter method is used to create a new array containing all the elements of the original array that satisfy a given condition. It does not modify the original array

```
array.filter (callback (element, index, array ));
```

this callback should return
true → to include the element in the new array.
false → to exclude the element.

```
let arr = [10, 22, 33, 41, 50];
const result = arr.filter ( (num) => num%2 == 0 );
console.log (result);
```
→ 10 22 50

```
const students = [
    { name: "rohan", age: 22, marks: 70 },
    { name: "mohan", age: 24, marks: 80 },
    { name: "Darshan", age: 28, marks: 30 },
    { name: "mohit", age: 32, marks: 40 },
    { name: "saurav", age: 20, marks: 90 }
]
```

```
const result2 = students.filter((obj) => {
                    return obj.marks > 50;
                });
console.log(result2); → [ { name: "rohan", age: 22, marks: 70 },
                          { name: "mohan", age: 24, marks: 80 },
                          { name: "saurav", age: 20, marks: 90 } ]
```

// As it is a object, we can destructure it.

```
const result3 = students.filter(({marks}) => marks > 50);
console.log(result3);
```

↓

```
[ { name: "rohan", age: 22, marks: 70 },
  { name: "mohan", age: 24, marks: 80 },
  { name: "saurav", age: 20, marks: 90 } ]
```

map

The map method is used to create a new array by applying a provided callback function to each element of the original array. It is often used for transforming data.

```
array.map(callback(currentValue, index, array));
```

```
const arr = [1,2,4,5];
const result = arr.map((num) => {
                   return num * num;
               });
console.log(result); → [1, 4, 16, 25]
```

Q. we want square of even number.

```
let arr2 = [1,2,3,4,5,6];
const result2 = arr2.filter((num) => num%2==0)
                    .map((num) => num * num);

console.log(result2);
```

↑

```
[4, 16, 36]
```

reduce

The reduce method is a powerful array method that executes a reducer function on each element of the array, resulting in a single output value. It is commonly used for tasks like summing numbers, flattening arrays or aggregating data

```
array.reduce(callback, initialValue)
```

```
const arr = [10, 20, 30, 40, 50];
const result = arr.reduce((acc, curr) => acc + curr, 0);
console.log(result); => 150
```

callback :- A function that is executed on each element in the array. It takes four arguments :-

- accumulator :- The accumulated value returned by the last execution of the callback, or the initial value on the first iteration.
- currentValue :- The current element being processed in the array.
- currentIndex :- The index of the current element. (optional)
- array :- The array on which reduce() was called.

initial Value :- (optional) The initial value of the accumulator. If not provided, the first element of the array will be used as the initial value, and the iteration will start from the second element.

eg :-

```
let arr = ["orange", "apple", "banana", "orange", "apple", "banana",
           "orange", "grapes"];
const result = arr.reduce((acc, curr) => {
    if (acc.hasOwnProperty(curr))
        acc[curr]++;
    else
        acc[curr] = 1;
    return acc;
}, {})
console.log(result); -> {orange: 3, apple: 2, banana: 2, grapes: 1}
```

map

It is a collection of key-value pairs where keys can be of any data type, unlike objects where keys are always string or Symbols. It preserves the order of insertion of the entries, making it ideal for scenarios where key-value mappings with various key types are needed.

creating a map

```
const myMap = new Map();

myMap.set(3,90);
myMap.set("Rohit","mohit");
myMap.set(20,"mohan");
console.log(myMap); -> Map(3) { 3 => 90, 'Rohit' => 'mohit', 20 => 'mohan' }
```

Method	Description
set(key, value)	Adds a key-value pair to the map. If the key already exists, its value is updated.
get(key)	Retrieves the value associated with the specified key. Returns undefined if the key is not found.
has(key)	checks if a key exists in the map. Returns true or false.
delete(key)	Removes a specific key-value pair. Returns true if deleted, otherwise false.
clear()	Removes all key-value pairs from the map.
size()	Returns the number of key-value pairs in the map.

Sets

→ A set is a built-in object that allows us to store unique values of any type, whether primitive values or references to objects

→ A set automatically removes duplicate values and provides various methods to interact with the collection.

→ The elements in a set are stored in the order in which they are added.

→ we can iterate through the elements of set using forEach and for ... of.

Creating a set

```
const set1 = new Set([10,20,30,40,10,30]);
console.log(set1); → Set(4) {10,20,30,40}

const set2 = new Set();
set2.add(4);
set2.add(6);
set2.add("Saurav");
console.log(set2); → Set(3) {4,6,'Saurav'}
```

delete

```
set2.delete(6);
Console.log(set2); → set(2) {4,'Saurav'}
```

has(value):- checks if the value exists in the set.

clear():- removes all values from the set.

size:- returns the no. of elements in the set.

Convert array in set

```
let arr = [10,20,30,10,40,50];
const set1 = new set(arr);
console.log(set1);
```

Convert set in array

```
arr = [...set1];
console.log(arr);
```

How does JS code works and Hoisting in JS

```
console.log(x);   → undefined
var x = 10;
```

```
console.log(y);   → error, cannot access y before initialization
let y = 20;
```

```
console.log(b);   → b is not defined
```

```
var x = 10;
let y = 20;
const z = 30;
console.log(x);
console.log(y);
console.log(z);
```

Execution context

Memory	Code
x: undefined → 10 y: 20 z: 30	x = 10 y = 20 z = 30 console.log(x) console.log(y) console.log(z)

temporal dead zone.

→ creates an execution context, it has two parts memory and code.

→ first it allocate the memory of x, and assign undefined to it, then it allocate memory to y and doesn't assign anything to it same happens with const variable (z). y and z is known as Temporal Dead zone (TDZ).

→ Now, code execution phase starts, 10 is assigned in x, 20 in y, 30 in z and at last it prints the value of x, y and z

JS is a Synchronous single threaded language.

- Synchronous → executes in a synchronise way.
- single threaded → one instruction at a time.

```
console.log(x);
console.log(y);
var x = 10;
let y = 20;
```

Execution context

memory	code

x: undefined → 10
y: 20
	console.log(x) → undefined
	console.log(y) → y is not initialised
	x = 10;
	y = 20;

execution context

1st phase :- memory allocate

2nd phase :- code execution.

Temporal Dead Zone :-

→ It refers to the period between the entering of a block scope (where a variable is declared using let or const and the point where the variable is initialized or assigned a value.

→ when the variable is initialised or assigned, the variable comes outside from temporal dead zone.

→ we can't access the variable which is in temporal dead zone, it gives reference error or variable is not initialized.

Hoisting

→ It is the behaviour where variable and function declarations are moved to the top of their containing scope during the compilation phase, before the code has been executed.

→ This means that JS "hoists" or lifts the declaration (but not the assignments) to the top, so we can refer to them before they are written in code. However, only the declaration is hoisted, not the assignments or initializations.

```
console.log(x);
console.log(y);
var x = 10;
let y = 20;
```

what hoisting did with this code

```
var x = undefined;
let y;
console.log(x);
console.log(y);
x = 10;
y = 20;
```

```
let a = 10;
let b = 20;
function add (num1, num2)
{
    let result = num1 + num2;
    return result;
}
var ans = add (a, b);
console.log (ans);
```

Execution context

memory	code
a : 10 b : 20 add : { function code } ans : undefined → 30	Execution context memory: num1 : 10, num2 : 20, result : 30 code return console.log (ans);

Scenario 1

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

Execution context

memory	code
greet: function code	greet()

Execution context

① memory allocation
greet : function code

② code execution phase
It sees it is declared with the function code, so it executes it without throwing an error.

Scenario 2

```
meet();
let meet = function(){
    console.log("meet");
}
```

Execution context

memory	code

meet:	meet()
	 meet = function(){
	 }

Execution context

① memory allocation
meet :

② code execution
It sees meet(), but there is nothing in the meet, so it throws an error, cannot access meet before initialisation

Scenario 3

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

Execution Context

memory	code

meet: undefined	meet();
	 meet = function(){
	 }

Execution context

① memory allocation
meet: undefined

② code execution
it sees meet(), it has undefined, so it throws an error meet is not a function.

Understanding this keyword in JS

The 'this' keyword in Javascript is a special keyword that refers to the context in which the current code is being executed. Its value depends on how the function where 'this' is used is called.

Global context (outside any function)

- In Browser :- 'this' refers to the window object.
- In Node.js :- 'this' refers to the module's exports object.

Inside a function

(i) Non-strict mode :- when 'this' is used inside a regular function, it refers to the global object.

```
function greet() {
    console.log(this); → logs the global object (e.g., window in browsers).
}
greet();
```

(ii) Strict mode :- 'this' will be undefined inside a regular function.

```
'use strict';
function greet() {
    console.log(this);
}
greet(); // logs undefined
```

Inside a method (Object context)

When 'this' is used inside an object's method, it refers to the object that owns the method.

```
const obj = {
    name: "Rohit",
    age: 20,
    meet: function() {
        console.log(this.name);
    }
};
obj.meet(); // logs "Rohit"
```

Arrow Functions

Arrow functions do not have their own 'this'. Instead, they inherit 'this' from the surrounding (lexical) scope.

```
let obj = {
    name: "Rohit",
    age : 21,
    meet : () => {
        console.log(this);
    }
};
obj.meet(); // logs the global object (not the obj itself)
```

However, when an arrow function is used inside a regular function or method, it inherits 'this' from enclosing function.

```
let obj = {
    name: "Rohit",
    age : 21,
    greet: function() {
        let ab = () => {
            console.log(this);
        };
        ab();
    }
};
obj.greet(); // logs the obj
```

Inside a constructor or a class

In constructors and classes, 'this' refers to the instance of the object being created.

```
class person {
    constructor (name, age) {
        this.name = name;
        this.age = age;
    }
}
let a = new Person ("Saurav", 20);
console.log(a); // logs the Person object
                // { name: "Saurav", age: 20 }
```

Document object model

→ It is a programming interface for web documents. It represents the structure of a web page in a way that allows programs (like Javascript) to interact with it.

→ A browser takes our HTML document and converts it into the DOM.

→ Using the Dom, Javascript can:

- change all the HTML elements on the page.
- modify all the HTML attributes on the page.
- Alter all the CSS styles on the page.
- Remove existing HTML elements and attributes.
- Add new HTML elements and attributes.
- React to all existing HTML events on the page.
- Create new HTML events for the page.

Accessing elements in the DOM

(1.) By ID

method :- document.getElementById('id')

description :- Retrieves a single element with the specified 'id' attribute.

```
const obj = document.getElementById('first');
console.log(obj.id); → first
console.log(obj.innerHTML); → Hello Coder Army
```

(2.) By class Name

method :- document.getElementByClassName(className)

description :- Return a live HTMLcollection of all elements with the specified class name.

```
const obj2 = document.getElementsByClassName('header1');
console.log(obj2); // HTMLCollection(2) [h1#first.header1, h2#second.header1, first: h1#first.header1, second: h2#second.header1]
•  console.log(obj2[0]); //<h1 id="first" class="header1">Hello Mahan Aadmi</h1>

obj2[1].style.backgroundColor = "pink";
console.log(typeof obj2[1].style); // object
console.log(obj2[1].style); // CSSStyleDeclaration {0: 'background-color', accentColor: '', additiveSymbols: '', alignContent: '', alignItems: '', alignSelf: '', …}
```

(3.) By css selectors

(i) Single element

method :- document.querySelector(selector)

Description :- Returns the first element matching the specified css selector.

```
const id = document.querySelector('#first');
id.innerHTML = "Hello money";
const id2 = document.querySelector('.header1');
id2.style.backgroundColor = "red";
```

it select the first element that having class header1

(ii) Multiple Elements

method :- document.querySelectorAll(selector)

Description :- Returns a static NodeList of all elements with the specified css selector.

```
Const obj = document.querySelectorAll('.header1');
```

→ we can iterate over Nodelist using for...of, forEach and normal forloop.

→ To use the properties like filter, map... we have to convert the nodelist into array.

```
Array.from(obj);
```

(4.) By tag Name

method :- document.getElementsByTagName(tagName)

Description :- Returns a live HTMLcollections of all elements with the specified tagName (e.g.. div, p, a).

```
let obj4 = document.getElementByTagName('li');
console.log(obj4); → HTMLcollection(4) [li, li, li, li]
```

→ we can iterate over HTMLcollections using for.. of and regular loop.

→ we can't use forEach or other methods on Htmlcollections. To use it, we have to convert our HTML collections into array.

```
let arr = Array.from(obj4);
```

(5.) Using Relationships

(i) Parent Node :-

Method :- element.parentNode or element.parentElement

Description :- Access the immediate parent of an element.

```
Const list = document.querySelector('li');
Console.log (list.parentNode);  → ul #third.header2
Console.log(list.parentElement); → ul #third.header2
```

(ii) Child Nodes :-

Methods :- element.childNodes (includes text nodes)
element.children (only element nodes)

Description :- Access all child nodes of an element.

```
Const par = document.querySelector('ul');
console.log (par.childNodes); → NodeList(9) [text, li, text, li, text, li, text, li, text]
Console.log (par.children); HTMLCollections(4) [li, li, li, li]
```

(iii) First and last child

Methods :- element.firstChild, element.lastChild
element.firstElementChild, element.lastElementChild

Description :- Access the first or last child of an element.

(iv) Sibling Nodes

Methods :- element.nextSibling, element.previousSibling,
element.nextElementSibling, element.previousElementSibling.

Description :- Access the sibling nodes of an element.

```
// innerHtml :- it prints all the things it has even other html,element
// textContent :- it gives only the text by removing all the tags used  inside them
// innerText :-  it gives only the text which can be seen in the browser, it doesn't gives us  hidden text

console.log(document.getElementById('first').innerHTML); // Hello Coder <strong style="display: none;">Army</strong>
console.log(document.getElementById('first').textContent); // Hello Coder Army
console.log(document.getElementById('first').innerText); // Hello Coder
```

The Dom provides an overview of creating and manipulating nodes in the Dom, accessing and modifying attributes, and various methods for adding and removing nodes.

Create nodes

Creating nodes is the first step in dynamically building a webpage. The three main types of nodes are:-

① create an element
This method creates a new HTML element

```
const element = document.createElement('div');
```

② create a Text node
This method creates a text node that contains plain text.

```
const textNode = document.createTextNode("Hello, world!");
```

③ create an Attribute node
This method creates an attribute node, which can then be assigned to an element.

```
const attribute = document.createAttribute('id');
attribute.value = 'unique-id';
```

Accessing attribute

Attributes of an HTML element can be accessed, modified, or removed using the following methods:-

① getAttribute :- Retrieves the value of specified attribute.

```
const classAttr = element.getAttribute('class');
Console.log(classAttr);
```

② setAttribute :- creates a new Attribute if it doesn't exist or updates its value if it does.

```
element.setAttribute('data-role', 'admin');
```

③ removeAttribute :- Removes the specified attribute from the element.

```
element.removeAttribute('data-role');
```

Adding Nodes to the Dom

Nodes can be added to the Dom using various methods :-

① appendChild(node) :- Adds a single node as the last child of a parent element.

```
Parent.appendChild(newNode);
```

② append(node1, node2, ----) :- Adds multiple nodes as the last children of a parent element.

```
Parent.append(node1, node2);
```

③ insertBefore(newNode, referenceNode) :- inserts a node before the specified reference node.

```
Parent.insertBefore(newNode, referenceNode);
```

④ prepend(node) :- Adds a node as the first child of a parent element.

```
Parent.prepend(newNode);
```

⑤ replaceChild(newNode, oldNode) :- Replaces an existing child node with a new Node.

```
Parent.replaceChild(newNode, oldNode);
```

⑥ using innerHTML :- sets the HTML content of an element directly.

```
Parent.innerHTML += '<li> New item </li>';
```

⑦ using insertAdjacentHTML/Element :- Allows insertion of HTML or elements at a specified positions relative to an element.

```
parent.insertAdjacentHTML('beforebegin', '<div> before parent </div>');
Parent.insertAdjacentHTML('afterend', '<div> after parent </div>');
Parent.insertAdjacentHTML('afterbegin', '<div> first child </div>');
Parent.insertAdjacentHTML('beforeend', '<div> last child </div>');

parent.insertAdjacentElement('beforebegin', element);
```

afterend
afterbegin
beforeend

Removing Nodes from the DOM

Nodes can be removed using these methods:-

① removeChild(node):- removes a specified child node.

```
parent.removeChild(childNode);
```

② remove():- Removes the element from the DOM.

```
element.remove();
```

Event in javascript

An event refers to an action or occurrence that happens in the browser, such as clicking a button, submiting a form, typing in a text field. Event allows us to make our web pages interactive and respond to user action.

Types of Events:-

(1.) mouse events

- click:- triggered when an element is clicked.
- dblclick:- triggred when an element is double-clicked.
- mousedown:- triggred when a mouse button is pressed.
- mouseup:- triggered when a mouse button is released.
- mousemove:- triggered when the mouse moves over an element.
- mouseover:- triggered when the mouse pointer enters an element.
- mouseout:- triggered when the mouse pointer leaves an element.

(2.) keyboard events

- keydown:- triggered when a key is pressed down.
- keyup:- triggered when a key is released.

(3.) Form Events

- submit:- triggered when a form is submitted.
- change:- triggered when the value of a form element changes.
- focus:- triggered when an element gains focus.
- blur:- triggered when an element loses focus

Adding Event listeners

To handle events, we use event listeners

ex:-

```
const button = document.getElementById('btn');
button.addEventListener('click', function(){
    alert('Button clicked');
});
```

Removing Event Listeners

we can remove an event listener using removeEventListener. The event listener must be assigned to a named function to be removable.

```
<button id="btn">Click Me</button>
<script>
  function showAlert() {
    alert('Button clicked!');
  }

  const button = document.getElementById('btn');
  button.addEventListener('click', showAlert);

  // Remove the event listener after 5 seconds
  setTimeout(() => {
    button.removeEventListener('click', showAlert);
  }, 5000);
</script>
```

Event object

It contains all the information about the event that occurred, such as the type of the event, the target element, and any additional properties specific to that event type.

The event object is automatically passed as a parameter to event listener functions

(1) common properties

type :- The type of event (e.g. click, keydown, submit)

```
element.addEventListener("click", (event) => {
    console.log(event.type); // output : "click"
});
```

target :- The element on which the event occurred.

```
element.addEventListener("click", (event) => {
    console.log(event.target); // the clicked element
});
```

currentTarget :- The element to which the event handler is attached (different from target in the case of event delegation)

```
parent.addEventListener("click", (event) => {
    console.log(event.currentTarget); // always the parent element
});
```

(2) methods of event object

preventDefault() :- prevents the default action associated with the event (e.g. stopping a form submission)

```
form.addEventListener("submit", (event) => {
    event.preventDefault(); // prevents form submission
    console.log("form submission prevented");
});
```

stopPropagation() :-

stops the event from propagating further up or down the DOM.

```
child.addEventListener("click", (event) => {
    event.stopPropagation(); // prevents even from reaching parent.
    console.log("propagation stopped");
});
```

③ mouse event properties

clientX and clientY :- The mouse pointer's coordinates relative to the viewport.

```
element.addEventListener("mousemove", (event) => {
    console.log(`mouse position: ${event.clientX},
                                 ${event.clientY}`);
});
```

pageX and pageY :- The mouse pointer's coordinate relative to the entire document.

button :- indicates which button was pressed (0 = left, 1 = middle, 2 = right)

④ Keyboard event properties

key :- The key value of the key that was pressed.

```
element.addEventListener("keydown", (event) => {
    console.log(event.key); //outputs the key pressed.
});
```

Event Bubbling

In event bubbling, the event starts from the target element and bubbles up to its ancestors (parent, grandparent etc...) By default, events bubbles from the innermost element to the outermost.

```
child.addEventListener('click', (event) => {
    console.log("child clicked");
}, false);

parent.addEventListener('click', (event) => {
    console.log("parent clicked");
}, false);

grandparent.addEventListener('click', (event) => {
    console.log("grandparent clicked");
}, false);
```

when we click on child, the event first triggers to the child, then it bubbles to the parent, and finally to the grandparent

Event Capturing

In event capturing, the event starts from the outermost element and works its way down to the target element. we can enable event capturing by passing true as the third argument in the addEventListener method.

```
child.addEventListener('click', (event) => {
        console.log("child clicked");
}, true);

parent.addEventListener('click', (event) => {
        console.log("parent clicked");
}, true);

grandparent.addEventListener('click', (event) => {
        console.log("grandparent clicked");
}, true);
```

Event Delegation

Event delegation allows us to attach a single event listener to a parent element instead of attaching one to each child element.

```
document.getElementById('parent').addEventListener('click', (event) => {
        if(event.target.id === "child")
                console.log("child clicked");
});
```

Form Handling in JS

Event listeners

① input event

This event triggers whenever there is any change in the input field. It's like when you're typing, and JS know it.

```
form.addEventListener('input', (event) => {
    console.log(event.target.value); // logs the value as we type.
});
```

② change event

Triggered when the input field loses focus and has been modified.

```
form.addEventListener('change', (event) => {
    console.log(event.target.value); // logs the value after focus change.
});
```

③ Focus and Focusin event

focus : doesn't bubble, it's attached to individual input fields.

focusin : Bubbles, so we can attach it to the form element itself and catch focus events on all inputs.

④ Blur and focusout event

blur : doesn't bubble, attached to individual input fields.

focusout : Bubbles, and it happens when we move focus out of the element.

⑤ click and double Click event

click :- gets triggered when we click anywhere in the form.

dblclick :- double click the mouse to see the event trigger.

⑥ submit event

Triggered when the form is submitted. But wait your page reloads! Not anymore with preventDefault()

```
form.addEventListener('submit', (event) => {
    event.preventDefault();
    console.log("Form submitted");
});
```

⑦ Reset Event

triggered when we reset the form (hit that reset button!). The form will clear all the input fields.

FormData API

→ It allows us to collect all form data easily.

→ The formdata object allows us to gather form data in key-value pairs and perform different operations like looping throw them or converting them into arrays

```
form.addEventListener('submit', (event) => {
    event.preventDefault();

    const data = new FormData(form); // Collects all form data
    // You can access individual values using keys

    // Get keys, values, or both
    for (let key of data.keys()) {
        console.log(key);  // Logs all the keys
    }

    for (let value of data.values()) {
        console.log(value);  // Logs all the values
    }

    // Get entries as key-value pairs
    for (let [key, value] of data.entries()) {
        console.log(key, value);  // Logs key and value pairs
    }

    // Convert the entries to arrays
    console.log(Array.from(data.keys()));  // Array of keys
    console.log(Array.from(data.values()));  // Array of values
    console.log(Array.from(data.entries()));  // Array of key-value pairs
})
```

Callback hell

What is a Callback?

A callback function is a function passed as an argument to another function, to be "called back" later when the task is complete. It's like saying:

> "Hey Javascript, once you're done fetching the user data, call me back so I can greet them!"

Example without callback

```
function fetchuser(){
        console.log("Fetching the user detail.....");
        setTimeout(()=>{
                Console.log("Data fetched successfully");
                const name="Saurav";
                greet(name);
                meet(name);
            },2000)
        }

    function greet(name){
            console.log(`Hello ${name}`);
        }
    function meet(name){
            console.log(`Hello ${name}, I will meet you in Delhi`);
        }
    fetchuser();
```

Here, the code works but lacks flexibility. We can't dynamically decide what to do with fetched data.

Example with callback

```
function fetchuser(callback){
        console.log("Fetching the user detail.....");
        setTimeout(()=>{
                console.log("Data Fetched successfully");
                const name="Saurav";
                callback(name);
            },2000);
}
```

with callbacks, we decide what task to execute after fetching the data. This is more reusable and flexible!

Callback Hell

The pizza delivery example

Think of a scenario where we order pizza from Domino's. Here's the sequence:

(1.) place the order

(2) prepare the order

(3) pickup the order from restaurant

(4.) Deliver the order

```
function placeOrder (callback) {
        Console.log (" Talking with Domino's");
        setTimeout (() => {
            console.log ("order placed successfully");
            callback();
        }, 2000);
}

function preparingOrder (callback) {
        console.log (" Pizza preparation started ...");
        setTimeout (() => {
            console.log ("Pizza preparation done ! ");
            callback();
        }, 5000);
}

function pickupOrder (callback) {
        console.log ("Reaching restaurant to pick up the order ...");
        setTimeout (() => {
            console.log ("order picked up by the delivery
                                                    boy");
            callback();
        }, 3000);
}
```

```
function deliverOrder(){
        console.log("Delivery boy is on the way ....");
        setTimeout(() =>{
            console.log("order delivered successfully");
        }, 5000);
}

placeOrder(() => {
        PreparingOrder(() => {
            Pickuporder(() =>{
                deliverOrder();
            });
        });
});
```

Talking with Domino's
order placed successfully
Pizza preparation started...
Pizza preparation done!
Reaching restaurant to pick up the order...
order picked up by the Delivery boy
Delivery boy is on the way...
order delivered successfully

Problems with Callback Hell

(1.) Reduced readability
- Nested callbacks make the code harder to read and understand.

(2.) Difficult Debugging
- Tracking errors in deeply nested callbacks is a nightmare.

(3.) Dependency Issues
- Each callback depends on the completion of the previous one, making it fragile.

Javascript: Single-Threaded and Asynchronous

Javascript is a single threaded synchronous programming language but also exhibits asynchronous behaviour

Single-threaded Nature

In JS, single-threaded means
- only one task at a time
- Tasks are processed sequentially.

```
console.log(10);
const timer = Date.now();
while (Date.now() - timer < 2000) {
        //wait for 2seconds
}
console.log(20);
console.log(30);
```

output: 10 20 30

Here, JS processes the while loop synchronously, blocking further execution until 2 seconds have elapsed.

Asynchronous Behaviour

Javascript's asynchronous behaviour comes into play when working with tasks like:
- setTimeout
- Event listeners
- HTTP request

These tasks are handled by web APIs provided by the browser allowing javascript to remain non-blocking.

```
console.log(10);
setTimeout(() => {
    console.log(20);
}, 2000);
console.log(30);
```

output:-
10
30
20

Why this order?

① console.log(10) executes first.

② setTimeout() registers the callback function and hands it to the web APIs

③ console.log(30) executes immediately since Javascript doesn't wait for the timer.

④ After 2 seconds, the callback function from setTimeout executes, printing 20

Key points to understand

① setTimeout is not part of JS

- It's a web API provided by the browser
- JS delegates tasks like settimeout to the browser

(2.) other Examples of web APIs

- Dom manipulation :- The window object, document, and event listeners are part of the browsers, not JS itself.
- Asynchronous Tasks:- These include HTTP requests, timers, and events.

(3.) Asynchronous Tasks use callbacks

- Every asynchronous task expects a callback function to execute once the task is complete.

Diagram of Execution

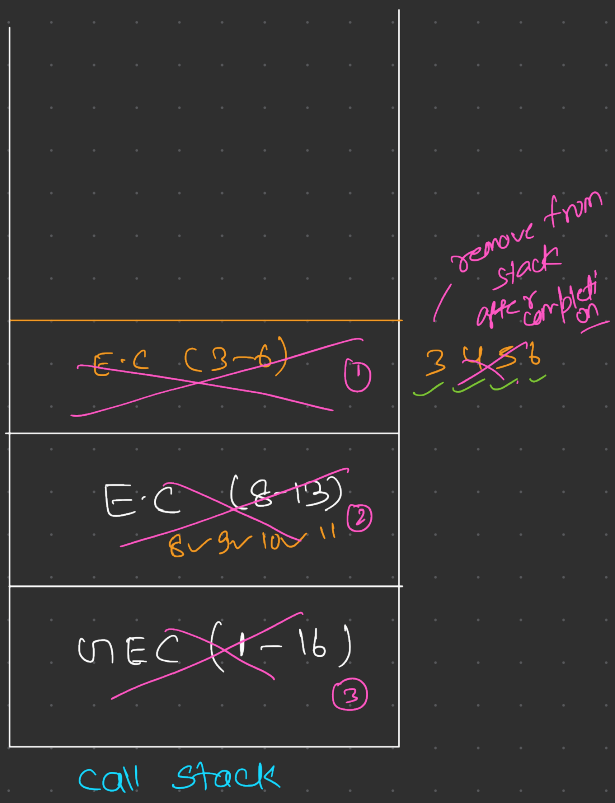
(1.) Javascript sends asynchronous tasks to web APIs.

(2.) web API handle the task.

(3.) once complete, the callback is sent to the event loop.

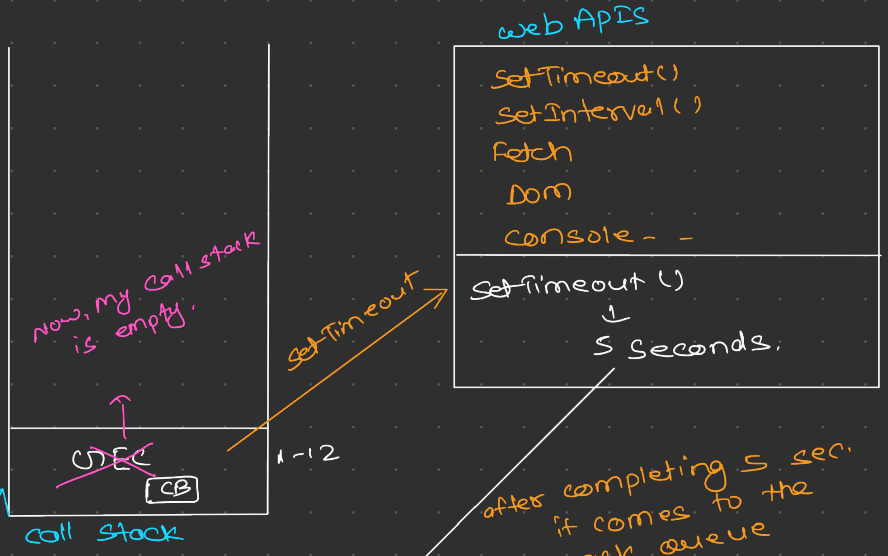
(4.) The event loop ensures callbacks are executed in the main thread once the call stack is empty.

```js
console.log("Hello Coder Army");

function meet(){
    const arr = [2,4,6];
    console.log(arr[0]);
}

function greet(){
    const a = 2 + 3;
    console.log(a);
    meet();
    console.log(a*a);
}

greet();
console.log("Program End");
```



Hello Coder Army
5
2
25
Program End

```
03JS > Day29 > JS second.js > ...
1   console.log("HEllo Coder Army");
2
3   setTimeout(()=>{
4     const a = 2+4;
5     console.log(a);
6   },5000);
7
8   let b = 20;
9   let arr = [20,30,11];
10
11  for(let i of arr)
12      console.log(i*b);
13
14
```

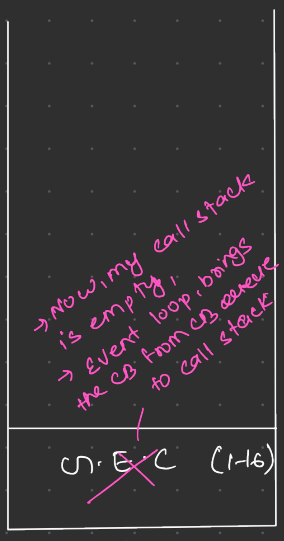


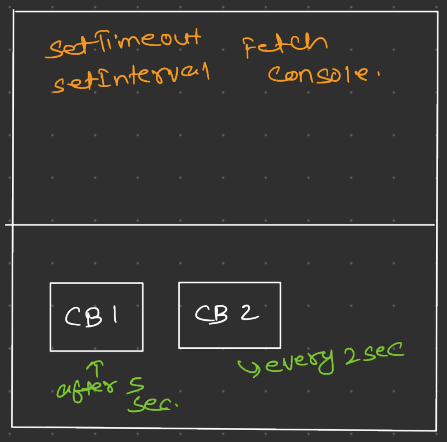
HEllo Coder Army
400
600
220
6 → after 5 seconds.

when the callstack is empty, it takes the CB from the queue and place it into the call stack.

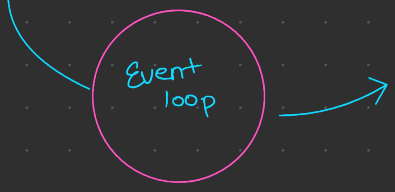
↳ at the end callback queue is also empty.

```
1  console.log("HEllo Coder Army");
2
3  setTimeout(()=>{
4    const a = 2+4;        -> CB 1
5    console.log(a);
6  },5000);
7
8  setInterval(() => {
9      console.log("I am fast");  ->CB 2
10 }, 2000);
11
12 let b = 20;
13 let arr = [20,30,11];
14
15 for(let i of arr)
16     console.log(i*b);
```





Hello coder Army
400
600
220
I am fast
I am fast
6
I am fast
:




```
console.log ("Hello")
setTimeout (() => {
      Console.log ("callback");
}, 0);
:
:
1000 lines of Code
Console.log ("End");
```

Hello
:
:
1000lines of code
End
callback.

This callback will always run when the callstack is empty.

↓ why ?

If we run in the middle of the code execution then, it will create a race around condition.

↓

A scenario where the outcome of the program depends on the sequence or timing of events.

→ Suppose we have 2 device, 1st device run 300 lines of code in 1sec, and by that my setTimeout is completed, now it will directly go to the callstack, after running 300 lines this will run. ←

→ In 2nd device, till the time setTimeout Completed it runs 500 lines of code

→ This leads to change in output in different devices, but we don't want this.

→ we always wants the same output

Suppose,

→ we have 4 buttons where we have added event listener on click event.

→ But for JS, it is not possible to listen at all these 4 buttons at same time.

→ As JS is single threaded, it will only listen one button at a time.

→ This is why eventlisteners is the part of web APIs.

Promises in JS

fetch()

The fetch API in javascript is used to make HTTP requests from web browsers. It provides a modern, flexible, and promise-based interface to interact with resources over the network.

```
fetch("url");
   ↑
it returns a promise
```

Promise

It is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises simplify managing asynchronous code by allowing you to write cleaner and more maintainable code compared to traditional callback functions

Basic syntax of a promise

```
let promise = new promise ((resolve, reject) => {
        //perform an asynchronous operation
        if (success)
            resolve(value); // Fulfill the promise
        else
            reject(error); // Reject the promise
});
```

resolve :- A function to resolve the promise when the operation is successful.

reject :- A function to reject the promise when the operation fails.

Promise state

(1) pending :- The initial state of the promise.

(2) Fulfilled :- The operation completed successfully, and resolve was called.

(3) Rejected :- The operation failed, and reject was called.

Handling promises

```
const promise = fetch("url");
promise.then(response =>{
        console.log(response.json());
});
```

→ it also returns a promise, it's also a async task.

```
const promise = fetch("url");
promise.then((response) => {
        const pro2 = response.json();
        pro2.then((data) => {
                console.log(data);
        })
})
```

```
const promise = fetch("url");
const pro2 = promise.then((response) => {
        return response.json();
});
pro2.then((data) => {
        console.log(data);
});
```

promise chaining

```
const promise = fetch("url");
const pro2 = promise.then((response) => {
        return response.json();
}).then((data) => {
        console.log(data);
})
promise.then(response => response.json()).then(data => console.log(data));
```

```
const promise = ("fetch")
  .then (response => response.json())
  .then (data => console.log (data))
  .catch (error => console.log (error));
```

Async and Await

async and await are used together to handle asynchronous operations in a cleaner and more readable way compared to traditional promises or callback functions. They allow us to write asynchronous code that looks and behaves more like synchronous code.

async function

- A function declared with the async keyword.
- Always returns a promise, regardless of whether the code explicitly returns a value.
- The returned promises resolves with the value returned by the function or rejects if an error is thrown.

```
async functionName() {
    // some code
}
```

await keyword

- can only be used inside an async function.
- pauses the execution of the async function until the promise resolves or rejects.
- The value of the resolved promise is returned, and if the promise rejects, the await expressions throws the rejected value as an error.

```
const result = await somePromise;

placeOrder (cart)
  .then (order => preparingOrder (order))
  .then (foodDetails => pickupOrder (foodDetails))
  .then (dropLocation => deliverOrder (dropLocation))
  .catch (error => console.log (error));
```

```
const order = placeOrder(cart);
const foodDetails = preparingOrder(order);
const droplocation = pickupOrder(foodDetails);
deliverOrder(droplocation);
```

This will run immediately with any value of the arguments.

But we want until we don't get any data in the order variable don't execute preparingOrder(order) and same goes on.....

```
async function greet(){          → it is doing the work of .then
    const order = await placeOrder(cart);
    const foodDetails = await preparingOrder(order);
    const droplocation = await pickupOrder(foodDetails);
    deliverOrder(droplocation);
}
greet();
```

Now, until we don't get any values in order variable, next line doesn't execute.

```
const p1 = new promise((resolve, reject) => {
                setTimeout(() => {
                    resolve("Hello Everyone");
                }, 5000);
        });

async function greet(){
    const data1 = await p1;
    console.log(data1);

    const data2 = await p1;
    console.log(data2);
}
```

Here, for data1 await waits for p1 to be completed but for data2 it doesnot wait, because the p1 is already resolved above.

```
const p1 = new promise ((resolve, reject) => {
        setTimeout (() => {
            resolve("first promise resolved");
        }, 5000);
    });

const p2= new promise ((resolve, reject) => {
        setTimeout (() => {
            resolve("second promise resolved");
        }, 5000);
    });

async function greet() {
        const data1 = await p1;
        console.log(data1);

        const data2 = await p2;
        console.log(data2);

    }
    greet();
```

Here, for data1 await waits for p1 to be resolved, after p1 is resolved, it prints data1.

But till that time p2 is also resolved, as p1 and p2 both are taking 5 sec.

greet() prints data1 and data2 immediately, after 5 sec.

```
const p1 = new promise ((resolve, reject) => {
        setTimeout (() => {
            resolve("first promise resolved");
        }, 8000);
    });

const p2= new promise ((resolve, reject) => {
        setTimeout (() => {
            resolve("second promise resolved");
        }, 5000);
    });

async function greet() {
        const data1 = await p1;
        console.log(data1);

        const data2 = await p2;
        console.log(data2);

    }
    greet();
```

Here, first it wait for 8sec, so that p1 is resolved, then it goes to p2, but till that time p2 is already resolved as it is taking 5 sec.

So, gt prints data1, data2 immediately after 8 sec.

```
p1.then (value => console.log (value));
p2.then (value => console.log (value));
```

Here, p2 is resolved first then p1 is resolved, as p2 is taking 5 sec and p1 is 8 sec.

But in await, the it wait for p1 to be resolved (it waits for 8 sec) then logs both (data1, data2) immediately.

```
function test1() {
    const p1 = new promise ((resolve, reject) => {
        setTimeout ( () => {
            resolve("First promise resolved");
        }, 5000);
    })
    return p1;
}
function test2() {
    const p2= new promise ((resolve, reject) => {
        setTimeout ( () => {
            resolve("Second promise resolved");
        }, 5000);
    })
    return p2;
}
async function greet() {
    const data1 = await test1();
    console.log(data1);

    const data2 = await test2();
    console.log(data2);
}
greet();
```

Here, First it wait for 5 sec for p1, and then again wait for 5 sec for p2.

This is why, because until a function is not called, it can't execute.

try and catch

```
async function greet() {
    try {
        const data1 = await test1();
        console.log(data1);

        const data2 = await test2();
        console.log(data2);
    }
    catch(error) {
        console.log(error);
    }
}
greet();
```

Till now, until test1() is not resolved, test2() is not executed.

We can execute these promises in parallel.

```
async function greet() {
    try {
        console.log("Hello I greet you");
        const [data1, data2] = await Promise.all([test1(), test2()]);
        console.log(data1);
        console.log(data2);
    }
    catch(error) {
        console.log(error);
    }
}
greet();
```

Here, it will run after the max(test1(), test2()) the promise takes.