Page No.
Date

# * JAVA SCRIPT :-

- JavaScript is a client side scripting language.
- It is used to make web pages alive.
- It is used to programmatically perform actions within the page.
- When JavaScript was created, it was initially called "LiveScript".
- But Java was a very popular language at that time, so it was decided that positioning a language as a "younger brother" of Java would help.

## * What can Js do :-

- Js can execute not only in the browser, but also on the server.
- ~~We will use Js~~
- Js has evolved greatly as a language and is now used to perform a wide variety of tasks.

* ECMAScript :- ECMAScript is a standard on which JavaScript is based.
It was created to ensure that different documents on JavaScript are talking about the same language

- ES1 was first released in 1997.
- ES6 was released in 2015
- latest version of JavaScript is ES13 in 2022.

Page No.
Date

* Different methods to run JavaScript :-

1) In-browser.
2) inside <script> tag.
3) using node js.

* What can In-Browser Js Do :-

1) Js can add new HTML & change existing HTML from DOM.
2) It can even react to any event (actions)
3) It can also manage the AJAX requests (GET or POST request)
4) JavaScript can get and set cookies and use local storage.

* What Can't In-Browser Js Do :-

1) Js cannot read or write to and from computer hard disk without user permissions.
2) The browser does not allow the Js of any website to collect the AJAX information of the other website because it generates the error of same origin policy.

Page No.
Date

## * Variable:-

- JS variables are dynamically typed i.e. we can change the type of variable during run time.

- Variable is a container that stores a value.

**Rules:-**

1) letters, digits, underscores and $ sign allowed
2) Must begin with a $, _ or a letter.
3) JavaScript reserved words cannot be used as a variable name.
4) variables are case sensitive.

## * var vs let vs const

**var**
- Used before ES6
- but after ES6 can bring bugs
  ↓
  globally scoped.

**let**
- let can be updated but not re-declared.
  ↓
  blocked scoped.

**const**
- Const can neither be updated nor be re-declared.
  ↓
  Const must be initialized during declaration.

## * Primitive Data Types :-

- Primitive data types are a set of basic data types in js.

- There are 7 primitive datatypes :-

| | | |
|---|---|---|
| 1) Null<br>2) Number | 3) String<br>4) Symbol | 5) Boolean<br>6) BigInt |
| 7) Undefined. | | |

- Object is a non-primitive datatype :-

An object in js can be created as follows:-

```
const item = {
    key ← name : "Bob", → value.
        age : 25
}
```

- To find type of any variable :-

console.log (typeof variable_name)

## * Expressions & Conditionals :-

A fragment of code that produces a value is called an expression.

Operators :-

1) Arithmetic Operators :

+ , - , * , ** , / , % , ++ , --

pre & post

"==" => Equal to

"===" => Equal value & type.

Page No.
Date

2) Assignment Operators:

=, +=, -=, *=, /=, %=, **=

3) Comparison Operators:

==, !=, ===, !==, >, <, >=, <=, ?:

4) Logical Operators:

&&, ||, !

Apart from these, we also have type and bitwise operators. Bitwise operators perform bit by bit operators on numbers.

Eg:- c = a + b

operator

stores result

operands

* Comments in JavaScript:-

```
// This is a single line comment.

/*
  This is a
  multi-line
  comment
*/
```

Page No.
Date

## * Conditional Statements:-

1) if statement

```
if (condition) {
        // execute this code
}
```

2) if-else statement

```
if (condition) {
        // block of code if condition is true
}
else {
        // block of code if condition is false
}
```

3) if-else if-else statement

```
if (condition) {
        // if condition true.
}
else if (condition) {
        // else second condition true.
}
else {
        // finally if all conditions are false.
}
```

4) ternary operator

```
condition ? exp1 : exp2
```

Page No.
Date

5) Switch case

```
switch (case value variable) {
    case case_name :
        // statements
        break;
    ⋮
    ~~cas~~ default :
        // default statement
}
```

* Loops in ~~JAVASCRIPT~~ JAVASCRIPT :-

1) for loop

```
for (statement 1 ; statement 2 ; statement 3) {
    // code to be executed
}
```

→ statement 1 is executed one time.
→ statement 2 is the condition based on which the loop runs (loop body is executed)
→ statement 3 is executed everywhere the loop body is executed.

2) for-in loop ⇒ loops through the keys of an object.

```
for (key in object) {
    // code to be executed
}
```

Page No.
Date

3) <u>for-of loop</u> ⇒ loops through the values of an object.

```
for (variable of iterable) {
    // code
}
```

4) <u>while loop</u>:

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

→ If the condition never becomes false, the loop will never end and this might crash the runtime.

5) <u>do-while loop</u>:

```
do {
    // code   → executed at least once
}
while (condition)
```

* break

* continue.

Page No.
Date

## * Functions in JAVASCRIPT :-

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

```
function func_name () {
        // code
}

function func_name (Para1, Para2, ....) {
        // Parameters acts like local variables.
}
```

⇒ Function Invocation is a way to use the code inside the function.

⇒ Arrow function :

```
const variable = (a, b) => {
        // code using a & b.
        return value
}
```

## * Strings :- Strings are used to store and manipulate text.

```
let str = "Hello"   ← double quote
(OR)
let str = 'Hello'   ← single quote
(OR)
let str = `Hello`   ← template literals
```

→ string interpolation.

(i) can use " " or ' ' inside strings
(ii) can use other variables using ${}.

Page No.
Date

⇒ Escape characters :- (\) back slash.

\n ⇒ new line.
\t ⇒ tab

⇒ String Methods :-

① length ⇒ str.length.

② toUpperCase () ⇒ str.toUpperCase ()

③ toLowerCase () ⇒ str.toLowerCase ()

④ slice () ⇒ str.slice (2, 4)

⑤ replace () ⇒ str.replace ("Hello", "Hi")

⑥ concat () ⇒ str.concat (str1, "Hi")
↳ we can even use '+' operator.

⑦ trim () ⇒ str.trim () // removes whitespace.

⇒ Strings are immutable.

⑧ includes ⇒ str.includes ('Hello')

⑨ startsWith ⇒ str.startsWith ('H')

⑩ endsWith ⇒ str.endsWith ('o')

Page No.
Date

## * Arrays :-

→ Arrays are variables which can hold more than one value. ⇒ arr = [1, 2, 3]

→ Arrays can hold more than one value.
↳ arr = [true, 13, "4"]

→ Accessing values ⇒ arr[0], .....

→ Length of array ⇒ arr.length.

→ Changing the value ⇒ arr[4] = 14 ⇒ Arrays are mutable.

NOTE → In JS, arrays are object.

### ⇒ Arrays Method :-

1. toString() ⇒ arr.toString()
2. join() ⇒ arr.join(" ")
3. * pop() ⇒ arr.pop() → returns popped value.
4. ** push() ⇒ arr.push(value) → returns array length
5. * shift() ⇒ arr.shift()
6. ** unshift() ⇒ arr.unshift(value)
7. delete operator ⇒ delete arr[index] → doesn't change arr length.

Page No.
Date

⑧ concat() ⇒ arr1.concat(arr2, arr3)
↳ returns joined array.

let compare = (1) => {
return a-b
}
→ compare function. ⇒ arr.sort(compare)

⑨ sort() ⇒ arr.sort() → modifies original array.

⑩ splice() ⇒ arr.splice(position to add, no. of elements remove, elements to be added)
↙ returns deleted value.

⑪ slice() ⇒ arr.slice(start, end)
~~arr.slice~~

⑫ reverse() ⇒ arr.reverse()

⇒ **Loops with Array:-**

① **for loop:**

```
for (let i=0 ; i < arr.length ; i++) {
        console.log(num[i])
}
```

② **forEach loop:**

```
arr.forEach((element) => {
        console.log(element * element)
}
```

③ Array.from(object_name)

Page No.
Date

④ <u>for-of loop</u>:

```
for (let i of arr) {
        console.log(i)
}
```

⑤ <u>for-in loop</u>: ⇒ returns key of an array i.e. index.

```
for (let i in arr) {
        console.log(i)
}
```

* <u>map()</u> ⇒ creates a new array by performing some operation on each array element.

<u>NOTE</u> ⇒ map creates new array while forEach only performs operation in existing array.

```
const a = [1,2,3]
a.map((value, index, array) => {
        return value * value;
})
```

* <u>filter()</u> ⇒ filters an array with values that passes a test. Creates a new array.

```
const a = [1, 2, 3, 4, 5]
a.filter((value) => {
        return value > 3;
})
```

Page No.
Date

* reduce() ⇒ Reduces an array to a single value.

```
const a = [1, 2, 8, 11, 7]
~~a.reduce((value) => {~~
let sum = a.reduce((v1, v2) => {
        return v1 + v2
    })
```

Question No. 01 ⇒ Guess the Number

Write a javascript program to generate a random number and store it in a variable. The program then takes an input from the user to tell them whether the guess was correct, greater or lesser than the original number.

Generate ~~Sco~~ score as "100 - (no. of guess)"

The program is expected to terminate once the number is guessed.
~~Ra~~ Random Number should be between 1-100.

HINT ⇒

```
Math.floor(Math.random() * 100) + 1;
```

Page No.
Date

## * Console Object Methods :-

log() → outputs a message to the console.
error()
warn()
info()
assert() → used to assert a condition.
clear()
table()
time()
timeEnd()

---

**Answer :-**

```
let number = Math.floor((Math.random() * 100) + 1);
let chances = 0
let guess;

console.log(" ..... ");
guess = prompt("Enter a number from 1 to 100");
chances ++;

do {
    guess = Number.parseInt(guess);
    if (guess > number) {
        // Number is greater
        console.log("Number entered is greater");
        guess = prompt("Enter again!")
        guess = Number.parseInt(guess);
        ++chances;
        continue;
    }
```

```
else if (guess < number) {
    console.log ("Number entered is smaller");
    guess = prompt(" Enter again!!")
    guess = Number.parseInt (guess);
    ++ chances;
    continue;
}

} while (guess != number);

let score = 100 - chances;
console.log (" Congrats, The number generated
    was ${number} and you guessed it
    right . Your final score is ${score}");
```

---

* alert() ⇒ Used to invoke a mini window with a msg.

* prompt() ⇒ used to take user input as string.

* confirm() ⇒ shows a message and waits for the user to press OK or Cancel.
  Returns true for OK and false for cancel.

The exact location & look is determined by the browser which is a limitation.

Page No.
Date

* **Window object :-** global object

* **Document Object Model (DOM) :-**

DOM represents the page content as HTML.

* **Browser Object Model (BOM) :-**

BOM represents additional objects provided by the browser (host environment) for working with everything except the document.

The functions alert / confirm / prompt are also a part of the BOM. e.g:- location.href = " "

* **Walking the DOM :-**

DOM tree refers to the HTML page where all the nodes are objects. There can be 3 main types of nodes in the DOM tree :

1. ~~tree~~ text nodes
2. element nodes
3. comment nodes

- In an HTML page, <html> is at the root and <head> and <body> are its children, etc.
- A text node is always a leaf of the tree.

Page No.
Date

## * Children of an element :-

Direct as well as deeply nested elements of an element are called its children.

Child nodes → Elements that are direct children.

Descendant nodes → All nested elements, children, their children and so on.

ele.firstChild
ele.lastChild
ele.childNodes ⇒ returns an object.

TRUE { ele.childNodes[0] === ele.firstChild
ele.childNodes[ele.childNodes.length - 1] === ele.lastChild.

Can use array.from method on childNodes object to convert it into an array.

## * DOM collections :-

1. They are read-only.
2. They are live ele.childNodes variable will automatically update if childNodes of ele is changed.
3. They are iterable using for...of loop.

Page No.
Date

* **Siblings and the parent :-**

Siblings are nodes that are children of the same parent.

Eg:- <head> and <body> are siblings.

<body> is said to be the "next" or "right" ~~sb~~ sibling of <head>
<head> is said to be the "previous" or "left" sibling of <body>

The next sibling is in "nextSibling" property and the previous one ~~is~~ in "previousSibling".

The parent is ~~avlable~~ available as "parentNode" or "parentElement".

* **Element only Navigation :-**

document.previousElementSibling.

document.nextElementSibling.

document.firstElementChild

document.lastElementChild.

Page No.
Date

## * Table navigation :-

Certain DOM elements may provide additional properties specific to their type for convenience.

Table element support the following properties :- table = document.body.firstElementChild.

1) table.rows // collection of tr elements
2) table.caption. // reference to <caption>
3) table.tHead // reference to <thead>
4) table.tFoot // reference to <tfoot>
5) table.tBodies. // collection to <tbody> elements
6) tbody.rows. // Collection of <tr> inside,
7) tr.cells // Collection of td and th.
8) tr.sectionRowIndex. // Index of tr inside enclosing element
9) tr.rowIndex. // Row number starting from 0
10) td.cellIndex. // no. of cells inside enclosing <tr>

## * Searching the DOM :-

DOM navigation properties are helpful when the elements are close to each other. If they are not close to each other, we have some more methods to search the DOM.

1) document.getElementById(" ")

This method is used to get the element with a given "id" attribute.

Page No.
Date

2) document.getElementsByName(" ")

Searches elements by the name attribute.

3) document.getElementsByTagName(" ")

Returns element with the given tag name.

4) document.getElementsByClassName(" ")

Returns elements that have the given CSS class.

5) document.querySelector

Returns the first element for the given CSS selector. A efficient version of elem.querySelectorAll(CSS)[0].

6) document.querySelectorAll

Returns all elements inside an element matching the given CSS selector.

* There are three important methods to search the DOM :-

① elem.matches(CSS) :- To check if element matches the given CSS selector.

Page No.
Date

② elem.closest(CSS) :- To look for the nearest ancestor that matches the given CSS-selector. The elem itself is also checked.

③ elemA.contains(elemB) :- Returns true if elemB is inside elemA (a descendant of elemA) or when elemA == elemB.

* console.dir() :-

- console.log shows the element ~~DOM tree~~.
- console.dir shows the element as an object with its properties.

* tagName / nodeName :-

Used to read tag name of an element.
- tagName → only exists for Element nodes.
- nodeName → defined for any node.

* innerHTML / outerHTML :-

- The innerHTML property allows to get the HTML inside the element as a string (valid for element nodes only).
- The outerHTML property contains the full HTML, innerHTML + the element itself.
- innerHTML is valid only for element nodes. For other node types we can use nodeValue or data.

Page No.
Date

* textContent ⇒ Provides access to the text inside the element: only text, minus all tags.

* The hidden property ⇒

The "hidden" ~~proper~~ attribute ~~and~~ in the DOM property specifies whether the element is visible or not.

elem.hidden.

* Attribute Methods :-

① elem.hasAttribute(name)

Method to check for existence of an attribute.

② elem.getAttribute(name)

Method used to get the value of an attribute.

③ elem.setAttribute(name, value)

Method used to set the value of an attribute

④ elem.removeAttribute(name)

Method to remove the attribute from elem.

⑤ elem.attributes

Method to get the collection of all attributes.

Page No.
Date

## * data-* attributes :-

- We can always create custom attributes but the ones starting with "data-" are reserved for programmers use.
- They are available in a property named dataset.
- If an element has an attribute named "data-one", its available as elem.dataset.one.

## * Insertion Methods :-

① using innerHTML.

② Steps :-

ⓐ create a element using.
let div = document.createElement('div')

ⓑ set the attributes and innerHTML.
div.className = "alert"
div.innerHTML = " Hi 

ⓒ append the created element.
document.body.append(div)

Methods :-

① append( ) ⇒ append at the end of node.
② prepend( ) ⇒ insert at the beginning of node.

Page No.
Date

③ before() ⇒ insert before node.
④ after() ⇒ insert after node.
⑤ replaceWith() ⇒ replaces node with the given node.

* **Insert Adjacent HTML / Text / Element :-**

insertAdjacentHTML / Text / Element (' ', ' ')

This method is used to insert HTML. The first parameter is a code word, specifying where to insert.

① beforebegin ⇒ insert HTML immediately before element.
② beforeend ⇒ insert HTML into element at the end.
③ afterbegin ⇒ insert HTML into element at the beginning.
④ afterend ⇒ insert HTML immediately after element.

The second parameter is an HTML string.

* **Node removal :-**

To remove a node, there's a method node.remove()

* **className** ⇒ changes class of HTML tags.

* **classList** ⇒ list of all classes of that tag.
  - add ⇒ adds a class.
  - remove ⇒ removes a class.
  - toggle ⇒ adds the class if it doesn't exist, otherwise removes it.

Page No.
Date

* elem.classList.contains("class")
  → checks for the given class, returns true/false.

* <u>setTimeout() function</u> :-

setTimeout allows us to run a function once after the interval of time.

<u>Syntax</u>:-

```
let timerId = setTimeout(function,
                         <delay> (in ms),
                         <arg1>, <arg2>)
```

⇒ clearTimeout(timerId) is used to cancel the execution (in case we change our mind).

* <u>setInterval() function</u> :-

unlike setTimeout, setInterval runs the function not only once, but regularly after the given interval of time.

<u>Syntax</u>:- same as setTimeout.

⇒ clearInterval(intervalId) is used to stop the execution.

Page No.
Date

## * Browser Events:-

An event is a signal that something has happened. All the DOM nodes generates such events / signals.

⇒ Important DOM events:-

1) Mouse events → click, context menu (right click), mouseover / mouseout, mousedown / mouse up, mousemove.

2) Keyboard events → keydown, keyup.

3) form elements events → submit, focus, etc

4) Document events → DOMContentLoaded.

## * Handling events:-

1) Events can be handled through HTML attributes.

<input value = "Hey" onclick = "alert('hey')">

Note: instead of alert(), we can write another JS function.

2) Events can also be handled through the onclick property.

```
elem.onclick = function() {
                alert("hey")
}
```

**Note:-** Adding a handler with JS overwrites the existing handler.

## * addEventListener :- & removeEventListener :-

addEventListener is used to assign multiple handlers to an event.

```
elem.addEventListener(event, handler)
elem.removeEventListener(event, handler)
```

**Note:-** handler must be the same function object for this to work.

## * Event Object :-

When an event happens, the browser creates an event object, puts details into it and passes it as an argument to the handler.

```
elem.onclick = (e) => {
    ...
}
```

e.target.
e.type.
e.clientX } mouse X
e.clientY } & Y coords
e.currentTarget.

Page No.
Date

## * Introduction to Callback :-

⇒ Asynchronous actions are the actions that we initiate now and they finish later.
E.g: setTimeout.

⇒ Synchronous actions are the actions that initiate and finish one-by-one.

## * Callback function:

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

eg:-
```
const loadScript = (src, callback) => {
    let script = document.createElement('script')
    script.src = src.
    script.onload = () => callback(script)
    document.head.append(script)
}
loadScript("some script", (script) => {
    alert('script is loaded')
});
```

This is called ~~"callbased~~ "callback-based" style of async programming. A function that does something asynchronously should provide a callback argument where we put the function to run after its complete.

Page No.
Date

## * Handling errors :-

```
function loadScript (src, callback) {
    ...
    ...
    script.onload = () => callback(null, script);
    script.onerror = () => callback (new
                                Error('failed'));
    ...
}
loadScript ('some script', (error, script) => {
    if (error) {
        // handle error
    } else {
        // script loaded
    }
});
```

## * ~~Callback hell~~ Disadvantage :-

1. Here we pass whole callback function with error, so anyone can use error as he wishes.

   To resolve this we use promises.

2. "Callback hell" or "Pyramid of Doom" :-

   When we have callback inside callback, the code gets difficult to manage.

```
~~loo~~ loadScript((...){
        loadScript ..          // Pyramid
            loadScript ...        of Doom.
                loadScript ...
                ...
```

This can also be resolved using promises.

## * Introduction to Promises :-

A promise is a "promise of code execution". The code either executes or fails, in both the cases the subscriber will be notified.

Syntax:

```
let promise = new Promise ((resolve, reject) => {
        // executor
});
// here resolve & reject both are predefined
                                in JS engine.
```

resolve & reject are two callbacks provided by JS itself. They are called like this :-

resolve (value) => If the job is finished successfully.

reject (error) => If the job fails.

Page No.
Date

⇒ The promise object returned by the new Promise constructor has these properties:

① <u>state</u> :- Initially pending, then changes to either "fulfilled" when resolve is called or "rejected" when reject is called.

② <u>result</u> :- Initially undefined, then changes to value if resolved (resolve(value)) or error when rejected (reject(~~value~~ error)).

* <u>then()</u> & <u>catch()</u> :-

```
prom.then( function (result) {
                // handle if resolved.
           }, function (error) {
                // handle if rejected.
           }
);
```

~~prom.catch~~

If we are interested only in successful completions we can provide only one function argument to .then().

```
prom.then(alert);
```

If we are interested only in errors, we can use null as the first argument:

```
prom.then(null, alert)
```

or we can use catch :-

```
prom.catch(alert)
```

Note: prom.finally(() => { }) is used to perform general cleanups.

## * Promise Chaining .then() calls :-

We can chain promises and make them pass the resolved values to one another like this :-

```
p.then(function(result) => {
        alert(result);
        return 2;
    }).then .....
```

The flow of execution:

1. The initial Promise resolves in 1 sec (assume)
2. The next .then() handler is then called, which returns a new promise (resolved with 2)
3. The next .then() gets the result of previous one and this keeps on going.

Page No.
Date

## * loadScript using Promise :-

```
const loadScript = (src) => {
    return new Promise ((resolve, reject) => {
        let script = docu. creteElem("script")
        script.src = src.
        script.type = "text/javascript"
        document.body. appenchild (script)
        script. onload = () => { resolve(1)}
        script. onerror = () => { reject(0)}
    })
}

let p1 = loadScript ("http://localhost:3000/index.js")

p1. then ((value) => {
        console. log(value)
}). catch ((error) => {
        console. log (error)
})
```

## * Promise API :-

⇒ methods of Promise class :-

1) Promise. all (promises)

Waits for all promises to resolve and returns the array of their results.
If any one fails it becomes the error and all other results are ignored.

2) Promise.allSettled(promises) ⇒ returns status & value.

3) Promise.resolve(value)

4) Promise.reject(value)

5) Promise.any(promises)

Waits for the first promise to fulfill, and its result becomes the outcome.

6) Promise.race(promises)

If error, then stops.

## * Async / Await :-

⇒ A function can be made async by using "async" keyword like :

```
async function name() {
    return "Hello"
}
```

⇒ An async function always returns a promise. we can do :

```
name().then(alert)
```

⇒ So, async ensures that the function returns a promise and wraps non-promises in it.

Page No.
Date

⇒ "await" keyword only works inside async functions.

```
let value = await promise;
```

⇒ The await keyword makes JS wait until the promise settles and returns its value.

## * Error Handling :-

We all make mistakes. Also, our script can have errors. Usually, a program halts when an error occurs.

The try...catch syntax allows us to catch errors so that the script instead of dying can do something more reasonable.

Eg:
```
try {
    // try the code
}
catch (error) {
    // handle the error
}
```
error object

NOTE :- try catch works synchronously.

```
try {
    setTimeout (function () {
        // error code
    }
}
catch ...
```
// script dies & catch won't work

Page No.
Date

## * Error object :-

while catching the error we use an object for that error.

the built-in object has two main properties:
① err. name
② err. message.

## * Explicitly throw custom error :-

Eg: throw new Error(" ")
↓ err. name
↓ err. message.

## * Finally clause

---

## * Fetch API :-

fetch is used to get data over the network.

let promise = fetch (url, [options])
↑
without options, a get request is send.

⇒ Getting a response is a 2-stage process:

① An object of Response class containing "status" & "ok" properties
status → The http status code. Eg: 200.
ok → Boolean, true if the HTTP status code is 200-299

Page No.
Date

② After that we need to call another method to access the body in different formats :

response.text() → Read & return the text.

response.json() → parse the response as JSON.

Other methods include response.formData(), response.blob(), response.arrayBuffer(), etc.

NOTE ⇒ We can use only one body reading method.

Syntax:

```
let p = fetch("      ")

p.then((response) => {
        return response.json()
}).then((value) => {
        console.log(value)
})
```

⇒ Response headers are available in response.headers

⇒ Request headers

To set request header in fetch, we use the header option.

```
let res = fetch(url, {
        headers: {
            Authentication: 'Secret'
        }
});
```

Page No.
Date

## * POST request with Fetch API :-

To make a POST request, we need to use fetch options.

1. method → HTTP-method, e.g. POST.
2. body → the request body.

```
let response = await fetch('/url', {
                    method : 'POST',
                    headers : {
                        'Content-type' : 'application/json'
                    },
                    body : {
                        'a' : 'John'
                    }

let result = await response.json()
```

Eg:-

```
const createTodo = async () => {
    let options = {
        method : "POST"
        headers : {"Content-type" : "application/json"}
        body : JSON.stringify({
                title : 'Bob',
                body : '',
                userId : 1100,
            }),
    }
    let p = await fetch('jsonplaceholder', options)
    let response = await p.json()
    return response
}
```

Page No.
Date

## * JavaScript Cookies : "document.cookie"

⇒ Cookies are small strings of data stored directly in the browser.

⇒ Cookies are set by a web server using the Set-Cookie HTTP-header. Next time when the request is sent to the same domain, the browser sends the cookie using the cookie HTTP-header.
That way the server knows who sent the request.

⇒ Writing to cookie (doesn't overwrite old cookies)

document.cookie = "name = John"

⇒ encodeURIComponent()

This way, the special characters are encoded.

NOTE : To decode use decodeURIComponent()

## * Cookie options

path = /a ; ⇒ path option makes the cookie visible at /a , /a/b , etc.

expires = Tue, 29 July 2024 03:18:22 GMT

expires sets the cookie expiration time.

Page No.
Date

Note :-

① The name = value pair, after encodeURIComponent should not exceed 4 kB.

② Total no. of cookies per domain is limited to around 20+ (Exact number is browser dependent).

## * Local Storage :-

⇒ localStorage is a web storage object which are not sent to server with each request.

⇒ This data survives a full page refresh and even a full browser restart.

⇒ Methods in localStorage:

① setItem (key, value) - Store key/value pair

② getItem (key) - get the value by key.

③ removeItem (key) - remove the key with its value.

④ clear() - delete everything.

⑤ key (index) - get the key on a given position.

⑥ length - the number of stored items.

Page No.
Date

⇒ We can get & set values like an object.

```
localStorage.one = 1
alert (localStorage.one)
delete localStorage.one.
```

⇒ NOTE:-

① Both key & value must be strings.

② We can use the two JSON methods to store objects in localStorage:

JSON.Stringify (object) ⇒ Converts objects to JSON Strings.

JSON.parse (String) ⇒ Converts string to objects (must be valid JSON)

* Session Storage :-

⇒ Used less often than local Storage, Properties and methods are same as localStorage but:-

① The sessionStorage exists only within the current browser tab. Another tab with same page will have a different storage.

② The data survives page refresh, but not closing / opening the tab.

Page No.
Date

## * Storage Event :-

When the data gets updated in localStorage or sessionStorage, storage event triggers with these properties:

1. ~~The~~ key → The Key.
2. newValue → New Value.
3. oldValue → Previous Value.
4. url → Page URL.
5. StorageArea → local or session.

⇒ We can listen the onstorage event of window which is triggered when updates are made to the same storage from the documents.

## * Introduction to Object-Oriented Programming :-

In programming we often take something and then extend it.

Eg: we might want to create a user object and "admin" and "guest" will be slightly modified variants of it.

⇒ [[Prototype]] : JS object have a special property called prototype that is either null or references another object.

When we try to read a property from a prototype and its missing, JS automatically takes it from the prototype. This is called "prototype inheritance"

Page No.
Date

* OOPS has 4 pillars :-

① Abstraction : Hidding of internal details.

② Encapsulation : Using of simple items to create a complex entity.

③ Inheritance : Child inherits the property from its parent.

④ Polymorphism : One thing many forms.

* Setting prototype :-

We can set prototype by setting --proto--

Eg:-
```
let a = {
    name : "",
    age :  ,
}
let p = {
    run : () => {
        alert("run")
    }
}
a.__proto__ = p
a.run()
```

Page No.
Date

## * Classes and Objects :-

⇒ In oops, a class is an extensible program-code template for creating objects, providing initial values for state (member variables) and implementation of behavior (member functions)

⇒ Basic syntax:

```
class MyClass {
    // class methods
    constructor() {...}
    method1() {...}
}
```

We can then use "new MyClass()" to create a new object with all the listed methods.

* "this" keyword is used in class to get that objects value.

## * Constructors :-

The constructor() method is called automatically by new, so we can initialize the object there.

* Inheritance is used using "extends" keyword in javascript.

eg: class Animal. {...}
class Monkey extends Animal {...}

## * super keyword :-

When we override a method, we don't want the method of the previous class to go in vain. We can execute it using "super" keyword.

Page No.
Date

* **Method Overriding:**

If we create own implementation of run, it will not be taken from the Animal class. This is called Method overriding.

* Inherited class generates constructor automatically.

```
# auto-generated.
constructor (...) {
        super (...)
}
```

* **Static method :-**

Static method are used to implement functions that belong to a class as a whole and not to any particular object.

Eg:
```
class Employee {
        static sMethod() {
                alert ("Hey");
        }
}
Employee.sMethod()
```
⇐ static method is called using class.

* **Getters & Setters :-** Classes may include getters & setters to get & set the computed properties.

```
class Person {
    ...
getter ⇒ get name() { return this.name; }
setter ⇒ set name (newName) {
            this.name = newName;
    }
}
```

Page No.
Date

## * instance of Operator :

The instance of operator allows to check whether an object belongs to a certain class.

Syntax: <obj> instance of <class>

It returns true if obj belongs to the class or any other class inheriting from it.

---

## * IIFE :- (Immediately Invoked Function Expression)

It is a js function that runs as soon as it is defined.

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

(OR)

```
(() => {
    ...
})();
```

It is used to avoid polluting the global namescape, execute an async-await, etc.

---

## * Destructuring assignment :-

It is used to unpack values from an array or properties from objects, into distinct variables.

Eg:

```
let arr = [3, 5]
let [a, b] = arr
```

(OR)

```
let arr = [5, 10, 9, 11, 13]
let [a, b, ...rest] = arr
```

(OR)

```
let [a, , ...rest] = arr.
```

Page No.
Date

## * Spread Syntax:

Spread syntax allows an iterable such as an array or String to be expanded in places where zero or more arguments are expected. In an object literal, the spread syntax enumerates the properties of an object and adds the key-value pairs to the object being created.

Eg:- ① const arr = [1, 7, 11]
const obj = { ... arr } // {0:1, 1:7, 2:11}

② const nums = [1, 2, 7]
console.log (sum(...nums)) // 10.

③ let obj = {
name: " "
company: " "
address: " "
}
console.log({ ...obj, name = " " })

always before changing.

Page No.
Date

# * Regular Expression :-

→ A regular expression is a pattern of characters.
→ The pattern is used for searching and replacing characters in strings.

Syntax:- /pattern/modifier(s);

⇒ Modifiers:

① g ⇒ perform a global match (all occurances)
② i ⇒ case-insensitive match.
③ m ⇒ multiline match.

⇒ Brackets:

① [abc] ⇒ Find any character between the brackets
② [^abc] ⇒ not ——— " ———
③ [0-9] ⇒ range
④ [^0-9] ⇒ not in range.
⑤ (x|y) ⇒ 'x' or 'y'.

⇒ Metacharacter:

① . ⇒ single character.
② \w ⇒ words.
③ \W ⇒ non-words
④ \d ⇒ digit.
⑤ \D ⇒ non-digit
⑥ \s ⇒ whitespace
⑦ \S ⇒ non-whitespace

| | |
|---|---|
| ^abc$ | ⇒ start (^) / end ($) |
| \b | ⇒ word boundary. |
| \B | ⇒ non-word boundary. |
| a*, a+, a? | ⇒ 0 or more, 1 or more, 0 or 1. |
| a{5}, a{2,}, a{1,3} | exactly 5, two or more, bet 1 to 3. |
| a+?, a{2,}? | match as few as possible. |