JavaScript Basics Notes

CodeHelp Dot Batch

JavaScript

→ JS is a light weight programming language.
→ It is a scripting language.
(JS kya h & kese krna h.)

⇒ Initially it was created for client-side scripting (Browser) with time,

JS + C++ ⟹ Node

Now, it can be used for server side scripting (outside the browser)

⇒ For C++, Java, etc, we need compiler to run. In similar way, we use JS Engine to run javascript code.

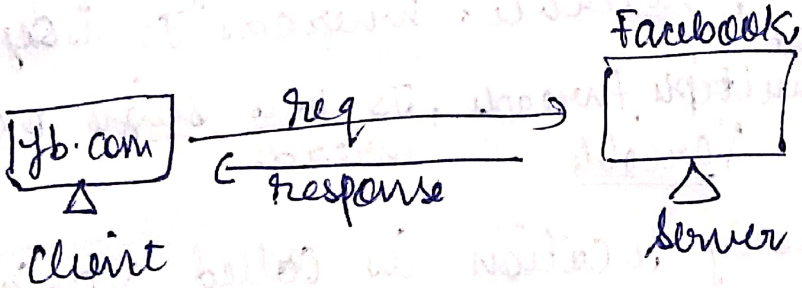
eg: Firefox JS Engine is Spider Monkey
Chrome JS Engine is V8.

⇒ Use of JS:→

- to make web apps
- mobile apps
- CLI tools
- Network apps
- Games

HTML → Structure
CSS → Styling
JS → Functionality

Client-Server



⇒ Adding JS in code
using <script> tag.

Example:

```
<html>
  <head>
  </head>
  <body>
    <script src="index.js"></script>
  </body>
</html>
```

can be added in head as well as body last.

Running JS Code

Way (1) Console
Way (2) Terminal ⇒ node index.js

How to setup JS?

(1) VS Code
(2) Install Node JS (Optional)

HW Java Vs Javascript

Similarities →

(1) Both Java & JS are OOP languages.
(2) Both can be used to develop frontend
(3) both can be used to develop backend.

Differences:

(1) Java is compiled language & JS is inpreted by browser.
(2) Java uses static type checking. Programmer must specify type of variable. whereas JS is dynamically typed language.
(3) Java uses multiple threads, JS uses single main thread.

Variable

→ Named memory location is called Variable.

How to create variable??

- using var
- using let
- using const
- We can also declare a variable without any keyword.

eg → var x = 5;
let y = 6;
z = 10;

let name = 'Babbar';
let b = 12.5;

When to use const?
When we declare any value with "const" keyword, then it means that these value are constant & cannot be changed.

```
const price = 5;
```

Price = 6; ⇒ error → Assignment to constant value

Literals

let a = 5; → Number literal
let name = "Rishabh"; → string literal
let status = true; → Boolean literal

Variable Naming Rules

⇒ cannot be a reserved keyword
⇒ cannot start with number.
⇒ cannot contain space or '-'
⇒ should be meaningful
⇒ should use camelcase

Primitive Types

In javascript, there are seven primitive data types:

- string → 'Love Babbar'
- number → 1, 2, 3, 4, — — 1.23, 12.85
- bigint
- boolean → True or False
- undefined → let a; console.log(a);
- symbol
- null → empty value eg → let a = null;

Dynamic Typing

JavaScript has dynamic types. This means that same variable can be used to hold different data types.

eg
```
let x;      // undefined
x = 5;      // Number
x = 'Akansha'; // String
```

Reference Types

* Objects * Arrays * Function

① Object → Real World Entity having some properties & behaviour.
↳ Multiple variables linked together.

eg
```
let person = {
    firstName: 'Love';
    age : 24
};
```
Key Value Pairs

Object Syntax

Access = ??
person.age
or
person['age']

(2) Array

D.S used to store a list of items.

```
let names = ['love', 'Rahul', 'Vijay']
```

Accessing →
```
names[0]
names[1] = 'Akansha';
names[2]
names[3] = 'Ramesh';
```
→ can replace like this
→ Will Add this

let Vs var

```
{
    let a = 5;
}
console.log(a); → will Not Print
```

Scope: Local Scope

Var.

Global Scope
Function Scope

let vs var

var	let
(1). The variable defined with var statement have function scope.	(1) The variable defined with let statement have block scope.
(2) We can declare a variable again even if it has been defined previously in same scope.	(2) We cannot declare variable more than once if we defined that previously in same scope.
(3). Hoisting is allowed with var.	(3). Hoisting is not allowed with let.

Operators

① Arithmetic → +, *, /, %, ** ⇒ exponentiation
x ** y ⇒ x^y

② Assignment → num += 2

③ Comparision → >, <, >=, <=, ===, !==

④ Bitwise → Bitwise AND, Bitwise OR, Right shift, left shift.

⑤ Logical → AND (&&), OR (||), NOT (!)

⑥ Pre / Post Inc / Dec

Equality Operator

Strict Equality → ===

type → Data → Same
Value → Same

```
let num = 1;
let str = '1';
1 === 1   False
```

Loose Equality → ==

→ Checks only value

```
let num = 1;
let str = '1';   num == str; ⇒ True
```

Ternary Operator

condition ? val1 : val2;

eg. let status = (age >= 18) ? 'I can vote' : 'can't vote';

Working With Non Booleans

Falsy
- undefined
- null
- 0
- false
- NaN

Truthy
- Anything that is not falsy.

Control Statements

- If - else
- switch - case

Loops →

(1) For loop
(2) While loop
(3) do - while loop
(4) For - in loop
(5) For - of loop

```
console.log(false || 'Babbar')
→ 'Babbar'
console.log(false || 5 || 1)
↳ 5
```

Short Circuit

```
console.log('Love' && 'Babbar')
↳ 'Babbar'
```

Day → II JavaScript Basics

Object in JS

- Various variables linked together
- Real World Entity
- With properties & behaviour

```
let circle = {
    radius : 1,
    draw()
}
```

```
Human
  ↳ noOfHands
  ↳ noOfLegs
  ↳ Eyes
```

```
Rectangle
  ↳ length = 1
  ↳ breadth = 2
```

⇓

```
let rectangle = {
    length = 1,
    breadth = 2
};
```

Object →

→ Collection of key - value pairs

```
let a = {};
```
⇒ Empty Object

Creating an Object

```
let rectangle = {
    length: 1,
    breadth: 2,
};
```
} Properties

// a function created inside object is known as method

```
draw: function () {
    console.log ('let's draw');
}
};
```
} Behaviour

If I want to create multiple rectangles, then if I copy this code again & again, code will be very bulky.
So, we will use →

Object Creation By →

* Factory Function
* Constructor Function

In Console

1) rectangle
{length: 1, breadth: 2, draw: f}
2) rectangle.breadth
2
3) rectangle.draw
f () {
console.log ("Creating Rectangle")
}
4) console.log (rectangle);
{length: 1, breadth: 2, draw: f}
5) rectangle.draw ();
Creating Rectangle
// function call ho gaya.

* Factory Function

eg (1)

```
function createRectangle() {
    return rectangle = {
        length: 1,
        breadth: 2,
        draw: function() {
            console.log ('Drawing Rectangle');
        }
    };
}
```

→ It is hardcoded

Calling →

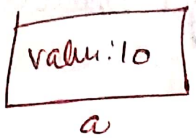
let rectangleObj1 = createRectangle();

---

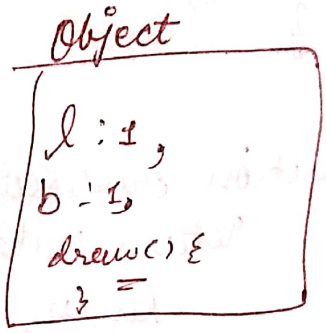
let a = 5; → 5 (a)

let a = {}; → Object (a)

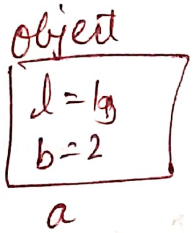
let a = {
  value: 10
};



let a = {
  l: 1,
  b: 1,
  draw: function() {
    console.log("Draw");
  }
};

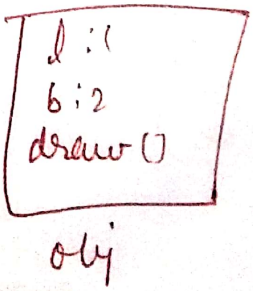


let a = {
  l: 1,
  b: 2
};



By Function Calling

Request Object Creation → Factory Function → Object



let obj = factoryFunction();

Function Calling

Creating Customized Object Using Factory Function

```
function createRectangle ( len, bre) {
    return  rectangle = {
            length : len,
            breadth : bre,
            draw() {
               console.log ('Drawing Rectangle');
               }
        };
}
```

OR

```
function creatRectangle (length, breadth) {
        return rectangle = {
            length,
            breadth,
            draw() {
                ==
            }
        };
}
```

Calling →

```
let rectangleObj1 = creatRectangle (4, 5);
let rectangle2 = creatRectangle (2,3);
let rectangle3 = creatRectangle (7, 8);
```

II Way : Constructor Function

→ We use Pascal Notation For creating Constructor

→ Constructor Function "initializes / define properties & methods.

```
function Rectangle() {
    this.length = 4;
    this.breadth = 2;
    this.draw = function() {
        console.log('drawing');
    }
}
// Object creation using constructor
let rectangleObj = new Rectangle();
```

Initialize ho rha h

// new keyword returns empty object in JS.

Customized Object

```
function Rectangle(len, bre) {
    this.length = len;
    this.breadth = bre;
    this.draw = function() {
        console.log('drawing');
    }
}
// Creating Rectangle
let rectangleObj = new Rectangle(4,6);
```


→ this → refers to current object

→ new → creates empty object

Dynamic Nature Of Objects

```
let a = 5;
a = 'babbar';
```

We can add & remove properties of object.

eg ⇒
```
function Rectangle (len, bre){
    this.length = len;
    this.breadth = bre;
    this.draw = function(){
        console.log('drawing');
    }
}
let rectangleObject = new Rectangle (4,6);
```

O/P → rectangleObject
Rectangle { length: 4, breadth: 6, draw: f }

② `rectangleObject.color = 'yellow';`

O/P → rectangleObject
Rectangle { length: 4, breadth: 6, color: 'Yellow', draw: f }

③.
```
delete rectangleObject.color;
console.log(rectangleObject);
```

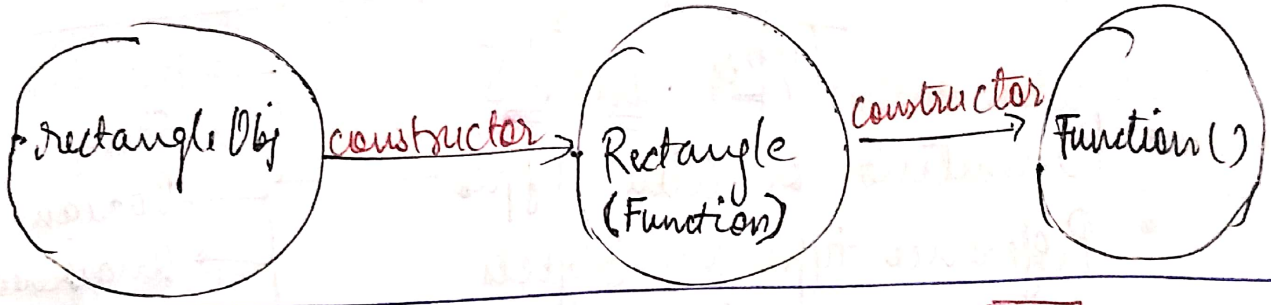
O/P → Rectangle { length: 4, breadth: 6, draw: f }

Constructor Properties

* Every object has a constructor.
* Every Function is also an object.
* So, Every constructor / Function also has a Constructor.

rectangleObj = new Rectangle()

Rectangle ka constructor → ƒ Function() { [native code] }



let Rectangle = new Function('length', 'breadth',

Entire Code

);

Internally ye hota h!!

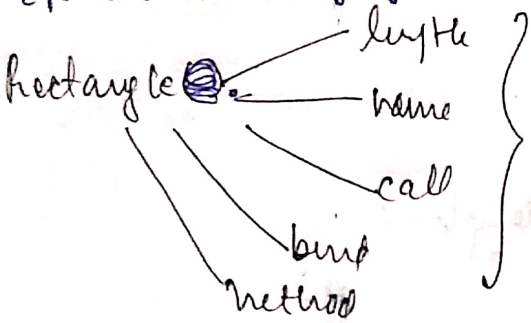
let Obj = new Rectangle(2,3);

rectangleObj.constructor ⟶ Rectangle
Rectangle.constructor ⟶ Function

(II) Functions are Objects

What is Object ? → which is having properties & behaviour.

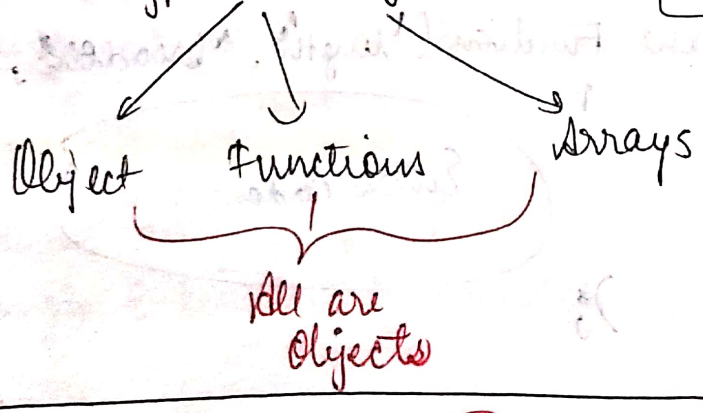
So,
Function has also properties & behaviour. And we can use '.' operator in any function & we will see many properties of it.



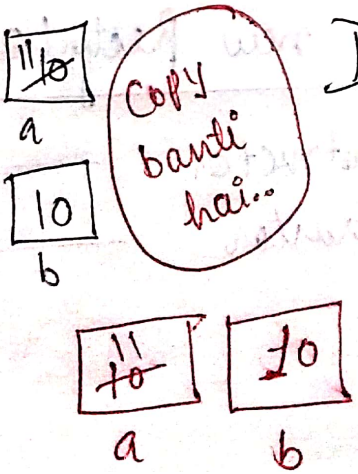
⇒ Functions are Objects. ☺

Types in JS

- Primitive or Value Types → Number, String, Boolean, Undefined, Null
- Reference types or Objects



```
let a = 10;
let b = a;
a++;
print (a) → 11
print (b) → 10
```

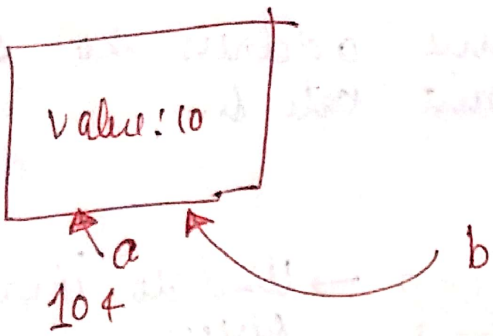


Difference between Primitives & References

Reference Type

```
let a = {value:10};
let b = a;
a.value++;
print (a.value);   ⟶ 11
print (b.value);   ⟶ 11
```

⟹ Points to the same address

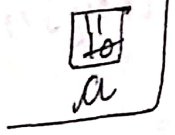


Note :→ Primitives are copied by their value.
References are copied by ~~reference~~ address.

---

eg (2)

```
let a=10;
function inc (a)
{
  a++;
}
inc (a);
console.log (a);
```



[this is Pass by value ?

op→ 10
=

```
let a = { value: 10 };
inc(a);
console.log(a)
```

```
function inc(a)
{
a.value++;
}
```

OP → 11

Obj → Function call me same address ko same/different name se point kate h.

For-in loop → Used to iterate over object.

Accessing
↳ key
↳ value
↳ key-values

```
let rectangle = {
    length: 2,
    breadth: 4
};
// For-in loop
for (let key in rectangle) {
    console.log(key, rectangle[key]);
}
```

OP length 2
breadth 4

// keys are reflected through key variable
// values are reflected through rectangle[key]

For-of : used to iterate over iterables.
- Arrays
- Maps

* Objects are not iterable.

⇒ However we can iterate on keys of object using hack.

```
// For-of loop on keys of Object:
for (let key of Object.keys(rectangle)) {
    console.log(key);
}
```

o/p → length
breadth

```
// For-of loop on key value-pair (entries) of Object:
for (let key of Object.entries(rectangle)) {
    console.log[key];
}
```

o/p
['length', 2]
['breadth', 4]

How to check whether a property belongs to an Object or not?

```
let Rectangle = {
    length: 2,
    breadth: 4
};
```

radius
Color
length
breadth

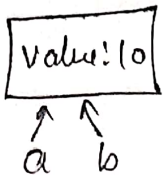
Exist or not?

```
if ('color' in rectangle) {
    console.log('Present');
}
else
    console.log('Absent');
```

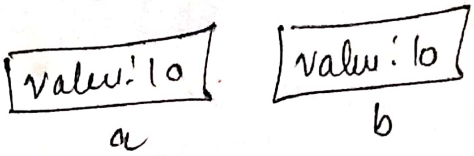
o/p → Absent

Object Cloning

```
let a = {
  value: 10
}
b = a;
```



this is Not cloning



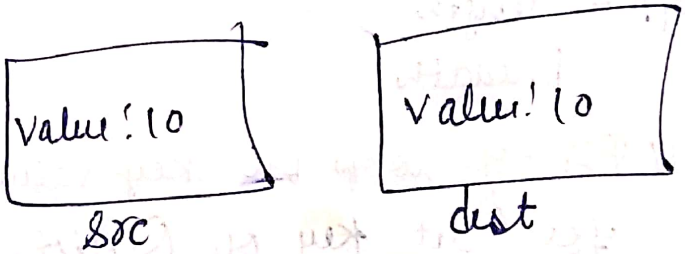
This is object cloning

It is used to create a copy of original object with same properties included in it.

```
for (let key in Rectangle)
  {
  consol.log (key, Rectangle[key]);
  }
```

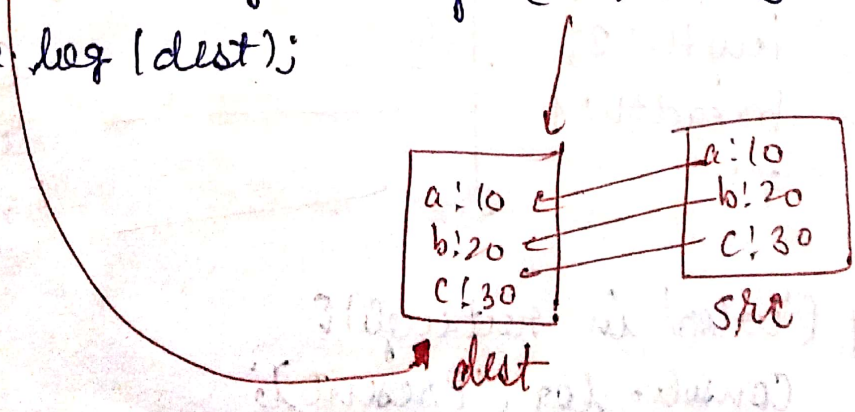
(I) Iteration

```
let src = {
  value: 10
};
let dest = {};
for (let key in src) {
  dest[key] = src[key];
}
```



(II). Assign

```
let dest = Object.assign ({}, src);
consol.log (dest);
```



(III) Spread

```
let dest = {...src};
consol.log(dest);
```

Garbage Collection in JS

↳ It is used to deallocate the memory of variables which are not in use.
↳ It runs automatically in background.
↳ We have no control over it, we can not control start / end of it.

→ We can also assign multiple objects into one object.

```
let src = {
    a:10,
    b:20,
    c:30
};
```

```
let src2 = {
    value:25
};
```

(i) `let dest = Object.assign({}, src, src2);`
`consol.log(dest);`

O/P { a:10, b:20, c:30, value:25 }

(ii) `let dest = {...src, ...src2}`

(iii)
```
for(let key in src){
    dest[key] = src[key];
}
for(let key in src2){
    dest[key] = src2[key];
}
```

Lecture → 3

Inbuilt Objects & Array

(1) Math Object :→ Math is an built-in object that has properties & methods for mathematical constants & function.

* Math properties → The syntax is : Math.property

eg Math.E
Math.PI; → 3.1415...
Math.SQRT2 → value of √2

* Math Methods →

(i) Math.round(X) Math.round(1.8) = 2
Math.round(1.2) = 1

(ii) Math.ceil(X) → round up

(iii) Math.floor(X) → round down

(iv) Math.trunc(X) → Returns integer part

(v) Math.pow(X,Y)

(vi) Math.sqrt(X)

Math.random();
0.90295 79675219606
Math.random();
0.11330611 39000 7084
generates random number everytime.

(vii) Math.abs(X) → absolute (true) value
Math.abs(-2) = 2 | Math.abs(2) = 2

(viii) Math.min(--) → lowest value in list of arguments
Math.min(3,2,1) = 1

(iv) Math.max(_ _) → Returns highest value in list of arguments.

eg → Math.max(2, 1, 4, 3) = 4

(II) String

String can be created as Primitive form (String literal) or as, Object using String() constructor.

Literal →

```
const str1 = 'Akanksha Singhal';
const str2 = "Akanksha Singhal";
const str3 = `Akanksha Singhal`;
```

```
typeof (str1)
String
typeof (str4)
Object
```

String Object →

```
const str4 = new String ('Akanksha Singhal');
```

⇒ We can convert primitive string into Object by using . notation. eg → lastName.length → 6

⇒ A string object can also be converted into primitive with valueOf() method.

String Method

(i) length

lastName.length = 6

(ii) substring(start, end)

firstName.substring(0,3) → 'Aka'

lastName.indexOf('B'); → 0

(iii) substr(start, end)

(iv) slice(start, end)

firstName.slice(0,3) → 'Aka'

(v). replace()
lastName.replace('Babb','car');
Carar

(vi) toUpperCase()
lastName.toUpperCase();
BABBAR

(vii) toLowerCase()
lastName.toLowerCase();
babbar

(viii) concat()

(ix) trim()
lastName = " Babbar ";
lastName.trim();
'Babbar'

(x) trimStart(), (xi) trimEnd()

lastName[0]
B
lastName.includes('Ba');
True
lastName.includes('Dav');
False
lastName.startsWith('Babb');
True
lastName.endsWith('ar');
True

~~lastName~~
let message = 'This is my message';
let words = message.split(' ');
['This', 'is', 'my', 'message'].
Array(4)

(III) Template Literals

It uses backticks (` `) rather than quotes (' ') or (" ") to define a string.

eg: let text = `Hello World!`;

⇒ You can use both single & double quotes inside string:

eg let text = `He's Often called "Johny"`;

⇒ Template literal allows multiline string. They will be printed as it is.

eg let text =
`This is
the
best news
ever!`;

(IV) Interpolation

Interpolation literals provide an easy way to interpolate variables & expression into string.

* Variable substitution ⇒ Template literals allows variables in string.

```
eg  let fname = "Akansha";
    let lname = "Singhal";
    let text = `Welcome ${fname}, ${lname}!`;
```

o/p Welcome Akansha Singhal!

(${fname} → Placeholder)

---

* Escape Sequence Notation

\0	null character
\'	Single Quote
\"	Double Quote
\\	Back slash
\n	New line
\t	Tab space

(V) Date Objects

→ Date objects are created with new Date() constructor.
→ There are 9 ways to create new date object.

(1) new Date() → Displays current date & time.
(2) new Date (date string)
    Kisi bhi form me de sakte h.
(3) new Date (year, month)
(4) new Date (year, month, day);
(5) new Date (year, month, day, hours);
(6) new Date (year, month, day, hour, minutes);
(7) new Date (year, month, day, hour, minutes, seconds);
(8) new Date (year, month, day, hour, minute, seconds, ms);
(9) new Date (milliseconds)

eg → let date = new Date(); → Shows Current Date & Time
console.log(date);

Tue Mar 07 2023 15:33:05 GMT+0530 (India Standard Time)

let date2 = new Date('Jun 20 1998 06:15');
console.log(date2);

Sat Jun 20 1998 06:15:00 GMT+0530 (India Standard Time)

let date3 = new Date(2001, 1, 17, 4, 30);
console.log(date3);

Mon Feb 17 2001 04:30:00 GMT+0530 (India Standard Time)

getter & setter Methods

(i) date3.setFullYear('1947');
console.log(date3);

Mon Feb 17 1947 04:30:00 GMT+0530 (India Standard Time)

date.setDate()
- setHours()
- setMinutes()
- setSeconds()
- setMonth()
- setTime()

We can set these values.

getters

console.log(date3.getMonth());
- .getDate()
- .getHours()

We can get values by getter functions.

Arrays

→ Object / Reference type
→ collection of different type of items

(1) Creating an array :→

let numbers = [1, 2, 3, 5];

(2) Insertion in array :→

(i) end → numbers.push(9);
[1, 2, 3, 5, 9];

(ii) begin → numbers.unshift(6);
[6, 1, 2, 3, 5, 9];

(iii) Middle → splice()

numbers.splice(2, 0, 'a', 'b', 'c');

- 2 → at index
- 0 → Delete count
- 'a', 'b', 'c' → to be inserted

['6', 1, 'a', 'b', 'c', 2, 3, 5, 9];

(3) Searching in Array :→

↳ For Primitives → indexOf
→ includes

eg = console.log(numbers.indexOf(4));
-1

console.log(numbers.indexOf(5));
2

console.log(numbers.includes(7));
false

numbers.indexOf(4, 2);
=

- 4 → search element
- 2 → search start from index

Searching in Array of Objects

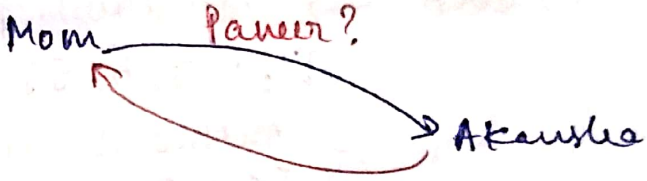
```
let courses = [ {no: 1, name: 'Love'},
                {no: 2, name: "Rahul"} ];
```

These references are different

```
console.log(courses.includes({no: 1, name: 'Love'}));
```

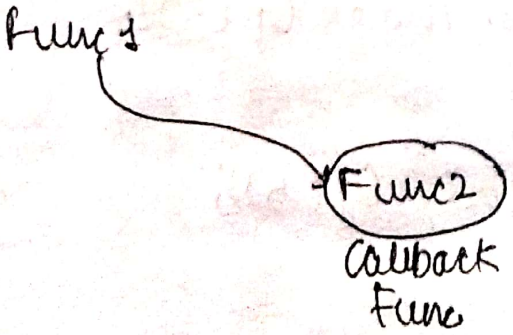
Ans = False

For searching in array of objects, we use callback functions.



Call Back Func:

* callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of action.




```
return Object  <-  arrayName.find( )
```

Callback function or Predicate Function

```
let course = courses.find(function(course){
    return course.name == 'Love';
});
console.log(course);
```

Predicate Func Returning an Object

```
function(course)
{
return course.naam == 'Love';
}
```

जैसा course return करो जिसका naam Love हो..

We can make this code more readable by converting it to Arrow function

```
let course = courses.find(course => course.naam == 'Love')
console.log(course);
```

```
let course = courses.find
    (function (course) =>
    return course.naam == 'Love')
});
```

```
let course =
courses.find(course =>
course.naam = 'Love');
```

We have to find a course with name 'Love'.

```
let course = courses.find(Predicate Function);
```

Predicate Function

```
function(course){
 return course.naam == 'Love';
}
```

Removing Element

numbers = [1, 2, 3, 4, 5, 6];

(i) end → pop()

numbers.pop(); ⇒ numbers = [1, 2, 3, 4, 5];

(ii) beginning → shift()

numbers.shift(); ⇒ numbers = [2, 3, 4, 5];

(iii) middle → splice(3, ——);

3 → index

no of element you want to delete)?

numbers.slice(2, 1);

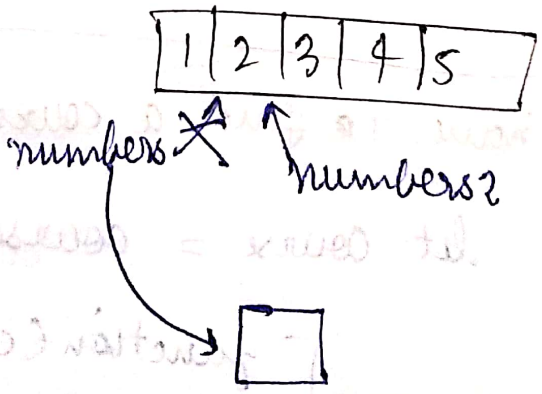
= [2, 3, 5];

Emptying an Array

numbers = [1, 2, 3, 4, 5]

numbers = [] → Automatically removed by garbage collector

```
let numbers = [1,2,3,4,5];
let numbers2 = numbers;
numbers = [];
Console.log(numbers);
Console.log(numbers2);
```



O/P
[]
[1,2,3,4,5].

So, only this does not delete the array.

So, best practice is :→

```
let numbers = [1,2,3,4,5];
let numbers2 = numbers;
numbers.length = 0;
console.log(numbers);
console.log(numbers2);
```

Best Way

Op [ ]
[ ]

Third Way :→ numbers.splice(0, numbers.length);

II way →

```
while (numbers.length > 0)
{
    numbers.pop();
}
```

Combining Arrays

```
let first = [1,2,3,4];
let second = [5,6];
let combined = first.concat(second);
```

Op → [1,2,3,4,5,6]

Slicing Arrays

slice (x, y)
x → included, y → excluded

```
let first = [1,2,3,4,5,6];
let sliced = first.slice(2,4);
console.log(sliced);
```

Op → [3,4]

is index se ek kam tak aayega..

(ii) let sliced = marks.slice(2); → 2 index se end tak
(iii) let sliced = marks.slice(); → same array copy

HW → combining & slicing of objects

Spread Operator

(i) Combining Arrays →

(a) let first = [1,2,3]
let second = [4,5,6]
let combind = [... first, ... second];

(b) let combined = [... first, 'a', 'b', ... second, true];

O/P → [1,2,3,'a','b',4,5,6,true];

(ii) Object Cloning :→

let another = [... combined];

Iterating an Array

for-of loop :→

```
let arr = [1,2,3,4];
for (let value of arr) {
    consol.log(value);
}
```

(iii) For-each loop

arrayName.forEach(callbackFunction);

```
arr.forEach(function(number){
    console.log(number);
});
```

Converting it into arrow function →

```
arr.forEach(number => console.log(number));
```

Joining Arrays to String

```
let numbers = [10,20,30,40,50];
let joined = numbers.join(',');
console.log(joined);
```

O/P 10,20,30,40,50

Split String to Array

```
let message = 'This is my first message';
let parts = message.split(' ');
console.log(parts);
```

O/P ['This', 'is', 'my', 'first', 'message']

```
// Joining it again
let joined = parts.join('_');
console.log(joined);
```

O/P This_is_my_first_message

Sorting Arrays

4	3	1	5	2

(i) let numbers = [40, 30, 10, 20, 50];
numbers.sort();
console.log(numbers);

O/P → [10, 20, 30, 40, 50]

(ii) let arr = [40, 4, 5, 1]
arr.sort();
console.log(arr);

O/P [1, 4, 40, 5]

Internally, string ki tarah leta h.

Reverse Array

(b) let arr = [10, 20, 30];
arr.reverse();
console.log(arr);

O/P → [30, 20, 10]

Filtering Array

numbers.filter(↓) ← callback Function

eg let numbers = [1, 2, -1, -4];
let filtered = numbers.filter(function(value){
return value >= 0;
});
console.log(filtered);

O/P [1, 2]

Converting into arrow function

let filtered = numbers.filter (value => value >=0);

Mapping Arrays

→ maps each element of array to something else.

eg let numbers = [7, 8, 9, 10];

let items = numbers.map(function(value){
return 'student_no' + value;
});

console.log(items);

Op ['student_no7', 'student_no8', 'student_no9', 'student_no10']

ASCII table
'a' → 97
'b' → 98
'c' → 99
'd' → 100

Arrow Function

let items = numbers.map(value => 'student_no' + value);

Mapping Objects (Converting Array to Items Object)

(1) let numbers = [1, 2, 3, -1, -6]

let filtered = numbers.filter(val => val >=0);

→ [1, 2, 3]

let items = filtered.map(function(num){
return {value: num};
});

items 0 → {value: 1}
1 → {value: 2}
2 → {value: 3}

Array

Converting this code into arrow function →

```
let items = filtered.map(num => {value: num});
console.log(items);
```

Chaining of Methods

Initially →

```
let numbers = [1,2,3,-1,-6];
let filtered = numbers.filter(value => value >=0);
let items = filtered.map(num => {value: num});
```

Using Chaning →

```
let items = numbers.filter(val => val >=0).map(num
                                    => {value: num});
```

Lecture → 4

Functions

→ A block of code that fulfills a specific task.

Syntax :

```
function functionName(Parameters){   // I/P
    Body
}
```

Why Functions ??

① Reusability
② Readability
③ Easy to debug

Function Declaration :→

(i) function run() { → Function Name, Parameters

console.log("Running");] Body

}

Function Call

run();

Hoisting

Hoisting is the process of moving function declaration at the top of file.

It is done automatically by JS Engine.

```
run();
function run() {
  console.log("run");
}
```

o/p → Run

Function Assignment

(i) Named Function assignment

```
let stand = function walk() {
    console.log('walking');
}
stand(); ✓
walk(); // can't be called now.
```

```
let jump = stand;
jump(); ✓
```

(ii) Anonymous Function Assignment

```
let stand = function() {
    console.log('Walking');
}
```

let a = function b() { } // Named	let = function() { } // Anonymous

Calling Function Assignment

stand(); ✓

walk(); ✗

If we call the walk() after assignment, it will not be called, we can call it only by stand();.

Can we call the function before function assignment ??

No, because hoisting works only with function declaration, not with function assignment.

JS is a dynamic language

```
let x = 1;
x = 'a';        // Perfect
```

```
function sum (a, b){
    return a+b;
}
```

(i) console.log (sum (1, 2)); O/P → 3

(ii) console.log (sum (1)); → O/P → NaN

a b
↓ ↓
1 + undefined = NaN

(iii) console.log (sum());

a b
↓ ↓
undefined + undefined = NaN

O/P → NaN

(iv) console.log (sum (1, 2, 3, 4));

O/P → 3

a b
↓ ↓
1 + 2 = 3

Arguments

Arguments are special Objects

(i)

```
function sum(a,b){
 console.log(arguments);
 return a+b;
}

sum(1,2);
```

O/P

[Arguments(2) [1,2]
0: 1
1: 2
Object

(ii)

```
function sum(a,b){
 console.log(arguments);
 return a+b;
}

sum(1,2,3,4,5);
```

O/P

Arguments(5) [1,2,3,4,5]
0: 1
1: 2
2: 3
3: 4
4: 5
Key value

Lets create a dynamic function that can add any number of arguments..

```
function sum(){ let total=0;
 for(let value of arguments)
    total += value;
return total;
}

console.log(sum(1,2,3,4,5,6));
```

O/P →

21

Rest Operator →...

↳ Used for concatenate Arrays
↳ to copy one array to other

→ Also known as spread operator.

* If in a function, we have multiple parameters, then we can handle all the parameters with the help of rest operator.

* we can store these parameter in form of array, using rest operator.

eg ① function sum (... args) {
  console.log (args);
}
sum (1, 2, 3, 4, 5, 6);

O/P →
0 : 1
1 : 2
2 : 3
3 : 4
4 : 5
5 : 6
Array

eg ② function sum (a, b, ... args) {
  console.log (args);
}
sum (1, 2, 3, 4, 5, 6);
a b args

O/P →
0 : 3
1 : 4
2 : 5
3 : 6

eg ③ function sum (a, b, ... args, c);

This is wrong.

Rest operator should be the last parameter.

Default Parameters

(a)
```
function interest (p, r, t) {
    return p * r * t / 100;
}
```

(i) `console.log (interest (1000, 10, 5));` O/P → 500

(ii) `console.log (interest (1000, 10));` O/P → NaN

(iii) `console.log (interest (10, 5));` O/P → NaN

(b)
```
function interest (p, r = 6, t = 9) {
    return p * r * t / 100;
}
console.log (interest (1000, undefined, 8));
```

O/P
540

p = 1000
r = 6
t = 8

---

* When a user does not pass any value to function the function will take default value.

→ In above case, for r = 6 is default Parameter.

So, if you pass a value to r, then it will take the passed value. Otherwise, it will take default value.

Note →

If 'r' is a default parameter, then all the parameter to right side of it must also be default. (It is a rule).

Getter & Setter Methods

Getter → to access the properties
Setter → to change the properties

```
let person={
    fName = 'Love',
    lName = 'Babbar'
```

getter & sitters are now not functions. They are properties

Getter:
```
get fullName(){
    return `${this.fName} ${this.lName}`;
},
```

Setter:
```
set fullName(value){
    let parts = value.split(' ');
this.fName= parts[0];
this.lName = parts[1];
}
}
```

Getter & Setter are now not functions. These are treated as properties.

Calling →

getter calling → `console.log(person.fullName);`
o/p → Love Babbar

setter calling →
```
person.fullName = 'Akansha Singhal';
console.log(person.fullName);
```
o/p Akansha Singhal

Error Handling

```
try {
person.fullName = true;
}
catch (e) {
  alert (e);
}
```

```
set fullName (value) {
  if (typeof value != String)
  throw new Error ('Not a string');
  let parts = value.split (' ');
  this.fName = parts[0];
  this.lName = parts[1];
}
```

error

alert
Not a String

O/P

Without throw →

```
try {
person.fullName = true;
}
catch (e) {
  alert ('You have not sent
        a string');
}
```

```
set fullName (value) {
  let parts = value.split (' ');
  this.fName = parts[0];
  this.lName = parts[1];
}
```

alert
you have not sent
a String

Try → The try statement defines a code block which can generate error.

Catch → The catch statement defines the code to handle error.

Finally → Finally defines code block to run regardless of the result.

Throw → The throw statement defines a custom error.

Scope

let is block scoped.

Var is function scoped.

NOfunction ⇒ Global Scoped

Creates Variable

Creates constants

Const is block scoped.

eg ①
```
{
  let a = 5;
}
console.log(a);
```
O/P → error : a is undefined

eg ②
```
{
  var a = 5;
}
console.log(a);
```
O/P → 5

eg ③
```
if (true){
  let a = 5;
}
console.log(a);
```
O/P → error

eg ④
```
function f(){
  var a = 5;
}
console.log(a);
```
O/P → error: a is not defined

eg ⑤
```
for (var i=0; i<5; i++)
{
  console.log(i);
}
console.log(i);
```
O/P → 5

eg ⑥
```
function a(){
  const ab = 5;
}
function b(){
  const ab = 5;
}
```
✓ ✓

Reducing An Array

The reduce() method executes a reducer function for an array.

Using Loop

```
let arr = [1,2,3,4,5];
let total = 0;
for (let value of arr) {
    total += value;
}
console.log(total);
```

O/P → 15

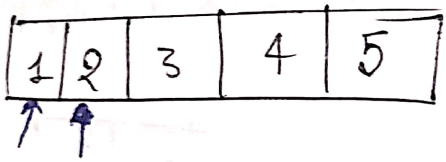
```
arr.reduce(callback Function, 0);
```

Initialize Accumulator

That total variable

accumulator = 0;

currentValue → iterating the array.




```
arr.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
```

accumulator = 0
CurrentValue = 1

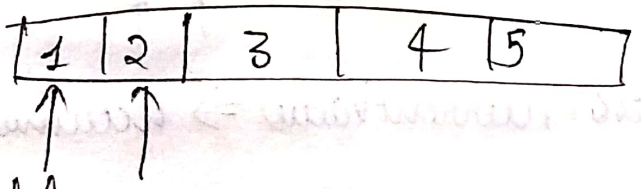
accumulator = accumulator + currentValue = 0 + 1 = 1
CurrentValue = 2
acc = acc + currentVal = 1 + 2 = 3
currentVal = 3
acc = acc + currentVal = 3 + 3 = 6

So final code is

```
let arr = [1,2,3,4,5];
let sum = arr.reduce((accumulator, currentValue)
        => accumulator + currentValue, 0);
console.log(sum);
```

* accumulator is like total variable that store sum.
* current value is iterating the array.

Note: What will happen if I dont initialize the accumulator?



→ Accumulator will take first value
&
→ currentValue will start from second value.

```
Const output = arr.reduce(function(acc, cur){
                acc = acc + cur;
                return acc;
            }, 0);
```